C#

**Especificación del lenguaje**

**Versión 5.0**

**Aviso**

© 1999-2012 Microsoft Corporation. Reservados todos los derechos.

Microsoft, Windows, Visual Basic, Visual C# y Visual C++ son marcas registradas o marcas comerciales de Microsoft Corporation en los EE.UU. y/o en otros países o regiones.

Los demás productos y nombres de compañías mencionados en esta publicación pueden ser marcas comerciales de sus respectivos propietarios.

**Tabla de contenido**

1. Introducción 1

1.1 Hola a todos 1

1.2 Estructura del programa 2

1.3 Tipos y variables 4

1.4 Expresiones 8

1.5 Instrucciones 10

1.6 Clases y objetos 14

1.6.1 Miembros 14

1.6.2 Accesibilidad 15

1.6.3 Parámetros de tipo 15

1.6.4 Clases base 16

1.6.5 Campos 16

1.6.6 Métodos 17

1.6.6.1 Parámetros 17

1.6.6.2 Cuerpo del método y variables locales 19

1.6.6.3 Métodos estáticos y de instancia 19

1.6.6.4 Métodos virtuales, de invalidación y abstractos 20

1.6.6.5 Sobrecarga de métodos 22

1.6.7 Otros miembros de función 23

1.6.7.1 Constructores 25

1.6.7.2 Propiedades 26

1.6.7.3 Indizadores 26

1.6.7.4 Eventos 27

1.6.7.5 Operadores 27

1.6.7.6 Destructores 28

1.7 Structs 28

1.8 Matrices 29

1.9 Interfaces 31

1.10 Enumeraciones 32

1.11 Delegados 33

1.12 Atributos 34

2. Estructura léxica 37

2.1 Programas 37

2.2 Gramáticas 37

2.2.1 Notación gramatical 37

2.2.2 Gramática léxica 38

2.2.3 Gramática sintáctica 38

2.3 Análisis léxico 38

2.3.1 Terminadores de línea 39

2.3.2 Comentarios 40

2.3.3 Espacio en blanco 41

2.4 Tokens 41

2.4.1 Secuencias de escape de caracteres Unicode 41

2.4.2 Identificadores 42

2.4.3 Palabras clave 44

2.4.4 Literales 45

2.4.4.1 Literales booleanos 45

2.4.4.2 Literales enteros 45

2.4.4.3 Literales reales 46

2.4.4.4 Literales de carácter 47

2.4.4.5 Literales de cadena 48

2.4.4.6 El literal null 50

2.4.5 Operadores y signos de puntuación 50

2.5 Directivas de preprocesamiento 50

2.5.1 Símbolos de compilación condicional 52

2.5.2 Expresiones de preprocesamiento 52

2.5.3 Directivas de declaración 53

2.5.4 Directivas de compilación condicional 54

2.5.5 Directivas de diagnóstico 56

2.5.6 Directivas de región 57

2.5.7 Directivas de línea 57

2.5.8 Directivas pragma 58

2.5.8.1 Pragma warning 58

3. Conceptos básicos 61

3.1 Inicio de la aplicación 61

3.2 Finalización de la aplicación 62

3.3 Declaraciones 62

3.4 Miembros 65

3.4.1 Miembros de espacio de nombres 65

3.4.2 Miembros de struct 65

3.4.3 Miembros de enumeraciones 66

3.4.4 Miembros de clase 66

3.4.5 Miembros de interfaz 66

3.4.6 Miembros de matriz 66

3.4.7 Miembros de delegados 66

3.5 Acceso a miembros 66

3.5.1 Accesibilidad declarada 66

3.5.2 Dominios de accesibilidad 67

3.5.3 Acceso protegido para miembros de instancia 70

3.5.4 Restricciones de accesibilidad 71

3.6 Firmas y sobrecargas 72

3.7 Ámbitos 73

3.7.1 Ocultar nombres 76

3.7.1.1 Ocultar mediante anidación 77

3.7.1.2 Ocultar mediante herencia 77

3.8 Espacios de nombres y nombres de tipos 78

3.8.1 Nombres completos 81

3.9 Administración automática de la memoria 82

3.10 Orden de ejecución 85

4. Tipos 87

4.1 Tipos de valor 87

4.1.1 Tipo System.ValueType 88

4.1.2 Constructores predeterminados 88

4.1.3 Tipos struct 89

4.1.4 Tipos simples 89

4.1.5 Tipos enteros 90

4.1.6 Tipos de punto flotante 92

4.1.7 Tipo decimal 93

4.1.8 El tipo booleano 93

4.1.9 Tipos de enumeración 94

4.1.10 Tipos que aceptan valores null 94

4.2 Tipos de referencia 94

4.2.1 Tipos de clase 95

4.2.2 Tipo object 96

4.2.3 Tipo dinámico 96

4.2.4 Tipo string 96

4.2.5 Tipos de interfaz 96

4.2.6 Tipos de matriz 96

4.2.7 Tipos delegados 96

4.3 Conversiones boxing y unboxing 96

4.3.1 Conversiones boxing 97

4.3.2 Conversiones Unboxing 98

4.4 Tipos construidos 99

4.4.1 Argumentos de tipo 100

4.4.2 Tipos cerrados y abiertos 100

4.4.3 Tipos enlazados y sin enlazar 101

4.4.4 Cumplimiento de las restricciones 101

4.5 Parámetros de tipo 102

4.6 Tipos de árbol de expresiones 103

4.7 Tipo dinámico 103

5. Variables 105

5.1 Categorías de variables 105

5.1.1 Variables estáticas 105

5.1.2 Variables de instancia 105

5.1.2.1 Variables de instancia en clases 105

5.1.2.2 Variables de instancia en structs 106

5.1.3 Elementos matriciales 106

5.1.4 Parámetros de valor 106

5.1.5 Parámetros de referencia 106

5.1.6 Parámetros de salida 107

5.1.7 Variables locales 107

5.2 Valores predeterminados 108

5.3 Estado de asignación definitiva 108

5.3.1 Variables asignadas inicialmente 109

5.3.2 Variables no asignadas inicialmente 109

5.3.3 Reglas precisas para determinar asignaciones definitivas 110

5.3.3.1 Reglas generales para instrucciones 110

5.3.3.2 Instrucciones de bloques e instrucciones checked y unchecked 111

5.3.3.3 Instrucciones de expresiones 111

5.3.3.4 Instrucciones de declaración 111

5.3.3.5 Instrucciones If 111

5.3.3.6 Instrucciones Switch 111

5.3.3.7 Instrucciones While 111

5.3.3.8 Instrucciones Do 112

5.3.3.9 Instrucciones For 112

5.3.3.10 Instrucciones Break, Continue y Goto 112

5.3.3.11 Instrucciones Throw 112

5.3.3.12 Instrucciones Return 113

5.3.3.13 Instrucciones Try-catch 113

5.3.3.14 Instrucciones try-finally 113

5.3.3.15 Instrucciones Try-finally-catch 114

5.3.3.16 Instrucciones Foreach 114

5.3.3.17 Instrucciones Using 115

5.3.3.18 Instrucciones Lock 115

5.3.3.19 Instrucciones yield 115

5.3.3.20 Reglas generales para expresiones simples 115

5.3.3.21 Reglas generales para expresiones con expresiones incrustadas 115

5.3.3.22 Expresiones de invocación y expresiones de creación de objetos 116

5.3.3.23 Expresiones de asignación simples 116

5.3.3.24 Expresiones && 116

5.3.3.25 Expresiones || 117

5.3.3.26 ! expresiones 118

5.3.3.27 Expresiones ?? 119

5.3.3.28 Expresiones ?: 119

5.3.3.29 Funciones anónimas 120

5.4 Referencias de variables 120

5.5 Atomicidad de las referencias de variable 120

6. Conversiones 121

6.1 Conversiones implícitas 121

6.1.1 Conversiones de identidad 122

6.1.2 Conversiones numéricas implícitas 122

6.1.3 Conversiones de enumeración implícitas 122

6.1.4 Conversiones implícitas que aceptan valores NULL 122

6.1.5 Conversiones del literal NULL 123

6.1.6 Conversiones de referencia implícitas 123

6.1.7 Conversiones boxing 124

6.1.8 Conversiones dinámicas implícitas 124

6.1.9 Conversiones implícitas de expresión constante 125

6.1.10 Conversiones implícitas con parámetros de tipo 125

6.1.11 Conversiones definidas por el usuario implícitas 126

6.1.12 Conversiones de función anónima y conversiones de grupo de métodos 126

6.2 Conversiones explícitas 126

6.2.1 Conversiones explícitas numéricas 126

6.2.2 Conversiones de enumeración explícitas 128

6.2.3 Conversiones explícitas que aceptan valores NULL 128

6.2.4 Conversiones explícitas de referencia 129

6.2.5 Conversiones Unboxing 130

6.2.6 Conversiones dinámicas explícitas 131

6.2.7 Conversiones explícitas con parámetros de tipo 131

6.2.8 Conversiones explícitas definidas por el usuario 132

6.3 Conversiones estándar 132

6.3.1 Conversiones implícitas estándar 132

6.3.2 Conversiones explícitas estándar 133

6.4 Conversiones definidas por el usuario 133

6.4.1 Conversiones permitidas definidas por el usuario 133

6.4.2 Operadores de conversión de elevación 133

6.4.3 Evaluación de conversiones definidas por el usuario 133

6.4.4 Conversiones explícitas definidas por el usuario 134

6.4.5 Conversiones explícitas definidas por el usuario 135

6.5 Conversiones de función anónima 137

6.5.1 Evaluación de conversiones de función anónima a tipos delegados 138

6.5.2 Evaluación de conversiones de función anónima a tipos delegados de árbol de expresiones 139

6.5.3 Ejemplo de implementación 139

6.6 Conversiones de grupo de métodos 141

7. Expresiones 145

7.1 Clasificaciones de expresión 145

7.1.1 Valores de expresiones 146

7.2 Enlace estático y dinámico 146

7.2.1 Tiempo de enlace 147

7.2.2 Enlace dinámico 147

7.2.3 Tipos de datos de expresiones constitutivas 148

7.3 Operadores 148

7.3.1 Prioridad y asociatividad de los operadores 149

7.3.2 Sobrecarga de operadores 150

7.3.3 Resolución de sobrecarga de operador unario 151

7.3.4 Resolución de sobrecarga de operador binario 151

7.3.5 Operadores candidatos definidos por el usuario 152

7.3.6 Promociones numéricas 152

7.3.6.1 Promociones numéricas unarias 153

7.3.6.2 Promociones numéricas binarias 153

7.3.7 Operadores de elevación 154

7.4 Búsqueda de miembros 154

7.4.1 Tipos base 156

7.5 Miembros de función 156

7.5.1 Listas de argumentos 159

7.5.1.1 Parámetros correspondientes 160

7.5.1.2 Evaluación en tiempo de ejecución de listas de argumentos 161

7.5.2 Inferencia de tipos 163

7.5.2.1 La primera fase 164

7.5.2.2 La segunda fase 164

7.5.2.3 Tipos de entrada 164

7.5.2.4 Tipos de resultado 164

7.5.2.5 Dependencia 164

7.5.2.6 Inferencias de tipo de resultado 164

7.5.2.7 Inferencias explícitas de tipo de parámetro 165

7.5.2.8 Inferencias exactas 165

7.5.2.9 Inferencias de límite inferior 165

7.5.2.10 Inferencias de límite superior 166

7.5.2.11 Fijar tipos 167

7.5.2.12 Tipo de resultado inferido 167

7.5.2.13 Inferencia de tipos para la conversión de grupos de métodos 168

7.5.2.14 Buscar el mejor tipo común de un conjunto de expresiones 169

7.5.3 Resolución de sobrecargas 169

7.5.3.1 Miembro de función aplicable 170

7.5.3.2 Mejor miembro de función 170

7.5.3.3 Mejor conversión de expresiones 171

7.5.3.4 Mejor conversión de tipos 172

7.5.3.5 Mejor destino de conversión 172

7.5.3.6 Sobrecargas en clases genéricas 172

7.5.4 Comprobación en tiempo de compilación de resolución de sobrecarga dinámica 173

7.5.5 Invocación de miembros de función 173

7.5.5.1 Invocaciones en instancias de conversión boxing 175

7.6 Expresiones primarias 175

7.6.1 Literales 176

7.6.2 Nombres sencillos 176

7.6.2.1 Significado invariable en bloques 177

7.6.3 Expresiones entre paréntesis 178

7.6.4 Acceso a miembros 178

7.6.4.1 Nombres simples y nombres de tipos idénticos 181

7.6.4.2 Ambigüedades gramaticales 181

7.6.5 Expresiones de invocación 182

7.6.5.1 Invocaciones de método 183

7.6.5.2 Invocaciones del método de extensión 184

7.6.5.3 Invocaciones de delegados 186

7.6.6 Acceso a elementos 187

7.6.6.1 Acceso a matrices 187

7.6.6.2 Acceso al indizador 188

7.6.7 Acceso a this 189

7.6.8 Acceso a bases 189

7.6.9 Operadores postfijos de incremento y decremento 190

7.6.10 El operador new 191

7.6.10.1 Expresiones de creación de objetos 191

7.6.10.2 Inicializadores de objeto 193

7.6.10.3 Inicializadores de colección 195

7.6.10.4 Expresiones de creación de matrices 196

7.6.10.5 Expresiones de creación de delegados 198

7.6.10.6 Expresiones de creación de objetos anónimos 200

7.6.11 Operador typeof 201

7.6.12 Los operadores checked y unchecked 203

7.6.13 Expresiones de valor predeterminadas 206

7.6.14 Expresiones de métodos anónimos 206

7.7 Operadores unarios 206

7.7.1 Operador unario de signo más 206

7.7.2 Operador unario de signo menos 207

7.7.3 Operador de negación lógica 207

7.7.4 Operador de complemento de bit a bit 208

7.7.5 Operadores prefijos de incremento y decremento 208

7.7.6 Expresiones de conversión 209

7.7.7 Expresiones await 210

7.7.7.1 Expresiones que admiten await 210

7.7.7.2 Clasificaciones de las expresiones await 211

7.7.7.3 Evaluación de expresiones await en tiempo de ejecución 211

7.8 Operadores aritméticos 211

7.8.1 Operador de multiplicación 212

7.8.2 Operador de división 213

7.8.3 Operador de resto 214

7.8.4 Operador de suma 215

7.8.5 Operador de resta 216

7.9 Operadores de desplazamiento 218

7.10 Operadores de comprobación de tipos y relacionales 219

7.10.1 Operadores de comparación de enteros 220

7.10.2 Operadores de comparación de punto flotante 221

7.10.3 Operadores de comparación decimales 222

7.10.4 Operadores de igualdad booleanos 222

7.10.5 Operadores de comparación de tipo de enumeración 222

7.10.6 Operadores de igualdad de tipos de referencia 222

7.10.7 Operadores de igualdad de cadenas 224

7.10.8 Operadores de igualdad de delegados 224

7.10.9 Operadores de igualdad y NULL 225

7.10.10 Operador Is 225

7.10.11 Operador As 226

7.11 Operadores lógicos 227

7.11.1 Operadores lógicos enteros 227

7.11.2 Operadores lógicos de enumeración 227

7.11.3 Operadores lógicos booleanos 228

7.11.4 Operadores lógicos booleanos que aceptan valores NULL 228

7.12 Operadores lógicos condicionales 228

7.12.1 Operadores lógicos condicionales booleanos 229

7.12.2 Operadores lógicos condicionales definidos por el usuario 229

7.13 El operador de incorporación de NULL 230

7.14 Operador condicional 231

7.15 Expresiones de funciones anónimas 232

7.15.1 Signaturas de función anónima 234

7.15.2 Cuerpos de función anónima 234

7.15.3 Resolución de sobrecargas 235

7.15.4 Funciones anónimas y enlace dinámico 236

7.15.5 Variables externas 236

7.15.5.1 Variables externas capturadas 236

7.15.5.2 Creación de instancias de variables locales 237

7.15.6 Expresiones de evaluación de funciones anónimas 239

7.16 Expresiones de consulta 239

7.16.1 Ambigüedad en expresiones de consulta 240

7.16.2 Traducción de expresiones de consulta 240

7.16.2.1 Cláusulas Select y Groupby con continuaciones 241

7.16.2.2 Tipos de variable de intervalo explícitos 241

7.16.2.3 Expresiones de consulta degeneradas 242

7.16.2.4 Cláusulas from, let, where, join y orderby 242

7.16.2.5 Cláusulas Select 246

7.16.2.6 Cláusulas Groupby 246

7.16.2.7 Identificadores transparentes 246

7.16.3 El patrón de expresiones de consulta 248

7.17 Operadores de asignación 249

7.17.1 Asignación simple 250

7.17.2 Asignación compuesta 252

7.17.3 Asignación de eventos 253

7.18 Expresión 253

7.19 Expresiones constantes 253

7.20 Expresiones booleanas 255

8. Instrucciones 257

8.1 Puntos finales y alcance 257

8.2 Bloques 259

8.2.1 Listas de instrucciones 259

8.3 Instrucción vacía 260

8.4 Instrucciones con etiqueta 260

8.5 Instrucciones de declaración 261

8.5.1 Declaraciones de variables locales 261

8.5.2 Declaraciones de constantes locales 263

8.6 Instrucciones de expresiones 263

8.7 Instrucciones de selección 264

8.7.1 Instrucción If 264

8.7.2 Instrucción Switch 265

8.8 Instrucciones de iteración 268

8.8.1 Instrucción While 269

8.8.2 Instrucción Do 269

8.8.3 Instrucción For 270

8.8.4 Instrucción Foreach 271

8.9 Instrucciones Jump 274

8.9.1 Instrucción Break 275

8.9.2 Instrucción continue 276

8.9.3 Instrucción Goto 276

8.9.4 Instrucción Return 277

8.9.5 Instrucción throw 278

8.10 Instrucción try 279

8.11 Instrucciones checked y unchecked 282

8.12 Instrucción lock 283

8.13 Instrucción using 283

8.14 La instrucción yield 286

9. Espacios de nombres 288

9.1 Unidades de compilación 288

9.2 Declaraciones de espacio de nombres 288

9.3 Alias extern 290

9.4 Directivas Using 290

9.4.1 Directivas de alias using 291

9.4.2 Directivas Using de espacio de nombres 294

9.5 Miembros de espacio de nombres 296

9.6 Declaraciones de tipo 296

9.7 Calificadores de alias de espacios de nombres 297

9.7.1 Unicidad de los alias 298

10. Clases 299

10.1 Declaraciones de clases 299

10.1.1 Modificadores de clase 299

10.1.1.1 Clases abstractas 300

10.1.1.2 Clases sealed 300

10.1.1.3 Clases estáticas 301

10.1.2 Modificador parcial 302

10.1.3 Parámetros de tipo 302

10.1.4 Especificación de clase base 302

10.1.4.1 Clases base 302

10.1.4.2 Implementaciones de interfaces 304

10.1.5 Restricciones de parámetros de tipo 304

10.1.6 Cuerpo de clase 309

10.2 Tipos parciales 309

10.2.1 Atributos 309

10.2.2 Modificadores 310

10.2.3 Parámetros de tipo y restricciones 310

10.2.4 Clase base 311

10.2.5 Interfaces base 311

10.2.6 Miembros 311

10.2.7 Métodos parciales 312

10.2.8 Enlace de nombres 314

10.3 Miembros de clase 315

10.3.1 El tipo de instancia 316

10.3.2 Miembros de tipos construidos 316

10.3.3 Herencia 317

10.3.4 Modificador New 318

10.3.5 Modificadores de acceso 319

10.3.6 Tipos constituyentes 319

10.3.7 Miembros estáticos y de instancia 319

10.3.8 Tipos anidados 320

10.3.8.1 Nombre completo 320

10.3.8.2 Accesibilidad declarada 320

10.3.8.3 Ocultar 321

10.3.8.4 Acceso this 322

10.3.8.5 Acceso a miembros privados y protegidos del tipo contenedor 322

10.3.8.6 Tipos anidados en clases genéricas 323

10.3.9 Nombres de miembro reservados 324

10.3.9.1 Nombres de miembros reservados para propiedades 324

10.3.9.2 Nombres de miembros reservados para eventos 325

10.3.9.3 Nombres de miembros reservados para indizadores 325

10.3.9.4 Nombres de miembros reservados para destructores 325

10.4 Constantes 325

10.5 Campos 327

10.5.1 Campos estáticos y de instancia 328

10.5.2 Campos de solo lectura 329

10.5.2.1 Utilizar campos de sólo lectura estáticos para constantes 330

10.5.2.2 Versiones de constantes y campos de sólo lectura estáticos 330

10.5.3 Campos volatile 331

10.5.4 Inicialización de campos 332

10.5.5 Inicializadores de variables 332

10.5.5.1 Inicialización de campos estáticos 333

10.5.5.2 Inicialización de campos de instancia 334

10.6 Métodos 335

10.6.1 Parámetros de métodos 337

10.6.1.1 Parámetros de valor 339

10.6.1.2 Parámetros de referencia 339

10.6.1.3 Parámetros de salida 340

10.6.1.4 Matrices de parámetros 341

10.6.2 Métodos estáticos y de instancia 343

10.6.3 Métodos virtuales 343

10.6.4 Métodos de reemplazo 345

10.6.5 Métodos sellados 348

10.6.6 Métodos abstractos 348

10.6.7 Métodos externos 350

10.6.8 Métodos parciales 350

10.6.9 Métodos de extensión 350

10.6.10 Cuerpo del método 351

10.6.11 Sobrecarga de métodos 352

10.7 Propiedades 352

10.7.1 Propiedades estáticas y de instancia 353

10.7.2 Descriptores de acceso 353

10.7.3 Propiedades automáticamente implementadas 359

10.7.4 Accesibilidad 359

10.7.5 Descriptores de acceso virtual, sellado, de invalidación y abstracto 361

10.8 Eventos 362

10.8.1 Eventos como campos 364

10.8.2 Descriptores de acceso de evento 366

10.8.3 Eventos estáticos y de instancia 367

10.8.4 Descriptores de acceso virtual, sellado, de invalidación y abstracto 367

10.9 Indizadores 368

10.9.1 Sobrecarga de indizadores 371

10.10 Operadores 371

10.10.1 Operadores unarios 373

10.10.2 Operadores binarios 374

10.10.3 Operadores de conversión 374

10.11 Constructores de instancia 377

10.11.1 Inicializadores de constructor 378

10.11.2 Inicializadores de variables de instancia 379

10.11.3 Ejecución de constructores 379

10.11.4 Constructores predeterminados 381

10.11.5 Constructores Private 381

10.11.6 Parámetros de constructor de instancia opcionales 382

10.12 Constructores static 382

10.13 Destructores 384

10.14 Iteradores 386

10.14.1 Interfaces de enumerador 386

10.14.2 Interfaces enumerables 386

10.14.3 Tipo Yield 386

10.14.4 Objetos del enumerador 386

10.14.4.1 El método MoveNext 387

10.14.4.2 La propiedad Current 388

10.14.4.3 El método Dispose 388

10.14.5 Objetos enumerables 389

10.14.5.1 El método GetEnumerator 389

10.14.6 Ejemplo de implementación 389

10.15 Funciones asincrónicas 395

10.15.1 Evaluación de una función asincrónica que devuelve tareas 396

10.15.2 Evaluación de una función asincrónica que devuelve void 396

11. Structs 397

11.1 Declaraciones de struct 397

11.1.1 Modificadores de struct 397

11.1.2 Modificador parcial 398

11.1.3 Interfaces struct 398

11.1.4 Cuerpo de struct 398

11.2 Miembros de struct 398

11.3 Diferencias entre clase y struct 398

11.3.1 Semánticas de valor 399

11.3.2 Herencia 400

11.3.3 Asignación 400

11.3.4 Valores predeterminados 400

11.3.5 Conversiones boxing y unboxing 401

11.3.6 Significado de this 403

11.3.7 Inicializadores de campo 403

11.3.8 Constructores 403

11.3.9 Destructores 404

11.3.10 Constructores static 404

11.4 Ejemplos de struct 404

11.4.1 Tipo entero de base de datos 404

11.4.2 Tipo booleano de base de datos 406

12. Matrices 409

12.1 Tipos de matriz 409

12.1.1 Tipo System.Array 410

12.1.2 Matrices y la interfaz IList genérica 410

12.2 Creación de matrices 411

12.3 Acceso a los elementos de matriz 411

12.4 Miembros de matriz 411

12.5 Covarianza de matrices 411

12.6 Inicializadores de matrices 412

13. Interfaces 415

13.1 Declaraciones de interfaz 415

13.1.1 Modificadores de interfaz 415

13.1.2 Modificador parcial 416

13.1.3 Listas de parámetros de tipo variante 416

13.1.3.1 Seguridad de la varianza 416

13.1.3.2 Conversión de varianza 417

13.1.4 Interfaces base 417

13.1.5 Cuerpo de interfaz 418

13.2 Miembros de interfaz 418

13.2.1 Métodos de interfaz 419

13.2.2 Propiedades de interfaz 420

13.2.3 Eventos de interfaz 420

13.2.4 Indizadores de interfaz 420

13.2.5 Acceso a miembros de interfaz 421

13.3 Nombres completos de miembros de interfaz 423

13.4 Implementaciones de interfaces 423

13.4.1 Implementaciones de miembro de interfaz explícitas 424

13.4.2 Exclusividad de interfaces implementadas 426

13.4.3 Implementación de métodos genéricos 427

13.4.4 Asignación de interfaces 428

13.4.5 Herencia de implementación de interfaces 431

13.4.6 Reimplementación de interfaces 432

13.4.7 Interfaces y clases abstractas 433

14. Enumeraciones 435

14.1 Declaraciones de enumeración 435

14.2 Modificadores de enumeración 435

14.3 Miembros de enumeración 436

14.4 Tipo System.Enum 438

14.5 Valores y operaciones de enumeración 438

15. Delegados 439

15.1 Declaraciones de delegados 439

15.2 Compatibilidad de delegados 441

15.3 Creación de instancias de delegados 442

15.4 Invocación de delegados 442

16. Excepciones 445

16.1 Causas de excepciones 445

16.2 Clase System.Exception 445

16.3 Cómo controlar excepciones 445

16.4 Clases de excepción comunes 446

17. Atributos 449

17.1 Clases de atributos 449

17.1.1 Uso de los atributos 449

17.1.2 Parámetros posicionales y con nombre 450

17.1.3 Tipos de parámetros de atributos 451

17.2 Especificación de atributos 451

17.3 Instancias de atributo 457

17.3.1 Compilación de un atributo 457

17.3.2 Recuperación en tiempo de ejecución de una instancia de atributo 457

17.4 Atributos reservados 458

17.4.1 Atributo AttributeUsage 458

17.4.2 Atributo Conditional 459

17.4.2.1 Métodos condicionales 459

17.4.2.2 Clases de atributos condicionales 461

17.4.3 Atributo Obsolete 462

17.4.4 Atributos de información del llamador 463

17.4.4.1 Atributo CallerLineNumber 464

17.4.4.2 Atributo CallerFilePath 464

17.4.4.3 Atributo CallerMemberName 464

17.5 Atributos para interoperabilidad 465

17.5.1 Interoperabilidad con componentes COM y Win32 465

17.5.2 Interoperabilidad con otros lenguajes .NET 465

17.5.2.1 Atributo IndexerName 465

18. Código no seguro 467

18.1 Contextos no seguros 467

18.2 Tipos de puntero 469

18.3 Variables fijas y móviles 472

18.4 Conversiones de puntero 473

18.4.1 Matrices de punteros 474

18.5 Punteros en expresiones 475

18.5.1 Direccionamiento indirecto de punteros 475

18.5.2 Acceso a miembros de puntero 476

18.5.3 Acceso a elementos de puntero 476

18.5.4 Operador address-of 477

18.5.5 Incremento y decremento de punteros 478

18.5.6 Aritmética con punteros 478

18.5.7 Comparación de punteros 479

18.5.8 Operador sizeof 479

18.6 Instrucción fixed 480

18.7 Búferes fijos 484

18.7.1 Declaraciones de búferes fijos 484

18.7.2 Búferes fijos en expresiones 485

18.7.3 Comprobación de asignación definitiva 486

18.8 Asignación de pila 486

18.9 Asignación dinámica de memoria 487

A. Comentarios de la documentación 491

A.1 Introducción 491

A.2 Etiquetas recomendadas 492

A.2.1 <c> \t "See <c>" \b 493

A.2.2 <code> \t "See <code>" \b 493

A.2.3 <example> \t "See <example>" \b 494

A.2.4 <exception> \t "See <exception>" \b 494

A.2.5 <include> 495

A.2.6 <list> \t "See <list>" \b 495

A.2.7 <para> \t "See <para>" \b 496

A.2.8 <param> \t "See <param>" \b 497

A.2.9 <paramref> \t "See <paramref>" \b 497

A.2.10 <permission> \t "See <permission>" \b 497

A.2.11 <remark> \t "See <remarks>" \b 498

A.2.12 <returns> \t "See <returns>" \b 498

A.2.13 <see> \t "See <see>" \b 499

A.2.14 <seealso> \t "See <seealso>" \b 499

A.2.15 <summary> \t "See <summary>" \b 499

A.2.16 <value> \t "See <value>" \b 500

A.2.17 <typeparam> 500

A.2.18 <typeparamref> 500

A.3 Procesamiento del archivo de documentación 501

A.3.1 Formato de cadena de Id. 501

A.3.2 Ejemplos de cadena de identificador 502

A.4 Un ejemplo 506

A.4.1 Código fuente C# 506

A.4.2 XML resultante 508

B. Gramática 512

B.1 Gramática léxica 512

B.1.1 Terminadores de línea 512

B.1.2 Comentarios 512

B.1.3 Espacio en blanco 513

B.1.4 Tokens 513

B.1.5 Secuencias de escape de caracteres Unicode 513

B.1.6 Identificadores 513

B.1.7 Palabras clave 514

B.1.8 Literales 515

B.1.9 Operadores y signos de puntuación 517

B.1.10 Directivas de preprocesamiento 517

B.2 Gramática sintáctica 519

B.2.1 Conceptos básicos 519

B.2.2 Tipos 519

B.2.3 Variables 521

B.2.4 Expresiones 521

B.2.5 Instrucciones 528

B.2.6 Espacios de nombres 531

B.2.7 Clases 532

B.2.8 Structs 539

B.2.9 Matrices 540

B.2.10 Interfaces 540

B.2.11 Enumeraciones 541

B.2.12 Delegados 542

B.2.13 Atributos 542

B.3 Extensiones de la gramática para el código no seguro 544

C. Referencias 547

# Introducción

C# (“C sharp”) es un lenguaje de programación sencillo y moderno, orientado a objetos y con seguridad de tipos. C# tiene su raíz en la familia de lenguajes C y resultará inmediatamente familiar a programadores de C, C++ y Java. ECMA International estandariza C# como el estándar ECMA-334, e ISO/CEI como el estándar ISO/IEC 23270. El compilador de C# de Microsoft para .NET Framework es una implementación compatible de ambos estándares.

C# es un lenguaje de programación orientado a objetos, pero también es compatible con la programación orientada a componentes. El diseño de software contemporáneo se basa cada vez más en componentes de software en forma de paquetes de funcionalidad autodescriptivos y autosuficientes. La clave de dichos componentes reside en que presentan un modelo de programación con propiedades, métodos y eventos, en que tienen atributos que facilitan información declarativa sobre el componente y en que incorporan su propia documentación. C# proporciona construcciones de lenguaje para que se admitan directamente estos conceptos, haciendo de C# un lenguaje muy natural en el que crear y utilizar los componentes de software.

Algunas de las características de C# ayudan en la construcción de aplicaciones sólidas y duraderas: la recolección de elementos no utilizados reclama automáticamente la memoria ocupada por objetos no utilizados; el control de excepciones proporciona un enfoque extensible y estructurado de la detección y recuperación de errores; y el diseño con seguridad de tipos del lenguaje hace que no se puedan leer variables no inicializadas, indizar las matrices más allá de sus límites o realizar conversiones de tipo no comprobado.

C# tiene un sistema de tipos unificado. Todos los tipos de C#, incluidos tipos primitivos como int y double, heredan de un tipo raíz object único. Por lo tanto, todos los tipos comparten un conjunto de operaciones comunes, y se pueden almacenar, transportar y tratar valores de cualquier tipo de una manera coherente. Además, C# admite tipos de referencia definidos por el usuario y tipos de valor, y permite la asignación dinámica de objetos así como el almacenamiento en línea de estructuras ligeras.

Para asegurarse de que los programas y bibliotecas de C# evolucionan con el tiempo de forma compatible, se ha dado mucha importancia al control de versiones en el diseño de C#. Muchos lenguajes de programación apenas prestan atención a este problema y, en consecuencia, los programas escritos en estos lenguajes se interrumpen más de lo necesario cuando se incluyen versiones más recientes de bibliotecas dependientes. Los aspectos del diseño de C# con influencia directa de las consideraciones sobre el control de versiones incluyen los modificadores virtual y override, las reglas de la resolución de sobrecarga de métodos y la compatibilidad con declaraciones explícitas de miembros de interfaz.

En el resto de este capítulo se explican las características esenciales del lenguaje C#. Mientras que en capítulos posteriores se describen las reglas y las excepciones de una forma muy detallada e incluso a veces matemática, este capítulo se ha redactado dando prioridad a la claridad y la brevedad, a veces incluso a expensas de la integridad. El propósito es proporcionar al lector una introducción al lenguaje que pueda facilitarle la programación de sus primeros programas y la lectura de posteriores capítulos.

## Hola a todos

El programa Hola a todos (“Hello, World”) se utiliza tradicionalmente para presentar un lenguaje de programación. Aquí está en C#:

using System;

class Hello  
{  
 static void Main() {  
 Console.WriteLine("Hello, World");  
 }  
}

Normalmente, la extensión de los archivos de código fuente de C# es .cs. Suponiendo que “Hello, World” se almacena en el archivo hello.cs, el programa se puede compilar con el compilador de Microsoft C#, utilizando la línea de comandos

csc hello.cs

que genera un ensamblado ejecutable denominado hello.exe. El resultado de la ejecución de esta aplicación es

Hello, World

El programa “Hello, World” se inicia con una directiva using que hace referencia al espacio de nombres System. Los espacios de nombres proporcionan una jerarquía que permite organizar programas y bibliotecas de C#. Los espacios de nombres contienen tipos y otros espacios de nombres, por ejemplo, el espacio de nombres System contiene varios tipos, como la clase Console, a la que se hace referencia en el programa, y otros espacios de nombres, como IO y Collections. Una directiva using que hace referencia a un espacio de nombres determinado permite el uso no calificado de los tipos que son miembros del espacio de nombres. Debido a la directiva using, el programa puede usar Console.WriteLine como forma abreviada de System.Console.WriteLine.

La clase Hello declarada por el programa “Hello, World” tiene un único miembro, el método denominado Main. El método Main se declara con el modificador static. Mientras que los métodos de instancia hacen referencia a una instancia de objeto determinada utilizando la palabra clave this, los métodos estáticos funcionan sin referencia a un objeto determinado. Por convención, un método estático denominado Main sirve como punto de entrada de un programa.

El método WriteLine genera el resultado del programa en la clase Console del espacio de nombres System. Esta clase es proporcionada por las bibliotecas de clases de .NET Framework, a las que de forma predeterminada hace referencia automáticamente el compilador de Microsoft C#. Observe que el propio C# no tiene una biblioteca en tiempo de ejecución independiente. En su lugar, .NET Framework es la biblioteca en tiempo de ejecución de C#.

## Estructura del programa

Los conceptos clave de organización en C# son programas, espacios de nombres, tipos, miembros y ensamblados. Los programas de C# constan de uno o varios archivos de código fuente. Los programas declaran tipos, que contienen los miembros y pueden organizarse en los espacios de nombres. Las clases y las interfaces son ejemplos de tipos. Los campos, métodos, propiedades y eventos son ejemplos de miembros. Cuando se compilan programas de C#, se empaquetan físicamente en ensamblados. Normalmente la extensión de archivo de los ensamblados es .exe o .dll, dependiendo de si implementan aplicaciones o bibliotecas.

El ejemplo

using System;

namespace Acme.Collections  
{  
 public class Stack  
 {  
 Entry top;

public void Push(object data) {  
 top = new Entry(top, data);  
 }

public object Pop() {  
 if (top == null) throw new InvalidOperationException();  
 object result = top.data;  
 top = top.next;  
 return result;  
 }

class Entry  
 {  
 public Entry next;  
 public object data;

public Entry(Entry next, object data) {  
 this.next = next;  
 this.data = data;  
 }  
 }  
 }  
}

declara una clase denominada Stack en un espacio de nombres denominado Acme.Collections. El nombre completo de la clase es Acme.Collections.Stack. La clase contiene varios miembros: un campo denominado top, dos métodos denominados Push y Pop, y una clase anidada denominada Entry. La clase Entry contiene tres miembros más: un campo denominado next, un campo denominado data y un constructor. Suponiendo que el código fuente del ejemplo se almacena en el archivo acme.cs, la línea de comandos

csc /t:library acme.cs

compila el ejemplo como una biblioteca (código sin un punto de entrada Main) y genera un ensamblado denominado acme.dll.

Los ensamblados contienen código ejecutable en forma de instrucciones de Lenguaje intermedio (IL) e información simbólica en forma de metadatos. Antes de ejecutarse, el código IL de un ensamblado se convierte automáticamente en código específico del procesador mediante el compilador Just-In-Time (JIT) de .NET Common Language Runtime.

Dado que un ensamblado es una unidad autodescriptiva de funcionalidad que contiene código y metadatos, las directivas #include y los archivos de encabezado no son necesarios en C#. Los tipos y miembros públicos contenidos en un ensamblado determinado estarán disponibles en un programa de C# haciendo referencia a dicho ensamblado al compilar el programa. Por ejemplo, este programa usa la clase Acme.Collections.Stack del ensamblado acme.dll:

using System;  
using Acme.Collections;

class Test  
{  
 static void Main() {  
 Stack s = new Stack();  
 s.Push(1);  
 s.Push(10);  
 s.Push(100);  
 Console.WriteLine(s.Pop());  
 Console.WriteLine(s.Pop());  
 Console.WriteLine(s.Pop());  
 }  
}

Si el programa se almacena en el archivo test.cs, cuando se compila test.cs se puede hacer referencia al ensamblado acme.dll utilizando la opción del compilador /r:

csc /r:acme.dll test.cs

De este modo se crea un ensamblado ejecutable denominado test.exe que, cuando se ejecuta, genera el resultado:

100  
10  
1

C# permite el almacenamiento del texto de origen de un programa en varios archivos de código fuente. Cuando se compila un programa de C# de varios archivos, se procesan todos los archivos de código fuente juntos, y éstos pueden hacer referencia libremente unos a otros. Conceptualmente, es como si todos los archivos de código fuente se concatenaran en un gran archivo antes de procesarse. En C# nunca son necesarias las declaraciones adelantadas porque, salvo alguna excepción, el orden de la declaración no es significativo. C# no limita un archivo de código fuente a declarar un único tipo público ni exige que el nombre del archivo de código fuente coincida con un tipo declarado en el archivo de código fuente.

## Tipos y variables

Hay dos clases de tipos en C#: tipos de valor y tipos de referencia. Las variables de tipos de valor contienen directamente sus datos mientras que las variables de tipos de referencia almacenan las referencias a sus datos, que se conocen como objetos. En el caso de los tipos de referencia, es posible que dos variables hagan referencia al mismo objeto y, por tanto, que las operaciones en una variable afecten al objeto al que hace referencia la otra variable. En el caso de los tipos de valor, cada variable tiene su propia copia de los datos, de manera que no es posible que las operaciones de una afecten a la otra (exceptuando las variables de los parámetros ref y out).

Los tipos de valor de C# se dividen a su vez en tipos simples, tipos de enumeración, tipos struct y tipos que aceptan valores NULL, y los tipos de referencia de C# en tipos de clase, tipos de interfaz, tipos de matriz y tipos de delegado.

La tabla siguiente proporciona información general sobre el sistema de tipos de C#.

|  |  |  |
| --- | --- | --- |
| **Categoría** | | **Descripción** |
| Tipos de valor | Tipos simples | Integral con signo: sbyte, short, int, long |
| Integral sin signo: byte, ushort, uint, ulong |
| Caracteres Unicode: char |
| Punto flotante IEEE: float, double |
| Decimal de gran precisión: decimal |
| Booleano: bool |
| Tipos de enumeración | Tipos definidos por el usuario en forma de enum E {...} |
| Tipos struct | Tipos definidos por el usuario en forma de struct S {...} |
| Tipos que aceptan valores null | Extensiones de todos los demás tipos de valores con un valor null |
| Tipos de referencia | Tipos de clase | Clase base definitiva de todos los demás tipos: object |
| Cadenas Unicode: string |
| Tipos definidos por el usuario en forma de class C {...} |
| Tipos de interfaz | Tipos definidos por el usuario en forma de interface I {...} |
| Tipos de matriz | Unidimensional y multidimensional, por ejemplo, int[] e int[,] |
| Tipos de delegados | Tipos definidos por el usuario en forma de, por ejemplo, delegate int D(...) |

Los ocho tipos enteros admiten los valores de 8 bits, 16 bits, 32 bits y 64 bits, con y sin signo.

Los dos tipos de punto flotante float y double se representan utilizando los formatos IEEE 754 de 32 bits de precisión sencilla y de 64 bits de doble precisión.

El tipo decimal es un tipo de datos de 128 bits apto para cálculos financieros y monetarios.

El tipo bool de C# permite representar valores booleanos, que son true o false.

El procesamiento de caracteres y cadenas en C# utiliza la codificación Unicode. El tipo char representa una unidad de código UTF de 16 bits y el tipo string representa una secuencia de unidades de código UTF de 16 bits.

La tabla siguiente resume los tipos numéricos de C#.

|  |  |  |  |
| --- | --- | --- | --- |
| **Categoría** | **Bits** | **Tipo** | **Intervalo/precisión** |
| Integral con signo | 8 | sbyte | –128...127 |
| 16 | short | –32,768...32,767 |
| 32 | int | –2,147,483,648...2,147,483,647 |
| 64 | long | –9,223,372,036,854,775,808...9,223,372,036,854,775,807 |
| Integral sin signo | 8 | byte | 0...255 |
| 16 | ushort | 0...65,535 |
| 32 | uint | 0...4,294,967,295 |
| 64 | ulong | 0...18,446,744,073,709,551,615 |
| Punto flotante | 32 | float | 1,5 × 10−45 a 3,4 × 1038, con precisión de 7 dígitos |
| 64 | double | 5,0 × 10−324 a 1,7 × 10308, con precisión de 15 dígitos |
| Decimal | 128 | decimal | 1,0 × 10−28 a 7,9 × 1028, con precisión de 28 dígitos |

Los programas de C# utilizan declaraciones de tipo para crear nuevos tipos. Una declaración de tipo especifica el nombre y los miembros del nuevo tipo. Cinco de las categorías de tipos de C# son definibles por el usuario: tipos de clase, tipos struct, tipos de interfaz, tipos de enumeración y tipos de delegado.

Un tipo de clase define una estructura de datos que contiene miembros de datos (campos) y miembros de función (métodos, propiedades y otros). Los tipos de clase admiten la herencia simple y el polimorfismo, mecanismos mediante los que las clases derivadas pueden ampliar y especializar clases base.

Un tipo struct es similar a un tipo de clase en cuanto a que ambas representan una estructura con miembros de datos y miembros de función. No obstante, a diferencia de las clases, los structs son tipos de valores y no requieren asignación del montón. Los tipos struct no admiten la herencia especificada por el usuario y heredan implícitamente del tipo object.

Un tipo de interfaz define un contrato como un conjunto con nombre de miembros de función pública. Una clase o struct que implementa una interfaz debe proporcionar implementaciones de los miembros de función de la interfaz. Una interfaz puede heredar de varias interfaces base, y una clase o struct puede implementar varias interfaces.

Un tipo delegado representa las referencias a los métodos con una lista de parámetros determinada y un tipo de valor devuelto. Los delegados permiten tratar métodos como entidades, que se pueden asignar a las variables y pasarse como parámetros. Los delegados son similares al concepto de punteros a función de otros lenguajes, pero al contrario de los punteros a función, los delegados están orientados a objetos y proporcionan seguridad de tipos.

Los tipos de delegado, interfaz, struct y clase admiten genéricos, lo que permite su parametrización con otros tipos.

Un tipo de enumeración es un tipo distinto con constantes con nombre. Todos los tipos de enumeración tienen un tipo subyacente, que debe ser uno de los ocho tipos enteros. El conjunto de valores de un tipo de enumeración es el mismo que el del conjunto de valores del tipo subyacente.

C# admite matrices unidimensionales y multidimensionales de cualquier tipo. Al contrario que los tipos mencionados antes, los tipos de matriz no tienen que ser declarados antes de poder utilizarse. En su lugar, los tipos de matriz se crean mediante un nombre de tipo seguido de corchetes. Por ejemplo, int[] es una matriz unidimensional de int, int[,] es una matriz bidimensional de int e int[][] es una matriz unidimensional de matrices unidimensionales de int.

Los tipos que aceptan valores NULL tampoco tienen que ser declarados antes de poder utilizarse. Para cada tipo de valor T que no acepta valores null hay un tipo de valor correspondiente T? que sí acepta valores null, que puede contener otro valor null. Por ejemplo, int? es un tipo que puede contener cualquier entero de 32 bits o el valor null.

El sistema de tipos de C# está unificado, de manera que un valor de cualquier tipo puede tratarse como un objeto. Todos los tipos de C# se derivan directa o indirectamente del tipo de clase object, que es la clase base definitiva de todos los tipos. Los valores de los tipos de referencia se tratan como objetos considerándolos sencillamente como del tipo object. Los valores de los tipos de valor se tratan como objetos mediante la realización de operaciones de conversión boxing y unboxing. En el ejemplo siguiente, un valor int se convierte en object y de nuevo en int.

using System;

class Test  
{  
 static void Main() {  
 int i = 123;  
 object o = i; // Boxing  
 int j = (int)o; // Unboxing  
 }  
}

Cuando un valor de un tipo de valor se convierte en el tipo object, se asigna una instancia de objeto, también denominada “box”, para que contenga el valor, y este se copia en dicha instancia. A la inversa, cuando se convierte una referencia object en un tipo de valor, se comprueba que el objeto al que se hace referencia es un cuadro del tipo con el valor correcto, y si la comprobación se lleva a cabo con éxito, el valor del cuadro se copia externamente.

El sistema de tipos unificado de C# significa en la práctica que los tipos de valor pueden convertirse en objetos“a petición”. Debido a la unificación, las bibliotecas de uso general que utilizan el tipo object pueden utilizarse tanto con los tipos de referencia como con los tipos de valor.

Hay varios tipos de variables en C#, entre los que se incluyen campos, elementos de matriz, variables locales y parámetros. Las variables representan ubicaciones de almacenamiento, y cada variable tiene un tipo que determina qué valores se pueden almacenar en ella, como se muestra en la tabla siguiente.

|  |  |
| --- | --- |
| **Tipo de variable** | **Posible contenido** |
| Tipo de valor que no acepta valores NULL | Un valor de ese tipo exacto |
| Tipo de valor que acepta valores NULL | Un valor NULL o un valor de ese tipo exacto |
| object | Una referencia nula, una referencia a un objeto de cualquier tipo de referencia o una referencia a un valor de tipo box de cualquier tipo de valor |
| Tipo de clase | Una referencia nula, una referencia a una instancia de ese tipo de clase o una referencia a una instancia de una clase derivada de dicho tipo de clase |
| Tipo de interfaz | Una referencia nula, una referencia a una instancia de un tipo de clase que implemente dicho tipo de interfaz o una referencia a un valor de tipo box que implemente dicho tipo de interfaz |
| Tipo de matriz | Una referencia nula, una referencia a una instancia de dicho tipo de matriz o una referencia a una instancia de un tipo de matriz compatible |
| Tipo delegado | Una referencia nula o una referencia a una instancia de dicho tipo delegado |

## Expresiones

Las expresiones se construyen a partir de operandos y operadores. Los operadores de una expresión indican qué operaciones se aplican a los operandos. Entre los ejemplos de operadores se incluyen +, -, \*, / y new. Son ejemplos de operandos los literales, campos, variables locales y expresiones.

Cuando una expresión contiene varios operadores, la prioridad de los operadores controla el orden de evaluación de los operadores individuales. Por ejemplo, la expresión x + y \* z se evalúa como x + (y \* z) porque el operador \* tiene prioridad sobre +.

La mayoría de los operadores se pueden sobrecargar. La sobrecarga de operadores permite utilizar implementaciones de operadores definidas por el usuario en operaciones en las que al menos uno de los operandos es de un tipo struct o clase definido por el usuario.

En la tabla siguiente se resumen los operadores de C#, enumerando las categorías en orden de prioridad de mayor a menor: Los operadores de la misma categoría tienen la misma prioridad.

|  |  |  |
| --- | --- | --- |
| **Categoría** | **Expresión** | **Descripción** |
| Principal | x.m | Acceso a miembros |
| x(...) | Invocación de métodos y delegados |
| x[...] | Matriz y acceso del indizador |
| x++ | Incremento posterior |
| x-- | Decremento posterior |
| new T(...) | Creación de objetos y de delegados |
| new T(...){...} | Creación de objeto con el inicializador |
| new {...} | Inicializador de objeto anónimo |
| new T[...] | Creación de matrices |
| typeof(T) | Obtenga un objeto System.Type para T |
| checked(x) | Evalúa la expresión en contexto comprobado |
| unchecked(x) | Evalúa la expresión en contexto no comprobado |
| default(T) | Obtiene el valor predeterminado del tipo T |
| delegate {...} | Función anónima (método anónimo) |
| Unario | +x | Identidad |
| -x | Negación |
| !x | Negación lógica |
| ~x | Negación bit a bit |
| ++x | Incremento previo |
| --x | Decremento previo |
| (T)x | Convierte explícitamente x en el tipo T |
| await x | Espere asincrónicamente a que x se complete |
| Multiplicativo | x \* y | Multiplicación |
| x / y | División |
| x % y | Resto |

|  |  |  |
| --- | --- | --- |
| Sumatorio | x + y | Suma, concatenación de cadenas, combinación de delegados |
| x – y | Resta, eliminación de delegados |
| Desplazamiento | x << y | Desplazamiento a la izquierda |
| x >> y | Desplazamiento a la derecha |
| Comprobación de tipos y relacionales | x < y | Menor que |
| x > y | Mayor que |
| x <= y | Menor o igual que |
| x >= y | Mayor o igual que |
| x is T | Devuelva true si x es T; de lo contrario, false |
| x as T | Devuelva x con tipo como T, o null si x no es T |
| Igualdad | x == y | Igual |
| x != y | No igual |
| AND lógico | x & y | AND bit a bit entero, AND lógico booleano |
| XOR lógico | x ^ y | XOR bit a bit entero, XOR lógico booleano |
| OR lógico | x | y | OR bit a bit entero, OR lógico booleano |
| AND condicional | x && y | Evalúa y solo si x es true |
| OR condicional | x || y | Evalúa y solo si x es false |
| Uso combinado de Null | X ?? y | Se evalúa como y si x es null; de lo contrario, se evalúa como x |
| Condicional | x ? y : z | Evalúa y si x es true, z si x es false |
| Asignación o función anónima | x = y | Asignación |
| x op= y | Asignación compuesta; los operadores compatibles son  \*= /= %= += -= <<= >>= &= ^= |= |
| (T x) => y | Función anónima (expresión lambda) |

## Instrucciones

Las acciones de un programa se expresan utilizando instrucciones. C# admite varios tipos de instrucciones, algunos de los cuales se definen en términos de instrucciones incrustadas.

Un bloque (block) permite escribir varias instrucciones en contextos donde se permite una única instrucción. Un bloque se compone de una lista de instrucciones escrita entre delimitadores { y }.

Las instrucciones de declaración se utilizan para declarar constantes y variables locales.

Las instrucciones de expresión se utilizan para evaluar las expresiones. Las expresiones que pueden utilizarse como instrucciones incluyen invocaciones de método, asignaciones de objetos que usan el operador new, asignaciones que usan = y operadores de asignación compuesta, operadores de incremento y decremento que utilizan los operadores ++ y -- y expresiones de espera.

Las instrucciones de selección seleccionan una de las instrucciones que se van a ejecutar en función del valor de alguna expresión. En este grupo están las instrucciones if y switch.

Las instrucciones de iteración se utilizan para ejecutar repetidamente una instrucción incrustada. En este grupo están las instrucciones while, do, for y foreach.

Las instrucciones de salto se utilizan para transferir el control. En este grupo están las instrucciones break, continue, goto, throw, return y yield .

La instrucción try...catch se utiliza para detectar las excepciones que se producen durante la ejecución de un bloque y la instrucción try...finally se usa para especificar el código de finalización que siempre se ejecuta, se produzca o no una excepción.

Las instrucciones checked y unchecked se utilizan con el fin de controlar el contexto de comprobación de desbordamiento para operaciones aritméticas y conversiones.

La instrucción lock se utiliza para bloquear un objeto determinado mediante exclusión mutua, ejecutar una instrucción y, a continuación, liberar el bloqueo.

La instrucción using se utiliza para obtener un recurso, ejecutar una instrucción y, a continuación, eliminar dicho recurso.

La tabla siguiente muestra las instrucciones de C# y proporciona un ejemplo para cada una.

|  |  |
| --- | --- |
| **Instrucción** | **Ejemplo** |
| Declaraciones de variables locales | static void Main() {  int a;   int b = 2, c = 3;   a = 1;  Console.WriteLine(a + b + c); } |
| Declaraciones de constantes locales | static void Main() {  const float pi = 3.1415927f;  const int r = 25;  Console.WriteLine(pi \* r \* r); } |
| Instrucciones de expresiones | static void Main() {  int i;  i = 123; // Expression statement  Console.WriteLine(i); // Expression statement  i++; // Expression statement  Console.WriteLine(i); // Expression statement } |
| Instrucción if | static void Main(string[] args) {  if (args.Length == 0) {  Console.WriteLine("No arguments");  }  else {  Console.WriteLine("One or more arguments");  } } |

|  |  |
| --- | --- |
| Instrucción switch | static void Main(string[] args) {  int n = args.Length;  switch (n) {  case 0:  Console.WriteLine("No arguments");  break;  case 1:  Console.WriteLine("One argument");  break;  default:  Console.WriteLine("{0} arguments", n);  break;  }  } } |
| Instrucción while | static void Main(string[] args) {  int i = 0;  while (i < args.Length) {  Console.WriteLine(args[i]);  i++;  } } |
| Instrucción do | static void Main() {  string s;  do {  s = Console.ReadLine();  if (s != null) Console.WriteLine(s);  } while (s != null); } |
| Instrucción for | static void Main(string[] args) {  for (int i = 0; i < args.Length; i++) {  Console.WriteLine(args[i]);  } } |
| Instrucción foreach | static void Main(string[] args) {  foreach (string s in args) {  Console.WriteLine(s);  } } |
| Instrucción break | static void Main() {  while (true) {  string s = Console.ReadLine();  if (s == null) break;  Console.WriteLine(s);  } } |
| Instrucción continue | static void Main(string[] args) {  for (int i = 0; i < args.Length; i++) {  if (args[i].StartsWith("/")) continue;  Console.WriteLine(args[i]);  } } |

|  |  |
| --- | --- |
| Instrucción goto | static void Main(string[] args) {  int i = 0;  goto check;  loop:  Console.WriteLine(args[i++]);  check:  if (i < args.Length) goto loop; } |
| Instrucción return | static int Add(int a, int b) {  return a + b; }  static void Main() {  Console.WriteLine(Add(1, 2));  return; } |
| Instrucción yield | static IEnumerable<int> Range(int from, int to) {  for (int i = from; i < to; i++) {  yield return i;  }  yield break; }  static void Main() {  foreach (int x in Range(-10,10)) {  Console.WriteLine(x);  } } |
| Instrucciones throw y try | static double Divide(double x, double y) {  if (y == 0) throw new DivideByZeroException();  return x / y; }  static void Main(string[] args) {  try {  if (args.Length != 2) {  throw new Exception("Two numbers required");  }  double x = double.Parse(args[0]);  double y = double.Parse(args[1]);  Console.WriteLine(Divide(x, y));  }  catch (Exception e) {  Console.WriteLine(e.Message);  }  finally {  Console.WriteLine(“Good bye!”);  } } |
| Instrucciones checked y unchecked | static void Main() {  int i = int.MaxValue;  checked {  Console.WriteLine(i + 1); // Exception  }  unchecked {  Console.WriteLine(i + 1); // Overflow  } } |

|  |  |
| --- | --- |
| Instrucción lock | class Account {  decimal balance;  public void Withdraw(decimal amount) {  lock (this) {  if (amount > balance) {  throw new Exception("Insufficient funds");  }  balance -= amount;  }  } } |
| Instrucción using | static void Main() {  using (TextWriter w = File.CreateText("test.txt")) {  w.WriteLine("Line one");  w.WriteLine("Line two");  w.WriteLine("Line three");  } } |

## Clases y objetos

Las clases son el tipo más importante de C#. Una clase es una estructura de datos que combina estados (campos) y acciones (métodos y otros miembros de función) en una unidad única. Una clase proporciona una definición para las instancias de la clase creadas dinámicamente, también conocidas como objetos. Las clases admiten la herencia y el polimorfismo, mecanismos mediante los que una clase derivada puede ampliar y especializar una clase base.

Las clases nuevas se crean utilizando declaraciones de clase. Las declaraciones de clase se inician con un encabezado que especifica los atributos y modificadores de la clase, el nombre, la clase base (si se conoce) y las interfaces implementadas por ella. El encabezado va seguido por el cuerpo de la clase, que consiste en una lista de declaraciones de miembros escritas entre los delimitadores { y }.

La siguiente es una declaración de una clase simple denominada Point:

public class Point  
{  
 public int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

Las instancias de clases se crean utilizando el operador new, que asigna memoria para una nueva instancia, invoca un constructor para inicializar la instancia y devuelve una referencia a la misma. Las instrucciones siguientes crean dos objetos Point y almacenan las referencias a dichos objetos en dos variables:

Point p1 = new Point(0, 0);  
Point p2 = new Point(10, 20);

La memoria ocupada por un objeto se reclama automáticamente cuando el objeto deja de estar en uso. No es posible ni necesario desasignar explícitamente objetos en C#.

### Miembros

Los miembros de una clase son miembros estáticos o miembros de instancia. Los miembros estáticos pertenecen a las clases y los miembros de instancia pertenecen a los objetos (instancias de clases).

La tabla siguiente proporciona información general sobre los tipos de miembros que puede contener una clase.

|  |  |
| --- | --- |
| **Miembro** | **Descripción** |
| Constantes | Valores constantes asociados a la clase |
| Campos | Variables de la clase |
| Métodos | Cálculos y acciones que puede realizar la clase |
| Propiedades | Acciones asociadas a la lectura y escritura de propiedades con nombre de la clase |
| Indizadores | Acciones asociadas a instancias de indización de la clase como una matriz |
| Eventos | Notificaciones que pueden ser generadas por la clase |
| Operadores | Conversiones y operadores de expresión admitidos por la clase |
| Constructores | Acciones requeridas para inicializar instancias de la clase o la propia clase |
| Destructores | Acciones que hay que llevar a cabo antes de que las instancias de la clase sean descartadas de forma permanente |
| Tipos | Tipos anidados declarados por la clase |

### Accesibilidad

Cada miembro de una clase tiene asociada una accesibilidad, que controla las regiones del texto del programa a las que tiene acceso el miembro. Existen cinco formas posibles de accesibilidad. Éstas se resumen en la siguiente tabla.

|  |  |
| --- | --- |
| **Accesibilidad** | **Significado** |
| public | Acceso no limitado |
| protected | Acceso limitado a esta clase o a las clases derivadas de la misma |
| internal | Acceso limitado a este programa |
| protected internal | Acceso limitado a este programa o a las clases derivadas de la misma |
| private | Acceso limitado a esta clase |

### Parámetros de tipo

Una definición de clase puede especificar conjunto de parámetros de tipo si a continuación del nombre de clase se colocan corchetes angulares que contengan una lista con los nombres de los parámetros de tipo. Los parámetros de tipo se pueden utilizar dentro de las declaraciones de clase para definir a los miembros de la clase. En el ejemplo siguiente los parámetros de tipo de Pair son TFirst y TSecond:

public class Pair<TFirst,TSecond>  
{  
 public TFirst First;

public TSecond Second;  
}

Un tipo de clase declarado para incluir parámetros de tipo se denomina tipo de clase genérico. Los tipos struct, interface y delegate también pueden ser genéricos.

Cuando se usa la clase genérica, se deben proporcionar argumentos de tipo para cada parámetro de tipo.

Pair<int,string> pair = new Pair<int,string> { First = 1, Second = “two” };  
int i = pair.First; // TFirst is int  
string s = pair.Second; // TSecond is string

Un tipo genérico con los argumentos de tipo proporcionados, como Pair<int,string> en el ejemplo anterior, se denomina tipo construido.

### Clases base

Una declaración de clase puede especificar una clase base si a continuación del nombre de clase y los parámetros de tipo se colocan dos puntos y el nombre de la clase base. Omitir una especificación de clase base es igual que derivar del tipo object. En el ejemplo siguiente, la clase base de Point3D es Point y la clase base de Point es object:

public class Point  
{  
 public int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

public class Point3D: Point  
{  
 public int z;

public Point3D(int x, int y, int z): base(x, y) {  
 this.z = z;  
 }  
}

Las clases heredan los miembros de sus clases base. La herencia significa que una clase contiene implícitamente todos los miembros de su clase base directa, excepto los constructores de instancia, los constructores estáticos y los destructores de la clase base. Una clase derivada puede agregar nuevos miembros a los heredados, pero no puede quitar la definición de un miembro heredado. En el ejemplo anterior, Point3D hereda los campos x e y de Point, y cada instancia de Point3D contiene tres campos , x, y y z.

Existe una conversión implícita desde un tipo de clase a cualquiera de sus tipos de clase base. Por consiguiente, una variable de un tipo de clase puede hacer referencia a una instancia de dicha clase o a una instancia de cualquier clase derivada. Por ejemplo, teniendo en cuenta las declaraciones de clase anteriores, una variable de tipo Point puede hacer referencia a un Point o a un Point3D:

Point a = new Point(10, 20);  
Point b = new Point3D(10, 20, 30);

### Campos

Un campo es una variable que está asociada a una clase o a una instancia de una clase.

Un campo declarado con el modificador static define un campo estático. Un campo estático identifica exactamente una ubicación de almacenamiento. Independientemente del número de instancias que se creen de una clase, sólo habrá una copia de un campo estático.

Un campo declarado sin el modificador static define un campo de instancia. Todas las instancias de una clase contienen una copia independiente de todos los campos de instancia de esa clase.

En el ejemplo siguiente, cada instancia de la clase Color tiene una copia independiente de los campos de instancia r, g y b, pero solo hay una copia de los campos estáticos Black, White, Red, Green y Blue:

public class Color  
{  
 public static readonly Color Black = new Color(0, 0, 0);  
 public static readonly Color White = new Color(255, 255, 255);  
 public static readonly Color Red = new Color(255, 0, 0);  
 public static readonly Color Green = new Color(0, 255, 0);  
 public static readonly Color Blue = new Color(0, 0, 255);

private byte r, g, b;

public Color(byte r, byte g, byte b) {  
 this.r = r;  
 this.g = g;  
 this.b = b;  
 }  
}

Como se puede ver en el ejemplo anterior, los campos de solo lectura se pueden declarar con un modificador readonly. Solamente se pueden realizar asignaciones a un campo readonly como parte de la declaración del campo o en un constructor en la misma clase.

### Métodos

Un método es un miembro que implementa un cálculo o una acción que puede realizar un objeto o una clase. El acceso a los métodos estáticos se obtiene a través de la clase. Se tiene acceso a los métodos de instancia a través de las instancias de la clase.

Los métodos tienen una lista (posiblemente vacía) de parámetros, que representan valores o referencias variables que se pasan al método y un tipo de valor devuelto que especifica el tipo del valor calculado y devuelto por el método. El tipo de valor devuelto de un método es void si no devuelve ningún valor.

Al igual que los tipos, los métodos también pueden tener un conjunto de parámetros de tipo, para los que se deben especificar argumentos de tipo cuando se llama al método. A diferencia de los tipos, los argumentos de tipo por lo general se pueden inferir a partir de los argumentos de una llamada de método y no necesitan darse de manera explícita.

La signatura de un método debe ser única en la clase en la que se declara el método. La firma de un método se compone del nombre del método, el número de parámetros de tipo, así como del número, modificadores y tipos de sus parámetros. La firma de un método no incluye el tipo de valor devuelto.

#### Parámetros

Los parámetros se utilizan para pasar valores o referencias variables a los métodos. Los parámetros de un método reciben sus valores reales de los argumentos que se especifican cuando se invoca el método. Existen cuatro clases de parámetros: parámetros de valores, parámetros de referencias, parámetros de salida y matrices de parámetros.

Para pasar el parámetro de entrada se utiliza un parámetro de valor. Un parámetro de valor corresponde a una variable local que obtiene su valor inicial del argumento proporcionado para el parámetro. Las modificaciones de un parámetro de valor no afectan al argumento proporcionado para el parámetro.

Los parámetros de valor pueden ser opcionales si se especifica un valor predeterminado de forma que se puedan omitir los argumentos correspondientes.

Un parámetro de referencia se utiliza tanto para proporcionar parámetros de entrada como de salida. El argumento pasado para un parámetro de referencia debe ser una variable, y durante la ejecución del método, el parámetro de referencia representa la misma ubicación de almacenamiento que la variable del argumento. Los parámetros de referencia se declaran con el modificador ref. En el ejemplo siguiente se ilustra el uso de los parámetros ref.

using System;

class Test  
{  
 static void Swap(ref int x, ref int y) {  
 int temp = x;  
 x = y;  
 y = temp;  
 }

static void Main() {  
 int i = 1, j = 2;  
 Swap(ref i, ref j);  
 Console.WriteLine("{0} {1}", i, j); // Outputs "2 1"  
 }  
}

Para pasar parámetros de salida se utiliza un parámetro de salida. Un parámetro de salida es similar a un parámetro de referencia, excepto en que el valor inicial del argumento suministrado por el llamador no es importante. Los parámetros de salida se declaran con un modificador out. En el ejemplo siguiente se ilustra el uso de los parámetros out.

using System;

class Test  
{  
 static void Divide(int x, int y, out int result, out int remainder) {  
 result = x / y;  
 remainder = x % y;  
 }

static void Main() {  
 int res, rem;  
 Divide(10, 3, out res, out rem);  
 Console.WriteLine("{0} {1}", res, rem); // Outputs "3 1"  
 }  
}

Una matriz de parámetros permite pasar un número variable de argumentos a un método. Una matriz de parámetros se declara con un modificador params. Sólo el último parámetro de un método puede ser una matriz de parámetros y el tipo de una matriz de parámetros debe ser un tipo de matriz unidimensional. Los métodos Write y WriteLine de la clase System.Console son buenos ejemplos del uso de una matriz de parámetros. Se declaran del siguiente modo:

public class Console  
{  
 public static void Write(string fmt, params object[] args) {...}

public static void WriteLine(string fmt, params object[] args) {...}

...  
}

Dentro de un método que utiliza una matriz de parámetros, la matriz de parámetros se comporta exactamente como un parámetro normal de un tipo de matriz. Sin embargo, en una invocación de un método con una matriz de parámetros, es posible pasar un argumento único del tipo de matriz de parámetros o cualquier número de argumentos del tipo elemento de la matriz de parámetros. En este último caso, se crea automáticamente una instancia de matriz y se inicializa con los argumentos especificados. Este ejemplo

Console.WriteLine("x={0} y={1} z={2}", x, y, z);

equivale a escribir lo siguiente.

string s = "x={0} y={1} z={2}";  
object[] args = new object[3];  
args[0] = x;  
args[1] = y;  
args[2] = z;  
Console.WriteLine(s, args);

#### Cuerpo del método y variables locales

El cuerpo de un método especifica las instrucciones que se deben ejecutar cuando se invoque el método.

Un cuerpo del método puede declarar variables específicas para la invocación del método. Tales variables se denominan variables locales. Una declaración de variable local especifica un nombre de tipo, un nombre de variable y posiblemente un valor inicial. El ejemplo siguiente declara una variable local i con un valor inicial de cero y una variable local j sin valor inicial.

using System;

class Squares  
{  
 static void Main() {  
 int i = 0;  
 int j;  
 while (i < 10) {  
 j = i \* i;  
 Console.WriteLine("{0} x {0} = {1}", i, j);  
 i = i + 1;  
 }  
 }  
}

C# exige que las variables locales estén definitivamente asignadas para que se puedan obtener sus valores. Por ejemplo, si la declaración de i anterior no incluyera un valor inicial, el compilador notificaría un error para la utilización subsiguiente de i, porque i no estaría definitivamente asignada en dichos puntos del programa.

Un método puede utilizar instrucciones return para devolver el control a su llamador. En un método que devuelve void, las instrucciones return no pueden especificar una expresión. En un método que no devuelve void, las instrucciones return deben incluir una expresión que calcule el valor devuelto.

#### Métodos estáticos y de instancia

Un método declarado con un modificador static es un método estático. Un método estático no funciona en una instancia específica y sólo tiene acceso directamente a miembros estáticos.

Un método declarado sin un modificador static es un método de instancia. Los métodos de instancia funcionan en instancias específicas y pueden tener acceso tanto a miembros estáticos como a los de instancia. Se puede tener acceso explícitamente a la instancia en la que se invocó un método de instancia como this. Es incorrecto hacer referencia a this en un método estático.

La clase Entity siguiente tiene tanto miembros estáticos como de instancia.

class Entity  
{  
 static int nextSerialNo;

int serialNo;

public Entity() {  
 serialNo = nextSerialNo++;  
 }

public int GetSerialNo() {  
 return serialNo;  
 }

public static int GetNextSerialNo() {  
 return nextSerialNo;  
 }

public static void SetNextSerialNo(int value) {  
 nextSerialNo = value;  
 }  
}

Cada instancia Entity contiene un número de serie (y probablemente alguna otra información que no se muestra aquí). El constructor Entity (que es como un método de instancia) inicializa la nueva instancia con el siguiente número de serie disponible. Dado que el constructor es un miembro de instancia, se puede tener acceso tanto al campo de instancia serialNo como al campo estático nextSerialNo.

Los métodos estáticos GetNextSerialNo y SetNextSerialNo pueden tener acceso al campo estático nextSerialNo, pero sería incorrecto que tuvieran acceso directamente al campo de instancia serialNo.

En el ejemplo siguiente se muestra el uso de la clase Entity.

using System;

class Test  
{  
 static void Main() {  
 Entity.SetNextSerialNo(1000);

Entity e1 = new Entity();  
 Entity e2 = new Entity();

Console.WriteLine(e1.GetSerialNo()); // Outputs "1000"  
 Console.WriteLine(e2.GetSerialNo()); // Outputs "1001"  
 Console.WriteLine(Entity.GetNextSerialNo()); // Outputs "1002"  
 }  
}

Tenga en cuenta que los métodos estáticos SetNextSerialNo y GetNextSerialNo se invocan en la clase, mientras que el método de instancia GetSerialNo se invoca en las instancias de la clase.

#### Métodos virtuales, de invalidación y abstractos

Cuando una declaración de método de instancia incluye un modificador virtual, se dice que el método es un método virtual. Si no existe un modificador virtual, se dice que el método es un método no virtual.

En la invocación de un método virtual, el tipo en tiempo de ejecución de la instancia para la que tiene lugar la invocación determina la implementación del método real que se va a invocar. Cuando se invoca un método no virtual, el factor determinante es el tipo en tiempo de compilación de la instancia.

En las clases derivadas, los métodos virtuales se pueden invalidar. Cuando una declaración del método de instancia incluye un modificador override, el método invalida un método virtual heredado con la misma signatura. Mientras que una declaración de método virtual introduce un método nuevo, una declaración de método de invalidación especializa un método virtual heredado existente proporcionando una nueva implementación de ese método.

Un método abstract es un método virtual sin implementación. Los métodos abstractos se declaran con el modificador abstract y sólo se permiten en una clase que también se declare como abstract. Los métodos abstractos deben invalidarse en cada clase derivada no abstracta.

El ejemplo siguiente declara una clase abstracta, Expression, que representa un nodo de árbol de expresión y tres clases derivadas Constant, VariableReference y Operation, que implementan los nodos de árbol de expresión para las constantes, referencias variables y operaciones aritméticas. (Esto es similar a los tipos de árbol de expresiones introducidos en la sección §4.6, aunque no debe confundirse con los mismos).

using System;  
using System.Collections;

public abstract class Expression  
{  
 public abstract double Evaluate(Hashtable vars);  
}

public class Constant: Expression  
{  
 double value;

public Constant(double value) {  
 this.value = value;  
 }

public override double Evaluate(Hashtable vars) {  
 return value;  
 }  
}

public class VariableReference: Expression  
{  
 string name;

public VariableReference(string name) {  
 this.name = name;  
 }

public override double Evaluate(Hashtable vars) {  
 object value = vars[name];  
 if (value == null) {  
 throw new Exception("Unknown variable: " + name);  
 }  
 return Convert.ToDouble(value);  
 }  
}

public class Operation: Expression  
{  
 Expression left;  
 char op;  
 Expression right;

public Operation(Expression left, char op, Expression right) {  
 this.left = left;  
 this.op = op;  
 this.right = right;  
 }

public override double Evaluate(Hashtable vars) {  
 double x = left.Evaluate(vars);  
 double y = right.Evaluate(vars);  
 switch (op) {  
 case '+': return x + y;  
 case '-': return x - y;  
 case '\*': return x \* y;  
 case '/': return x / y;  
 }  
 throw new Exception("Unknown operator");  
 }  
}

Las cuatro clases anteriores se pueden utilizar para modelar expresiones aritméticas. Por ejemplo, utilizando instancias de estas clases, la expresión x + 3 se puede representar como sigue.

Expression e = new Operation(  
 new VariableReference("x"),  
 '+',  
 new Constant(3));

El método Evaluate de una instancia Expression se invoca para evaluar la expresión determinada y generar un valor double. El método toma como argumento un Hashtable que contiene nombres de variable (como claves de las entradas) y valores (como valores de las entradas). El método Evaluate es un método abstracto virtual, lo que significa que deben invalidarlo las clases derivadas no abstractas para proporcionar una implementación real.

Una implementación de Constant de Evaluate simplemente devuelve la constante almacenada. Una implementación de VariableReference busca el nombre de la variable en la tabla hash y devuelve el valor resultante. Una implementación de Operation evalúa primero los operandos izquierdo y derecho (invocando de forma recursiva sus métodos Evaluate) y, a continuación, lleva a cabo la operación aritmética correspondiente.

El programa siguiente utiliza las clases Expression para evaluar la expresión x \* (y + 2) para distintos valores de x e y.

using System;  
using System.Collections;

class Test  
{  
 static void Main() {

Expression e = new Operation(  
 new VariableReference("x"),  
 '\*',  
 new Operation(  
 new VariableReference("y"),  
 '+',  
 new Constant(2)  
 )  
 );

Hashtable vars = new Hashtable();

vars["x"] = 3;  
 vars["y"] = 5;  
 Console.WriteLine(e.Evaluate(vars)); // Outputs "21"

vars["x"] = 1.5;  
 vars["y"] = 9;  
 Console.WriteLine(e.Evaluate(vars)); // Outputs "16.5"  
 }  
}

#### Sobrecarga de métodos

La sobrecarga de métodos permite que varios métodos de la misma clase tengan el mismo nombre siempre que sus signaturas sean únicas. Al compilar una invocación de un método sobrecargado, el compilador utiliza la resolución de sobrecarga para determinar el método concreto que se va a invocar. La resolución de sobrecarga busca el método que mejor se ajusta a los argumentos o informa de un error si no se puede encontrar ningún ajuste adecuado. El ejemplo siguiente muestra la resolución de sobrecarga en vigor. El comentario para cada invocación en el método Main muestra qué método se ha invocado.

class Test  
{  
 static void F() {  
 Console.WriteLine("F()");  
 }

static void F(object x) {  
 Console.WriteLine("F(object)");  
 }

static void F(int x) {  
 Console.WriteLine("F(int)");  
 }

static void F(double x) {  
 Console.WriteLine("F(double)");  
 }

static void F<T>(T x) {  
 Console.WriteLine("F<T>(T)");  
 }

static void F(double x, double y) {  
 Console.WriteLine("F(double, double)");  
 }

static void Main() {  
 F(); // Invokes F()  
 F(1); // Invokes F(int)  
 F(1.0); // Invokes F(double)  
 F("abc"); // Invokes F(object)  
 F((double)1); // Invokes F(double)  
 F((object)1); // Invokes F(object)  
 F<int>(1); // Invokes F<T>(T)  
 F(1, 1); // Invokes F(double, double) }  
}

Como muestra el ejemplo, un método determinado siempre puede ser seleccionado convirtiendo explícitamente los argumentos en los tipos de parámetros exactos y/o proporcionando explícitamente los argumentos de tipo.

### Otros miembros de función

Los miembros que contienen código ejecutable se conocen colectivamente como miembros de función de una clase. La sección anterior describe métodos, que son el tipo primario de miembros de función. En esta sección se describen los otros tipos de miembros de función admitidos por C#: constructores, propiedades, indizadores, eventos, operadores y destructores.

La tabla siguiente muestra una clase genérica llamada List<T> que implementa una lista creciente de objetos. La clase contiene varios ejemplos de los tipos más comunes de miembros de función.

|  |  |
| --- | --- |
| public class List<T> { | |
| const int defaultCapacity = 4; | Constante |
| T[] items;  int count; | Campos |
| public List(int capacity = defaultCapacity) {  items = new T[capacity];  } | Constructores |
| public int Count {  get { return count; }  }  public int Capacity {  get {  return items.Length;  }  set {  if (value < count) value = count;  if (value != items.Length) {  T[] newItems = new T[value];  Array.Copy(items, 0, newItems, 0, count);  items = newItems;  }  }  } | Propiedades |

|  |  |
| --- | --- |
| public T this[int index] {  get {  return items[index];  }  set {  items[index] = value;  OnChanged();  }  } | Indizador |
| public void Add(T item) {  if (count == Capacity) Capacity = count \* 2;  items[count] = item;  count++;  OnChanged();  }  protected virtual void OnChanged() {  if (Changed != null) Changed(this, EventArgs.Empty);  }  public override bool Equals(object other) {  return Equals(this, other as List<T>);  }  static bool Equals(List<T> a, List<T> b) {  if (a == null) return b == null;  if (b == null || a.count != b.count) return false;  for (int i = 0; i < a.count; i++) {  if (!object.Equals(a.items[i], b.items[i])) {  return false;  }  }  return true;  } | Métodos |
| public event EventHandler Changed; | Evento |
| public static bool operator ==(List<T> a, List<T> b) {  return Equals(a, b);  }  public static bool operator !=(List<T> a, List<T> b) {  return !Equals(a, b);  } | Operadores |
| } | |

#### Constructores

C# admite tanto constructores de instancia como estáticos. Un constructor de instancia es un miembro que implementa las acciones que se requieren para inicializar una instancia de una clase. Un constructor estático es un miembro que implementa las acciones exigidas para inicializar una clase cuando se carga por primera vez.

Un constructor se declara como método sin devolución de tipo y con el mismo nombre que la clase contenedora. Si una declaración de un constructor incluye un modificador static, declara un constructor estático. De lo contrario, declara un constructor de instancia.

Los constructores de instancia se pueden sobrecargar. Por ejemplo, la clase List<T> declara dos constructores de instancia, uno sin parámetros y otro que toma un parámetro int. Los constructores de instancia se invocan utilizando el operador new. Las siguientes instrucciones asignan dos instancias List<string> utilizando cada uno de los constructores de la clase List.

List<string> list1 = new List<string>();  
List<string> list2 = new List<string>(10);

Al contrario que otros miembros, los constructores de instancia no se heredan, y una clase sólo tiene los constructores de instancia realmente declarados en la clase. Si no se suministra ningún constructores de instancia para una clase, se proporcionará automáticamente uno vacío sin parámetros.

#### Propiedades

Las propiedades son una extensión natural de los campos. Los dos son miembros denominados con tipos asociados, y la sintaxis que se utiliza para el acceso a campos y propiedades es la misma. No obstante, a diferencia de los campos, las propiedades no denotan ubicaciones de almacenamiento. Las propiedades tienen descriptores de acceso que especifican las instrucciones que deben ejecutarse para leer o escribir sus valores.

Una propiedad se declara como un campo, solo que la declaración finaliza con un descriptor de acceso get y/o un descriptor de acceso set escrito entre los delimitadores { y } en lugar de finalizar con un punto y coma. Una propiedad que tiene un descriptor de acceso get y un set es una propiedad de lectura y escritura, una propiedad que tiene solo un descriptor de acceso get es una propiedad de solo lectura, y una propiedad que tiene solo un descriptor de acceso set es una propiedad de solo escritura.

Un descriptor de acceso get corresponde a un método sin parámetros que devuelve un valor del tipo de la propiedad. Exceptuando cuando se trata del destino de una asignación, al hacer referencia a una propiedad en una expresión, se invoca al descriptor de acceso get de la propiedad para calcular su valor.

Un descriptor de acceso set corresponde a un método con un parámetro único denominado value y ningún tipo de valor devuelto. Cuando se hace referencia a una propiedad como el destino de una asignación o como el operando de ++ o --, se invoca al descriptor de acceso set con un argumento que proporciona el nuevo valor.

La clase List<T> declara dos propiedades, Count y Capacity; la primera de ellas es de solo lectura y la segunda de lectura y escritura. El siguiente es un ejemplo del uso de estas propiedades.

List<string> names = new List<string>();  
names.Capacity = 100; // Invokes set accessor  
int i = names.Count; // Invokes get accessor  
int j = names.Capacity; // Invokes get accessor

Al igual que los campos y métodos, C# admite tanto las propiedades de instancia como las propiedades estáticas. Las propiedades estáticas se declaran con el modificador static, y las propiedades de instancias, sin él.

Los descriptores de acceso de una propiedad pueden ser virtuales. Cuando una declaración de propiedad incluye un modificador virtual, abstract u override, se aplica a los descriptores de acceso de la propiedad.

#### Indizadores

Un indizador es un miembro que permite indizar objetos de la misma manera que una matriz. Un indizador se declara como una propiedad, salvo en que el nombre del miembro sea this seguido de una lista de parámetros escritos entre delimitadores [ y ]. Los parámetros están disponibles en los descriptores de acceso del indizador. Al igual que las propiedades, los indizadores pueden ser de lectura y escritura, de sólo lectura y de sólo escritura, y los descriptores de acceso de un indizador pueden ser virtuales.

La clase List declara un indizador de lectura y escritura único que toma un parámetro int. El indizador permite indizar instancias List con valores int. Por ejemplo:

List<string> names = new List<string>();  
names.Add("Liz");  
names.Add("Martha");  
names.Add("Beth");  
for (int i = 0; i < names.Count; i++) {  
 string s = names[i];  
 names[i] = s.ToUpper();  
}

Los indizadores se pueden sobrecargar, lo que significa que pueden declararse varios indizadores siempre y cuando sus parámetros difieran en número o tipos.

#### Eventos

Un evento es un miembro que permite a una clase u objeto proporcionar notificaciones. Un evento se declara como un campo, salvo que la declaración incluya una palabra clave event y el tipo deba ser un tipo delegado.

En una clase que declara un miembro de evento, el evento se comporta como un campo de un tipo delegado (siempre que el evento no sea abstracto y no declare descriptores de acceso). El campo almacena una referencia a un delegado que representa los controladores de eventos que se han agregado al evento. Si no hay ningún controlador de eventos presente, el campo es null.

La clase List<T> declara un miembro de evento único llamado Changed, que indica que se ha agregado un nuevo elemento a la lista. El método virtual OnChanged inicia el evento Changed comprobando previamente que el evento es null (lo que significa que no hay ningún controlador presente). La noción de iniciar un evento equivale exactamente a invocar el delegado representado por el evento; por lo tanto, no hay construcciones especiales del lenguaje para producir eventos.

Los clientes reaccionan a los eventos a través de los controladores de eventos. Los controladores de eventos se asocian utilizando el operador += y se quitan utilizando el operador -=. El ejemplo siguiente asocia un controlador de eventos al evento Changed de List<string>.

using System;

class Test  
{  
 static int changeCount;

static void ListChanged(object sender, EventArgs e) {  
 changeCount++;  
 }

static void Main() {  
 List<string> names = new List<string>();  
 names.Changed += new EventHandler(ListChanged);  
 names.Add("Liz");  
 names.Add("Martha");  
 names.Add("Beth");  
 Console.WriteLine(changeCount); // Outputs "3"  
 }  
}

Para escenarios avanzados donde se busca el control del almacenamiento subyacente de un evento, una declaración de evento puede proporcionar explícitamente descriptores de acceso add y remove que en cierto modo son similares al descriptor de acceso set de una propiedad.

#### Operadores

Un operador es un miembro que define el significado de aplicar un operador de expresión determinado a las instancias de una clase. Pueden definirse tres categorías de operadores: operadores unarios, operadores binarios y operadores de conversión. Todos los operadores deberán declararse como public y static.

La clase List<T> declara dos operadores operator == y operator !=, proporcionando de este modo un nuevo significado a expresiones que aplican dichos operadores a instancias List. Específicamente, los operadores definen la igualdad de dos instancias List<T> comparando todos los objetos contenidos que utilizan sus métodos Equals. El ejemplo siguiente utiliza el operador == para comparar dos instancias List<int>.

using System;

class Test  
{  
 static void Main() {  
 List<int> a = new List<int>();  
 a.Add(1);  
 a.Add(2);  
 List<int> b = new List<int>();  
 b.Add(1);  
 b.Add(2);  
 Console.WriteLine(a == b); // Outputs "True"   
 b.Add(3);  
 Console.WriteLine(a == b); // Outputs "False"  
 }  
}

El resultado del primer Console.WriteLine es True porque las dos listas contienen el mismo número de objetos con los mismos valores en el mismo orden. Si List<T> no tuviera definido operator ==, la primera Console.WriteLine tendría el resultado False porque a y b hacen referencia a diferentes instancias List<int>.

#### Destructores

Un destructor es un miembro que implementa las acciones necesarias para destruir una instancia de una clase. Los destructores no pueden tener parámetros, no pueden tener modificadores de accesibilidad y no se pueden invocar de forma explícita. El destructor de una instancia se invoca automáticamente durante la recolección de elementos no utilizados.

El recolector de elementos no utilizados tiene una amplia libertad para decidir cuándo debe recolectar objetos y ejecutar destructores. En concreto, el control de tiempo de las invocaciones del destructor no es determinista y los destructores se pueden ejecutar en cualquier subproceso. Por estas y otras razones, las clases sólo deberían implementar los destructores cuando no es factible ninguna otra solución.

La instrucción using constituye una alternativa más recomendable para la destrucción de objetos.

## Structs

Como las clases, los structs son estructuras de datos que pueden contener miembros de datos y miembros de función, pero al contrario que las clases, los structs son tipos de valor y no requieren asignación del montón. Una variable de un tipo struct almacena directamente los datos de la estructura, mientras que una variable de un tipo de clase almacena una referencia a un objeto dinámicamente asignado. Los tipos struct no admiten la herencia especificada por el usuario y heredan implícitamente del tipo object.

Los structs son particularmente útiles para estructuras de datos pequeñas que tienen semánticas de valor. Los números complejos, los puntos de un sistema de coordenadas o los pares clave-valor de un diccionario son buenos ejemplos de structs. El uso de structs en lugar de clases para estructuras de datos pequeñas puede suponer una diferencia sustancial en el número de asignaciones de memoria que realiza una aplicación. Por ejemplo, el programa siguiente crea e inicializa una matriz de 100 puntos. Con Point implementado como una clase, se inicializan 101 objetos separados, uno para la matriz y uno para cada uno de los 100 elementos.

class Point  
{  
 public int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

class Test  
{  
 static void Main() {  
 Point[] points = new Point[100];  
 for (int i = 0; i < 100; i++) points[i] = new Point(i, i);  
 }  
}

Una alternativa es convertir Point en un struct.

struct Point  
{  
 public int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

Ahora, sólo se crea una instancia de un objeto, el de la matriz, y las instancias de Point se almacenan en línea en la matriz.

Los constructores de struct se invocan con el operador new, pero esto no implica que se esté asignando la memoria. En lugar de asignar dinámicamente un objeto y devolver una referencia al mismo, un constructor de struct simplemente devuelve el propio valor de la estructura (normalmente en una ubicación temporal en la pila) y a continuación se copia este valor como sea necesario.

En el caso de las clases, es posible que dos variables hagan referencia al mismo objeto y, por tanto, que las operaciones en una variable afecten al objeto al que hace referencia la otra variable. En el caso de los structs, cada variable tiene su propia copia de los datos, de manera que no es posible que las operaciones de una afecten a la otra. Por ejemplo, el resultado generado por el fragmento de código siguiente depende de si Point es una clase o un struct.

Point a = new Point(10, 10);  
Point b = a;  
a.x = 20;  
Console.WriteLine(b.x);

Si Point es una clase, el resultado es 20 porque a y b hacen referencia al mismo objeto. Si Point es un struct, el resultado es 10 porque la asignación de a a b crea una copia del valor y esta copia no resulta afectada por la subsiguiente asignación a a.x.

El ejemplo anterior resalta dos de las limitaciones de structs. En primer lugar, copiar un struct completo normalmente es menos eficaz que copiar una referencia a un objeto, por lo que la asignación y el modo de pasar parámetros de valor puede ser más costoso con estructuras que con tipos de referencia. En segundo lugar, salvo en el caso de los parámetros ref y out, no es posible crear referencias a structs, lo que descarta su uso en varias situaciones.

## Matrices

Una matriz es una estructura de datos que contiene una serie de variables a las que se obtiene acceso a través de índices calculados. Las variables contenidas en una matriz, también conocidas como elementos de la matriz, son todas del mismo tipo, denominado tipo de elemento de la matriz.

Los tipos de matriz son tipos de referencia, por lo que la declaración de una variable matricial solamente reserva espacio para una instancia de matriz. Las instancias de matriz reales se crean dinámicamente en tiempo de ejecución utilizando el operador new. La operación new especifica la longitud de la nueva instancia de matriz, que se establece para el período de duración de la instancia. Los índices de los elementos de una matriz pueden ir desde 0 a Length - 1. El operador new inicializa automáticamente los elementos de una matriz con su valor predeterminado que, por ejemplo, es cero para todos los tipos numéricos y null para todos los tipos de referencia.

El ejemplo siguiente crea una matriz de elementos int, inicializa la matriz e imprime su contenido.

using System;

class Test  
{  
 static void Main() {  
 int[] a = new int[10];  
 for (int i = 0; i < a.Length; i++) {  
 a[i] = i \* i;  
 }  
 for (int i = 0; i < a.Length; i++) {  
 Console.WriteLine("a[{0}] = {1}", i, a[i]);  
 }  
 }  
}

Este ejemplo crea una matriz unidimensional y opera en ella. C# también admite las matrices multidimensionales. El número de dimensiones de un tipo de matriz, también conocido como el rango del tipo de matriz, es uno más el número de comas escrito entre los corchetes del tipo de matriz. El ejemplo siguiente asigna una matriz unidimensional, una bidimensional y una tridimensional.

int[] a1 = new int[10];  
int[,] a2 = new int[10, 5];  
int[,,] a3 = new int[10, 5, 2];

La matriz a1 contiene 10 elementos, la matriz a2 contiene 50 (10 × 5) elementos y la matriz a3 contiene 100 (10 × 5 × 2) elementos.

Los elementos de una matriz pueden ser de cualquier tipo, incluido un tipo de matriz. Una matriz con elementos de tipo de matriz a veces se denomina matriz escalonada porque las longitudes de las matrices de elementos no tienen que ser siempre las mismas. El ejemplo siguiente asigna una matriz de matrices de int:

int[][] a = new int[3][];  
a[0] = new int[10];  
a[1] = new int[5];  
a[2] = new int[20];

La primera línea crea una matriz con tres elementos, cada uno de tipo int[] y cada uno con un valor inicial null. A continuación, las líneas subsiguientes inicializan los tres elementos con referencias a instancias de matriz individuales de longitudes diversas.

El operador new permite especificar los valores iniciales de los elementos de matriz utilizando un inicializador de matrices, que es una lista de expresiones escrita entre los delimitadores { y }. El ejemplo siguiente asigna e inicializa un int[] con tres elementos.

int[] a = new int[] {1, 2, 3};

Observe que la longitud de la matriz se infiere del número de expresiones existentes entre { y }. La variable local y las declaraciones de campo se pueden acortar más, de tal modo que no es necesario volver a especificar el tipo de matriz.

int[] a = {1, 2, 3};

Los dos ejemplos anteriores son equivalentes a lo siguiente:

int[] t = new int[3];  
t[0] = 1;  
t[1] = 2;  
t[2] = 3;  
int[] a = t;

## Interfaces

Una interfaz define un contrato que puede ser implementado por clases y structs. Una interfaz puede contener métodos, propiedades, eventos e indizadores. Una interfaz no proporciona implementaciones de los miembros que define, simplemente especifica los miembros que deben proporcionar las clases o structs que implementan la interfaz.

Las interfaces pueden utilizar una herencia múltiple. En el ejemplo siguiente, la interfaz IComboBox hereda de ITextBox y de IListBox.

interface IControl  
{  
 void Paint();  
}

interface ITextBox: IControl  
{  
 void SetText(string text);  
}

interface IListBox: IControl  
{  
 void SetItems(string[] items);  
}

interface IComboBox: ITextBox, IListBox {}

Las clases y los structs pueden implementar varias interfaces. En el ejemplo siguiente, la clase EditBox implementa IControl e IDataBound.

interface IDataBound  
{  
 void Bind(Binder b);  
}

public class EditBox: IControl, IDataBound  
{  
 public void Paint() {...}

public void Bind(Binder b) {...}  
}

Cuando una clase o un struct implementa una interfaz determinada, las instancias de dicha clase o struct se pueden convertir implícitamente en dicho tipo de interfaz. Por ejemplo:

EditBox editBox = new EditBox();  
IControl control = editBox;  
IDataBound dataBound = editBox;

En los casos en los que no se sabe estáticamente que una interfaz pueda implementar una interfaz determinada, se pueden utilizar conversiones de tipo dinámicas. Por ejemplo, las instrucciones siguientes utilizan conversiones de tipo dinámicas para obtener implementaciones de interfaces IControl e IDataBound de un objeto. Dado que el tipo real del objeto es EditBox, las conversiones se realizaron con éxito.

object obj = new EditBox();  
IControl control = (IControl)obj;  
IDataBound dataBound = (IDataBound)obj;

En la clase EditBox anterior, el método Paint de la interfaz IControl y el método Bind de la interfaz IDataBound se implementan utilizando miembros public. C# también admite implementaciones explícitas de miembro de la interfaz y, al utilizarlas la clase o el struct, se puede evitar que los miembros sean public. Una implementación explícita de miembro de interfaz se escribe utilizando el nombre completo de miembro de interfaz. Por ejemplo, la clase EditBox podría implementar los métodos IControl.Paint y IDataBound.Bind utilizando implementaciones de explícitas de miembro de interfaz de la forma siguiente.

public class EditBox: IControl, IDataBound  
{  
 void IControl.Paint() {...}

void IDataBound.Bind(Binder b) {...}  
}

Sólo se puede tener acceso a los miembros de interfaz explícitos mediante el tipo de interfaz. Por ejemplo, la implementación de IControl.Paint proporcionada por la clase EditBox anterior solo se puede invocar convirtiendo primero la referencia EditBox en el tipo de interfaz IControl.

EditBox editBox = new EditBox();  
editBox.Paint(); // Error, no such method  
IControl control = editBox;  
control.Paint(); // Ok

## Enumeraciones

Un tipo enum (enumeración) es un tipo de valor distinto con un conjunto de constantes con nombre. El ejemplo siguiente declara y utiliza un tipo enum denominado Color con tres valores constantes: Red, Green y Blue.

using System;

enum Color  
{  
 Red,  
 Green,  
 Blue  
}

class Test  
{  
 static void PrintColor(Color color) {  
 switch (color) {  
 case Color.Red:  
 Console.WriteLine("Red");  
 break;  
 case Color.Green:  
 Console.WriteLine("Green");  
 break;  
 case Color.Blue:  
 Console.WriteLine("Blue");  
 break;  
 default:  
 Console.WriteLine("Unknown color");  
 break;  
 }  
 }

static void Main() {  
 Color c = Color.Red;  
 PrintColor(c);  
 PrintColor(Color.Blue);  
 }  
}

Cada tipo enum tiene un tipo entero correspondiente denominado tipo subyacente del tipo enum. Un tipo enum que no declara explícitamente un tipo subyacente tiene int como tipo subyacente. El formato de almacenamiento y el intervalo de posibles valores de un tipo enum vienen determinados por su tipo subyacente. El conjunto de valores que puede tomar un tipo enum no está limitado por sus miembros de enumeración. En concreto, cualquier valor del tipo subyacente de una enumeración puede convertirse en el tipo enum, y es un valor aceptado y distintivo de dicho tipo enum.

El ejemplo siguiente declara un tipo enum Alignment con un tipo subyacente de sbyte.

enum Alignment: sbyte  
{  
 Left = -1,  
 Center = 0,  
 Right = 1  
}

Como se muestra en el ejemplo anterior, una declaración de miembro de enumeración puede incluir una expresión constante que especifica el valor del miembro. El valor constante de cada miembro de enumeración debe estar comprendido en el intervalo del tipo subyacente de la enumeración. Cuando una declaración de miembro de enumeración no especifica explícitamente un valor, se da al miembro el valor cero (si es el primer miembro del tipo enum) o el valor del miembro de la enumeración precedente más uno.

Los valores de enumeración se pueden convertir en valores de tipo integral y viceversa, utilizando las conversiones de tipo. Por ejemplo:

int i = (int)Color.Blue; // int i = 2;  
Color c = (Color)2; // Color c = Color.Blue;

El valor predeterminado de cualquier tipo de enumeración es el valor cero de tipo integral convertido en el tipo enum. En los casos en los que las variables se inicializan automáticamente con un valor predeterminado, éste es el valor proporcionado para las variables de tipos enum. Para que el valor predeterminado de un tipo enum esté fácilmente disponible, el literal 0 se convierte implícitamente en cualquier tipo enum. De este modo, se permite lo siguiente.

Color c = 0;

## Delegados

Un tipo delegado representa las referencias a los métodos con una lista de parámetros determinada y un tipo de valor devuelto. Los delegados permiten tratar métodos como entidades, que se pueden asignar a las variables y pasarse como parámetros. Los delegados son similares al concepto de punteros a función de otros lenguajes, pero al contrario de los punteros a función, los delegados están orientados a objetos y proporcionan seguridad de tipos.

El ejemplo siguiente declara y utiliza un tipo delegado denominado Function.

using System;

delegate double Function(double x);

class Multiplier  
{  
 double factor;

public Multiplier(double factor) {  
 this.factor = factor;  
 }

public double Multiply(double x) {  
 return x \* factor;  
 }  
}

class Test  
{  
 static double Square(double x) {  
 return x \* x;  
 }

static double[] Apply(double[] a, Function f) {  
 double[] result = new double[a.Length];  
 for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);  
 return result;  
 }

static void Main() {  
 double[] a = {0.0, 0.5, 1.0};

double[] squares = Apply(a, Square);

double[] sines = Apply(a, Math.Sin);

Multiplier m = new Multiplier(2.0);  
 double[] doubles = Apply(a, m.Multiply);  
 }  
}

Una instancia del tipo delegado Function puede hacer referencia a cualquier método que tome un argumento double y que devuelva un valor double. El método Apply aplica un tipo Function determinado a los elementos de un tipo double[] y devuelve un tipo double[] con los resultados. En el método Main, se utiliza Apply para aplicar tres funciones diferentes a un double[].

Un delegado puede hacer referencia a un método estático (como Square o Math.Sin en el ejemplo anterior) o a un método de instancia (como m.Multiply en el ejemplo anterior). Un delegado que hace referencia a un método de instancia también hace referencia a un objeto determinado y, cuando se invoca el método de instancia a través del delegado, dicho objeto pasa a ser this en la invocación.

Los delegados también se pueden crear mediante funciones anónimas, que son “métodos insertados” creados sobre la marcha. Las funciones anónimas pueden ver las variables locales de los métodos adyacentes. Así, el ejemplo anterior del multiplicador se puede escribir más fácilmente sin usar una clase Multiplier:

double[] doubles = Apply(a, (double x) => x \* 2.0);

Una propiedad interesante y útil de un delegado es que no conoce ni necesita conocer la clase del método a la que hace referencia; lo único que importa es que el método al que se hace referencia tenga los mismos parámetros y el mismo tipo de valor devuelto que el delegado.

## Atributos

Los tipos, miembros y otras entidades de un programa de C# admiten modificadores que controlan ciertos aspectos de su comportamiento. Por ejemplo, la accesibilidad de un método se controla mediante los modificadores public, protected, internal y private . C# generaliza esta función de tal forma que los tipos definidos por el usuario de información declarativa puedan agregarse a las entidades del programa y ser recuperados en tiempo de ejecución. Los programas especifican esta información declarativa adicional definiendo y utilizando atributos.

El ejemplo siguiente declara un atributo HelpAttribute que se puede colocar en entidades de programa para proporcionar vínculos a la documentación asociada.

using System;

public class HelpAttribute: Attribute  
{  
 string url;  
 string topic;

public HelpAttribute(string url) {  
 this.url = url;  
 }

public string Url {   
 get { return url; }  
 }

public string Topic {  
 get { return topic; }  
 set { topic = value; }  
 }  
}

Todas las clases de atributos derivan de la clase base System.Attribute proporcionada por .NET Framework. Los atributos se pueden aplicar indicando su nombre, junto con los argumentos, entre corchetes, justo antes de la declaración asociada. Si el nombre de un atributo termina en Attribute, se puede omitir dicha parte del nombre si se hace referencia al atributo. Por ejemplo, el atributo HelpAttribute se puede utilizar del siguiente modo.

[Help("http://msdn.microsoft.com/.../MyClass.htm")]  
public class Widget  
{  
 [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]  
 public void Display(string text) {}  
}

Este ejemplo adjunta un HelpAttribute a la clase Widget y otro HelpAttribute al método Display de la clase. Los constructores públicos de una clase de atributos controlan la información que debe proporcionarse al asociar el atributo a una entidad de programa. Puede proporcionarse información adicional haciendo referencia a las propiedades públicas de lectura y escritura de la clase de atributos (como la referencia previa a la propiedad Topic).

El ejemplo siguiente muestra cómo utilizar la reflexión para recuperar en tiempo de ejecución la información de atributos para una entidad determinada del programa.

using System;  
using System.Reflection;

class Test  
{  
 static void ShowHelp(MemberInfo member) {  
 HelpAttribute a = Attribute.GetCustomAttribute(member,  
 typeof(HelpAttribute)) as HelpAttribute;  
 if (a == null) {  
 Console.WriteLine("No help for {0}", member);  
 }  
 else {  
 Console.WriteLine("Help for {0}:", member);  
 Console.WriteLine(" Url={0}, Topic={1}", a.Url, a.Topic);  
 }  
 }

static void Main() {  
 ShowHelp(typeof(Widget));  
 ShowHelp(typeof(Widget).GetMethod("Display"));  
 }  
}

Cuando se solicita un atributo determinado a través de la reflexión, se invoca al constructor para la clase de atributos con la información proporcionada en el código fuente del programa y se devuelve la instancia de atributo resultante. Si la información adicional se proporciona a través de propiedades, dichas propiedades se establecen en los valores dados antes de que se devuelva la instancia de atributo.

# Estructura léxica

## Programas

Un programa de C# consta de uno o varios archivos de código fuente, formalmente conocidos como unidades de compilación (§9.1). Un archivo de código fuente es una secuencia ordenada de caracteres Unicode. Los archivos de código fuente normalmente tienen una correspondencia de uno a uno con los archivos de un sistema de archivos, pero esta correspondencia no es necesaria. Para maximizar la portabilidad, se recomienda utilizar la codificación UTF-8 para los archivos de un sistema de archivos.

En términos conceptuales, un programa se compila en tres fases:

1. Transformación, que convierte un archivo, a partir de un repertorio de caracteres y un esquema de codificación concretos, en una secuencia de caracteres Unicode.
2. Análisis léxico, que convierte una secuencia de caracteres de entrada Unicode en una secuencia de símbolos (token).
3. Análisis sintáctico, que convierte la secuencia de tokens en código ejecutable.

## Gramáticas

Esta especificación presenta la sintaxis del lenguaje de programación C# mediante el uso de dos gramáticas. La gramática léxica (§2.2.2) define cómo se combinan los caracteres Unicode para formar terminadores de línea, espacios en blanco, comentarios, símbolos (tokens) y directivas de preprocesamiento. La gramática sintáctica (§2.2.3) define cómo se combinan los tokens resultantes de la gramática léxica para formar programas de C#.

### Notación gramatical

Las gramáticas léxica y sintáctica se presentan mediante producciones gramaticales. Cada producción gramatical define un símbolo no terminal y las posibles expansiones de dicho símbolo en secuencias de símbolos no terminales o terminales. En las producciones gramaticales, los símbolos no terminales se muestran en cursiva, y los símbolos terminal se muestran en una fuente de ancho fijo.

La primera línea de una producción gramatical es el nombre del símbolo no terminal que se define, seguido de un carácter de dos puntos. Cada línea con sangría sucesiva contiene una expansión posible del símbolo no terminal dado como una secuencia de símbolos no terminales o terminales. Por ejemplo, la producción:

while-statement:  
while ( boolean-expression ) embedded-statement

define una instrucción while formada por el token while, seguida del símbolo “(”, de una expresión booleana, del símbolo “)” y de una instrucción incrustada.

Cuando hay más de una expansión posible de un símbolo no terminal, las alternativas se muestran en líneas individuales. Por ejemplo, la producción:

statement-list:  
statement  
statement-list statement

define una lista de instrucciones que consta de una instrucción o de una lista de instrucciones seguida de una instrucción. Es decir, la definición es recursiva y especifica que una lista de instrucciones consta de una o varias instrucciones.

Se utiliza un sufijo en subíndice “opt” para indicar un símbolo opcional. La producción:

block:  
{ statement-listopt }

es una forma abreviada de:

block:  
{ }  
{ statement-list }

y define un bloque (block) compuesto por una lista de instrucciones (statement-list) opcional entre los tokens “{” y “}”.

Las alternativas normalmente se enumeran en líneas distintas, aunque en los casos en los que hay varias alternativas, el texto “una de las siguientes” puede preceder a una lista de expansiones proporcionadas en la misma línea. Esto es sencillamente una forma abreviada de mostrar todas las alternativas en líneas distintas. Por ejemplo, la producción:

real-type-suffix: one of  
F f D d M m

es una forma abreviada de:

real-type-suffix:  
F  
f  
D  
d  
M  
m

### Gramática léxica

La gramática léxica de C# se presenta en §2.3, §2.4 y §2.5. Los símbolos terminales de la gramática léxica son los caracteres del juego de caracteres Unicode, y la gramática léxica especifica cómo se combinan los caracteres para formar tokens (§2.4), espacios en blanco (§2.3.3), comentarios (§2.3.2) y directivas de preprocesamiento (§2.5).

Todos los archivos de código fuente de un programa de C# deben ajustarse a la producción entrada (input) de la gramática léxica (§2.3).

### Gramática sintáctica

La gramática sintáctica de C# se presenta en los capítulos y apéndices que siguen a este capítulo. Los símbolos terminales de la gramática sintáctica son los tokens definidos por la gramática léxica, mientras la gramática sintáctica es la que especifica cómo se combinan los símbolos para formar programas de C#.

Todos los archivos de código fuente de un programa de C# deben cumplir con la producción de unidades de compilación (compilation-unit) de la gramática sintáctica (§9.1).

## Análisis léxico

La producción entrada (input) define la estructura léxica de un archivo de código fuente de C#. Un archivo de código fuente de un programa de C# debe ajustarse a esta producción de la gramática léxica.

input:  
input-sectionopt

input-section:  
input-section-part  
input-section input-section-part

input-section-part:  
input-elementsopt new-line  
pp-directive

input-elements:  
input-element  
input-elements input-element

input-element:  
whitespace  
comment  
token

Cinco elementos básicos constituyen la estructura léxica de un archivo de código fuente de C#: los terminadores de línea (§2.3.1), el espacio en blanco (§2.3.3), los comentarios (§2.3.2), los símbolos (token) (§2.4) y las directivas de preprocesamiento (§2.5). De estos elementos básicos, solamente los tokens son significativos en la gramática sintáctica de un programa de C# (§2.2.3).

El procesamiento léxico de un archivo de código fuente de C# consiste en reducir el archivo a una secuencia de tokens para formar la entrada del análisis sintáctico. Los terminadores de línea, el espacio en blanco y los comentarios pueden corresponder a tokens diferentes, y las directivas de preprocesamiento pueden hacer que se omitan secciones del archivo de código fuente, pero por lo demás estos elementos léxicos no tienen repercusiones en la estructura sintáctica de un programa de C#.

Cuando varias producciones de la gramática léxica coinciden con una secuencia de caracteres de un archivo de código fuente, el procesamiento léxico siempre forma el elemento léxico más largo posible. Por ejemplo, la secuencia de caracteres // se procesa como el principio de un comentario en una sola línea, porque este elemento léxico es más largo que un token de una sola /.

### Terminadores de línea

Los terminadores de línea dividen los caracteres de un archivo de código fuente de C# en líneas.

new-line:  
Carriage return character (U+000D)  
Line feed character (U+000A)  
Carriage return character (U+000D) followed by line feed character (U+000A)  
Next line character (U+0085)  
Line separator character (U+2028)  
Paragraph separator character (U+2029)

Por compatibilidad con las herramientas de edición de código fuente que agregan marcadores de fin de archivo, y para permitir que un archivo de código fuente pueda verse como una secuencia de líneas terminadas correctamente, se aplican las siguientes transformaciones, por orden, a todos los archivos de código fuente de un programa de C#:

* Si el último carácter del archivo de código fuente es un carácter Control-Z (U+001A), el carácter se elimina.
* Al final del archivo de código de fuente se agrega un carácter de retorno de carro (U+000D) si este archivo no está vacío y si su último carácter no es un retorno de carro (U+000D), un salto de línea (U+000A), un separador de línea (U+2028) o un separador de párrafo (U+2029).

### Comentarios

Se aceptan dos formas de comentarios: comentarios en una línea y comentarios delimitados. Los comentarios de una sola línea empiezan con los caracteres // y se extienden hasta el final de la línea de código fuente. Los comentarios delimitados empiezan con los caracteres /\* y terminan con los caracteres \*/. Estos comentarios pueden estar en varias líneas.

comment:  
single-line-comment  
delimited-comment

single-line-comment:  
// input-charactersopt

input-characters:  
input-character  
input-characters input-character

input-character:  
Any Unicode character except a new-line-character

new-line-character:  
Carriage return character (U+000D)  
Line feed character (U+000A)  
Next line character (U+0085)  
Line separator character (U+2028)  
Paragraph separator character (U+2029)

delimited-comment:  
/\* delimited-comment-textopt asterisks /

delimited-comment-text:  
delimited-comment-section  
delimited-comment-text delimited-comment-section

delimited-comment-section:  
/  
asterisksopt not-slash-or-asterisk

asterisks:  
\*  
asterisks \*

not-slash-or-asterisk:  
Any Unicode character except / or \*

Los comentarios no pueden anidarse. Las secuencias de caracteres /\* y \*/ no tienen ningún significado especial dentro de un comentario //, y las secuencias de caracteres // y /\* no tienen ningún significado especial dentro de un comentario delimitado.

Los comentarios no se procesan dentro de literales de carácter y de cadena.

El ejemplo

/\* Hello, world program  
 This program writes “hello, world” to the console  
\*/  
class Hello  
{  
 static void Main() {  
 System.Console.WriteLine("hello, world");  
 }  
}

incluye un comentario delimitado.

El ejemplo

// Hello, world program  
// This program writes “hello, world” to the console  
//  
class Hello // any name will do for this class  
{  
 static void Main() { // this method must be named "Main"  
 System.Console.WriteLine("hello, world");  
 }  
}

muestra varios comentarios en una línea.

### Espacio en blanco

Espacio en blanco se define como cualquier carácter con la clase Unicode Zs (que incluye el carácter de espacio) así como el carácter de tabulación horizontal, el carácter de tabulación vertical y el carácter de avance de página.

whitespace:  
Any character with Unicode class Zs  
Horizontal tab character (U+0009)  
Vertical tab character (U+000B)  
Form feed character (U+000C)

## Tokens

Existen varias clases de símbolos (token): identificadores, palabras clave, literales, operadores y signos de puntuación. Los espacios en blanco y los comentarios no son símbolos (token), si bien actúan como separadores entre ellos.

token:  
identifier  
keyword  
integer-literal  
real-literal  
character-literal  
string-literal  
operator-or-punctuator

### Secuencias de escape de caracteres Unicode

Una secuencia de escape de caracteres Unicode representa un carácter Unicode. Las secuencias de escape de carácter Unicode se procesan en identificadores (§2.4.2), literales de carácter (§2.4.4.4) y literales de cadena normales (§2.4.4.5). Un carácter de escape Unicode no se procesa en ninguna otra ubicación (por ejemplo, para formar un operador, un signo de puntuación o una palabra clave).

unicode-escape-sequence:  
\u hex-digit hex-digit hex-digit hex-digit  
\U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit

Una secuencia de escape Unicode representa el único carácter Unicode formado por el número hexadecimal que sigue a los caracteres “\u” o “\U”. Dado que C# usa una codificación de 16 bits de los puntos de código Unicode en caracteres y valores de cadena, un carácter Unicode en el intervalo de U+10000 a U+10FFFF no está permitido en un literal de caracteres y se representa mediante un par Unicode suplente en un literal de cadena. Los caracteres Unicode con puntos de código por encima de 0x10FFFF no se aceptan.

No se llevan a cabo conversiones múltiples. Por ejemplo, el literal de cadena “\u005Cu005C” es equivalente a “\u005C”, no a “\”. El valor Unicode \u005C es el carácter “\”.

El ejemplo

class Class1  
{  
 static void Test(bool \u0066) {  
 char c = '\u0066';  
 if (\u0066)  
 System.Console.WriteLine(c.ToString());  
 }   
}

muestra varios usos de \u0066, que es la secuencia de escape de la letra “f”. El programa equivale a

class Class1  
{  
 static void Test(bool f) {  
 char c = 'f';  
 if (f)  
 System.Console.WriteLine(c.ToString());  
 }   
}

### Identificadores

Las reglas de los identificadores explicadas en esta sección se corresponden exactamente con las recomendadas por la norma Unicode Anexo 31, a excepción del carácter de subrayado, que está permitido como carácter inicial (como era tradicional en el lenguaje de programación C). Las secuencias de escape Unicode están permitidas en los identificadores y el carácter “@” está permitido como prefijo para habilitar el uso de palabras clave como identificadores.

identifier:  
available-identifier  
@ identifier-or-keyword

available-identifier:  
An identifier-or-keyword that is not a keyword

identifier-or-keyword:  
identifier-start-character identifier-part-charactersopt

identifier-start-character:  
letter-character  
\_ (the underscore character U+005F)

identifier-part-characters:  
identifier-part-character  
identifier-part-characters identifier-part-character

identifier-part-character:  
letter-character  
decimal-digit-character  
connecting-character  
combining-character  
formatting-character

letter-character:  
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl   
A unicode-escape-sequence representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:  
A Unicode character of classes Mn or Mc   
A unicode-escape-sequence representing a character of classes Mn or Mc

decimal-digit-character:  
A Unicode character of the class Nd   
A unicode-escape-sequence representing a character of the class Nd

connecting-character:   
A Unicode character of the class Pc  
A unicode-escape-sequence representing a character of the class Pc

formatting-character:   
A Unicode character of the class Cf  
A unicode-escape-sequence representing a character of the class Cf

Para obtener más información sobre las anteriores clases de caracteres, consulte la sección 4.5 de la versión 3.0 de la norma Unicode (Unicode Standard, Version 3.0).

Entre los ejemplos de identificadores válidos se encuentran “identifier1”, “\_identifier2” y “@if”.

Los identificadores incluidos en programas según la norma deberán seguir el formato canónico definido en el documento Unicode Normalization Form C (anexo 15 de la norma Unicode). El comportamiento ante identificadores que no se ajusten a dicho formato dependerá de la implementación; no obstante, no serán necesarios diagnósticos.

El prefijo “@” habilita el uso de palabras clave como identificadores, lo cual resulta útil cuando se interactúa con otros lenguajes de programación. El carácter @ en realidad no forma parte del identificador, por lo que el identificador podría considerarse en otros lenguajes como un identificador normal, sin el prefijo. Los identificadores con prefijo @ se conocen como identificadores textuales. El uso del prefijo @ para los identificadores que no son palabras clave está permitido, pero no se recomienda por cuestión de estilo.

El ejemplo:

class @class  
{  
 public static void @static(bool @bool) {  
 if (@bool)  
 System.Console.WriteLine("true");  
 else  
 System.Console.WriteLine("false");  
 }   
}

class Class1  
{  
 static void M() {  
 cl\u0061ss.st\u0061tic(true);  
 }  
}

define una clase denominada “class” con un método estático denominado “static” que toma un parámetro denominado “bool”. Tenga en cuenta que, dado que los escapes de Unicode no están permitidos en las palabras clave, el token “cl\u0061ss” es un identificador idéntico a “@class”.

Dos identificadores se consideran el mismo si son idénticos después de aplicarse las siguientes transformaciones, por orden:

* El prefijo “@”, si aparece, se quitará.
* Cada secuencia de escape Unicode (unicode-escape-sequence) se transformará en su carácter Unicode correspondiente.
* Todos los caracteres de formato (formatting-characters) serán quitados.

Los identificadores que contienen dos caracteres de subrayado (U+005F) consecutivos se reservan para uso de la implementación. Por ejemplo, una implementación podría proporcionar palabras clave extendidas que empiecen por dos caracteres de subrayado.

### Palabras clave

Una palabra clave es una secuencia de caracteres similar a un identificador que está reservada y no puede utilizarse como identificador excepto cuando tiene como prefijo el carácter @.

keyword: one of  
abstract as base bool break  
byte case catch char checked  
class const continue decimal default  
delegate do double else enum  
event explicit extern false finally  
fixed float for foreach goto  
if implicit in int interface  
internal is lock long namespace  
new null object operator out  
override params private protected public  
readonly ref return sbyte sealed  
short sizeof stackalloc static string  
struct switch this throw true  
try typeof uint ulong unchecked  
unsafe ushort using virtual void  
volatile while

En algunos lugares de la gramática, hay identificadores específicos con significados especiales, pero no son palabras clave. Estos identificadores se denominan a veces “palabras clave contextuales”. Por ejemplo, en una declaración de propiedad, los identificadores “get” y “set” tienen un significado especial (§10.7.2). En estas ubicaciones no se permite usar identificadores distintos de get o set, por lo que este uso no presenta conflictos si se utilizan estas palabras como identificadores. En otros casos, por ejemplo con el identificador “var” en declaraciones de variables locales con tipo implícito (§8.5.1), las palabras clave contextuales pueden entrar en conflicto con los nombres declarados. En esos casos, el nombre declarado tiene prioridad sobre el uso del identificador como palabra clave contextual.

### Literales

Un literal es una representación en código fuente de un valor.

literal:  
boolean-literal  
integer-literal  
real-literal  
character-literal  
string-literal  
null-literal

#### Literales booleanos

Existen dos valores literales booleanos: true y false.

boolean-literal:  
true  
false

El tipo de un literal booleano (boolean-literal) es bool.

#### Literales enteros

Los literales enteros permiten escribir valores de los tipos int, uint, long y ulong. Los literales enteros tienen dos formatos posibles: decimal y hexadecimal.

integer-literal:  
decimal-integer-literal  
hexadecimal-integer-literal

decimal-integer-literal:  
decimal-digits integer-type-suffixopt

decimal-digits:  
decimal-digit  
decimal-digits decimal-digit

decimal-digit: one of  
0 1 2 3 4 5 6 7 8 9

integer-type-suffix: one of  
U u L l UL Ul uL ul LU Lu lU lu

hexadecimal-integer-literal:  
0x hex-digits integer-type-suffixopt  
0X hex-digits integer-type-suffixopt

hex-digits:  
hex-digit  
hex-digits hex-digit

hex-digit: one of  
0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

El tipo de un literal entero se determina como sigue:

* Si el literal no tiene sufijo, su tipo es el primero de los tipos en los cuales se puede representar su valor: int, uint, long, ulong.
* Si el literal tiene el sufijo U o u, su tipo es el primero de los tipos en los cuales se puede representar su valor: uint, ulong.
* Si el literal tiene el sufijo L o l, su tipo es el primero de los tipos en los cuales se puede representar su valor: long, ulong.
* Si el literal tiene el sufijo UL, Ul, uL, ul, LU, Lu, lU o lu, es de tipo ulong.

Si el valor representado por un literal entero está fuera del intervalo de valores del tipo ulong, se produce un error en tiempo de compilación.

Por cuestiones de estilo, a la hora de escribir literales de tipo long se recomienda el uso de “L” en lugar de “l” para evitar la fácil confusión de la letra “l” con el dígito “1”.

Para que el número de valores int y long escritos como literales enteros decimales sea el mínimo posible, existen las dos reglas siguientes:

* Cuando aparece un literal entero decimal (decimal-integer-literal) con el valor 2147483648 (231) y sin sufijo de tipo entero (integer-type-suffix) como el símbolo (token) inmediatamente posterior a un símbolo de operador unario menos (§7.7.2), el resultado es una constante de tipo int con el valor −2147483648 (−231). En todas las demás situaciones, un literal entero decimal (decimal-integer-literal) de estas características es de tipo uint.
* Cuando aparece un literal entero decimal (decimal-integer-literal) con el valor 9223372036854775808 (263) y sin sufijo de tipo entero (integer-type-suffix) o con el sufijo de tipo entero L o l como token inmediatamente posterior de un token de operador unario menos (§7.7.2), el resultado es una constante de tipo long con el valor −9223372036854775808 (−263). En todas las demás situaciones, un literal entero decimal (decimal-integer-literal) de estas características es de tipo ulong.

#### Literales reales

Los literales reales permiten escribir valores de los tipos float, double y decimal.

real-literal:  
decimal-digits . decimal-digits exponent-partopt real-type-suffixopt  
. decimal-digits exponent-partopt real-type-suffixopt  
decimal-digits exponent-part real-type-suffixopt  
decimal-digits real-type-suffix

exponent-part:  
e signopt decimal-digits  
E signopt decimal-digits

sign: one of  
+ -

real-type-suffix: one of  
F f D d M m

Si no se especifica el sufijo de tipo real (real-type-suffix), el tipo del literal real es double. De lo contrario, el sufijo de tipo real determina el tipo del literal real, como sigue:

* Un literal real con el sufijo F o f es de tipo float. Por ejemplo, los literales 1f, 1.5f, 1e10f y 123.456F son todos de tipo float.
* Un literal real con el sufijo D o d es de tipo double. Por ejemplo, los literales 1d, 1.5d, 1e10d y 123.456D son todos de tipo double.
* Un literal real con el sufijo M o m es de tipo decimal. Por ejemplo, los literales 1m, 1.5m, 1e10m y 123.456M son todos de tipo decimal. Este literal se convierte en un valor decimal tomando el valor exacto y, si es necesario, redondeando al valor más cercano que se pueda representar mediante el redondeo de banca (§4.1.7). Se conserva cualquier escala que se detecte en el literal, salvo que el valor esté redondeado o sea cero (en este caso, el signo y la escala son 0). Por lo tanto, el análisis del literal 2.900m se interpreta para formar un decimal con signo 0, coeficiente 2900 y escala 3.

Si el literal especificado no puede representarse en el tipo indicado, se produce un error en tiempo de compilación.

El valor de un literal real de tipo float o double se determina mediante el uso del modo “redondeo al más cercano” de IEEE.

Tenga en cuenta que, en un literal real, siempre son necesarios dígitos decimales tras el punto decimal. Por ejemplo, 1.3F es un literal real pero 1.F no lo es.

#### Literales de carácter

Un literal de carácter representa un carácter único y normalmente está compuesto por un carácter entre comillas, como por ejemplo 'a'.

character-literal:  
' character '

character:  
single-character  
simple-escape-sequence  
hexadecimal-escape-sequence  
unicode-escape-sequence

single-character:  
 Any character except ' (U+0027), \ (U+005C), and new-line-character

simple-escape-sequence: one of  
\' \" \\ \0 \a \b \f \n \r \t \v

hexadecimal-escape-sequence:  
\x hex-digit hex-digitopt hex-digitopt hex-digitopt

Todo carácter que siga a un carácter de barra diagonal (\) en un carácter (character) debe ser uno de los siguientes: ', ", \, 0, a, b, f, n, r, t, u, U, x, v. En caso contrario, se producirá un error en tiempo de compilación.

Una secuencia de escape hexadecimal representa un solo carácter Unicode cuyo valor está formado por el número hexadecimal que siga a “\x”.

Si el valor representado por un literal de carácter es mayor que U+FFFF, se produce un error en tiempo de compilación.

Una secuencia de escape de caracteres Unicode (§2.4.1) en un literal de caracteres debe estar comprendido en el intervalo de U+0000 a U+FFFF.

Una secuencia de escape sencilla representa una codificación de caracteres Unicode, como se describe en la tabla inferior.

|  |  |  |
| --- | --- | --- |
| **Secuencia de escape** | **Nombre del carácter** | **Codificación Unicode** |
| \' | Comilla simple | 0x0027 |
| \" | Comilla doble | 0x0022 |
| \\ | Barra diagonal inversa | 0x005C |
| \0 | Nulo | 0x0000 |
| \a | Alerta | 0x0007 |
| \b | Retroceso | 0x0008 |
| \f | Avance de página | 0x000C |
| \n | Nueva línea | 0x000A |
| \r | Retorno de carro | 0x000D |
| \t | Tabulación horizontal | 0x0009 |
| \v | Tabulación vertical | 0x000B |

El tipo de un literal de caracteres (character-literal) es char.

#### Literales de cadena

C# admite dos formatos de literales de cadena: literales de cadena regulares y literales de cadena textuales.

Un literal de cadena regular consta de cero o más caracteres entre comillas dobles (como en "hello") y puede incluir secuencias de escape sencillas (como \t para el carácter de tabulación) y secuencias de escape hexadecimales y Unicode.

Un literal de cadena textual consta de un carácter @ seguido de un carácter de comillas dobles, cero o más caracteres y un carácter de comillas dobles de cierre. Un ejemplo sencillo puede ser @"hello". En un literal de cadena textual, los caracteres comprendidos entre los delimitadores se interpretan de manera literal, siendo la única excepción una secuencia de escape comillas (quote-escape-sequence). En concreto, las secuencias de escape sencillas, las secuencias de escape hexadecimales y Unicode no se procesan en los literales de cadena textuales. Un literal de cadena textual puede estar en varias líneas.

string-literal:  
regular-string-literal  
verbatim-string-literal

regular-string-literal:  
" regular-string-literal-charactersopt "

regular-string-literal-characters:  
regular-string-literal-character  
regular-string-literal-characters regular-string-literal-character

regular-string-literal-character:  
single-regular-string-literal-character  
simple-escape-sequence  
hexadecimal-escape-sequence  
unicode-escape-sequence

single-regular-string-literal-character:  
Any character except " (U+0022), \ (U+005C), and new-line-character

verbatim-string-literal:  
@" verbatim-string-literal-charactersopt "

verbatim-string-literal-characters:  
verbatim-string-literal-character  
verbatim-string-literal-characters verbatim-string-literal-character

verbatim-string-literal-character:  
single-verbatim-string-literal-character  
quote-escape-sequence

single-verbatim-string-literal-character:  
Any character except "

quote-escape-sequence:  
""

Todo carácter que siga a un carácter de barra diagonal (\) en un carácter literal de cadena regular (regular-string-literal-character) debe ser uno de los siguientes: ', ", \, 0, a, b, f, n, r, t, u, U, x, v. En caso contrario, se producirá un error en tiempo de compilación.

El ejemplo

string a = "hello, world"; // hello, world  
string b = @"hello, world"; // hello, world

string c = "hello \t world"; // hello world  
string d = @"hello \t world"; // hello \t world

string e = "Joe said \"Hello\" to me"; // Joe said "Hello" to me  
string f = @"Joe said ""Hello"" to me"; // Joe said "Hello" to me

string g = "\\\\server\\share\\file.txt"; // \\server\share\file.txt  
string h = @"\\server\share\file.txt"; // \\server\share\file.txt

string i = "one\r\ntwo\r\nthree";  
string j = @"one  
two  
three";

muestra varios literales de cadena. El último literal de cadena, j, es un literal de cadena textual que ocupa varias líneas. Los caracteres encerrados entre comillas, incluidos los espacios en blanco (por ejemplo, caracteres de nueva línea), se conservan literalmente.

Puesto que una secuencia de escape hexadecimal puede tener un número variable de dígitos hexadecimales, el literal de cadena "\x123" contiene un solo carácter con un valor hexadecimal de 123. Para crear una cadena que contenga el carácter con el valor hexadecimal 12 seguido del carácter 3, se podría escribir "\x00123" o "\x12" + "3" en su lugar.

El tipo de un literal de cadena (string-literal) es string.

Cada literal de cadena no genera necesariamente una nueva instancia de cadena. Cuando aparecen dos o más literales de cadena equivalentes según el operador de igualdad de cadenas (§7.10.7) en el mismo programa, estos literales se refieren a la misma instancia de cadena. Por ejemplo, el resultado del programa

class Test  
{  
 static void Main() {  
 object a = "hello";  
 object b = "hello";  
 System.Console.WriteLine(a == b);  
 }  
}

es True porque los dos literales hacen referencia a la misma instancia de cadena.

#### El literal null

null-literal:  
null

El literal null (null-literal) se puede convertir de manera implícita en un tipo de referencia o un tipo que acepta valores NULL.

### Operadores y signos de puntuación

Existen varias clases de operadores y signos de puntuación. Los operadores se utilizan en las expresiones para describir operaciones que implican uno o más operandos. Por ejemplo, la expresión a + b usa el operador + para sumar los dos operandos a y b. Los signos de puntuación permiten agrupar y separar.

operator-or-punctuator: one of  
{ } [ ] ( ) . , : ;  
+ - \* / % & | ^ ! ~  
= < > ? ?? :: ++ -- && ||  
-> == != <= >= += -= \*= /= %=  
&= |= ^= << <<= =>

right-shift:  
>|>

right-shift-assignment:  
>|>=

La barra vertical en las producciones de desplazamiento a la derecha (right-shift) y de asignación de desplazamiento a la derecha (right-shift-assignment), al contrario de lo que ocurre con otras producciones en la gramática sintáctica, no se puede insertar ningún tipo de caracteres (ni siquiera espacios en blanco) entre símbolos (token). Estas producciones se tratan de manera especial para permitir el correcto control de type-parameter-lists (§10.1.3).

## Directivas de preprocesamiento

Las directivas de preprocesamiento ofrecen la capacidad de omitir condicionalmente secciones de los archivos de código fuente, con el fin de notificar errores y advertencias, así como de delimitar regiones características del código fuente. El término “directivas de preprocesamiento” se utiliza por motivos de coherencia con los lenguajes de programación C y C++. En C#, no existe un paso de preprocesamiento individual; las directivas de preprocesamiento se procesan como parte de la fase de análisis léxico.

pp-directive:  
pp-declaration  
pp-conditional  
pp-line  
pp-diagnostic  
pp-region   
pp-pragma

A continuación se indican las directivas de preprocesamiento disponibles:

* #define y #undef, que permiten definir y anular, respectivamente, los símbolos de compilación condicional (§2.5.3).
* #if, #elif, #else y #endif, que se usan para omitir de forma condicional secciones de código fuente (§2.5.4).
* #line, que permite controlar números de línea de errores y advertencias (§2.5.7).
* #error y #warning, que permiten emitir errores y advertencias, respectivamente (§2.5.5).
* #region y #endregion, para marcar de forma explícita secciones del código fuente (§2.5.6).
* #pragma, que se usa para especificar información contextual opcional en el compilador (§2.5.8).

Una directiva de preprocesamiento siempre ocupa una línea independiente del código fuente y siempre empieza por un carácter # y un nombre de directiva de preprocesamiento. Puede haber un espacio en blanco antes del carácter # y entre este y el nombre de la directiva.

Una línea de código fuente que contiene una directiva #define, #undef, #if, #elif, #else, #endif, #line o #endregion puede finalizar con un comentario de una sola línea. Los comentarios delimitados (el estilo de comentarios /\* \*/) no están permitidos en las líneas de código fuente que contienen directivas de preprocesamiento.

Las directivas de preprocesamiento no son símbolos (tokens) y no forman parte de la gramática sintáctica de C#. No obstante, las directivas de preprocesamiento pueden utilizarse para incluir o excluir secuencias de tokens y, de esta forma, pueden afectar al significado de un programa de C#. Por ejemplo, una vez compilado, el programa:

#define A  
#undef B

class C  
{  
#if A  
 void F() {}  
#else  
 void G() {}  
#endif

#if B  
 void H() {}  
#else  
 void I() {}  
#endif  
}

produce como resultado exactamente la misma secuencia de tokens que el programa:

class C  
{  
 void F() {}  
 void I() {}  
}

Por lo tanto, aunque los dos programas sean muy diferentes léxicamente, sintácticamente son idénticos.

### Símbolos de compilación condicional

La compilación condicional que suministran funcionalmente las directivas #if, #elif, #else y #endif se controla mediante expresiones de preprocesamiento (§2.5.2) y símbolos de compilación condicional.

conditional-symbol:  
Any identifier-or-keyword except true or false

Un símbolo de compilación condicional tiene dos estados posibles: definido o no definido. Al principio del procesamiento léxico de un archivo de código fuente, un símbolo de compilación condicional tiene el estado no definido, a menos que haya sido definido de forma explícita por un mecanismo externo (como una opción del compilador de la línea de comandos). Cuando se procesa una directiva #define, el símbolo de compilación condicional nombrado en la directiva queda definido en el archivo de código fuente. El símbolo permanece definido hasta que se procesa una directiva #undef del mismo símbolo o hasta que se llega al final del archivo de código fuente. Una de las implicaciones es que las directivas #define y #undef de un archivo de código fuente no surten efecto en otros archivos de código fuente del mismo programa.

Cuando se hace referencia a un símbolo de compilación condicional definido en una expresión de preprocesamiento, este adquiere el valor booleano true, y un símbolo de compilación condicional no definido tiene el valor booleano false. No es necesario que los símbolos de compilación condicional se declaren explícitamente antes de que se haga referencia a ellos en expresiones de preprocesamiento. En lugar de ello, los símbolos no declarados simplemente no se definen y, por lo tanto, tienen el valor false.

El espacio de nombres de los símbolos de compilación condicional es único y exclusivo de todas las demás entidades con nombre de un programa de C#. Solo puede hacerse referencia a los símbolos de compilación condicional en directivas #define y #undef, y en expresiones de preprocesamiento.

### Expresiones de preprocesamiento

Las expresiones de preprocesamiento pueden encontrarse en las directivas #if y #elif. En las expresiones de preprocesamiento se permiten los operadores !, ==, !=, && y ||, además de los paréntesis, empleados para agrupar.

pp-expression:  
whitespaceopt pp-or-expression whitespaceopt

pp-or-expression:  
pp-and-expression  
pp-or-expression whitespaceopt || whitespaceopt pp-and-expression

pp-and-expression:  
pp-equality-expression  
pp-and-expression whitespaceopt && whitespaceopt pp-equality-expression

pp-equality-expression:  
pp-unary-expression  
pp-equality-expression whitespaceopt == whitespaceopt pp-unary-expression  
pp-equality-expression whitespaceopt != whitespaceopt pp-unary-expression

pp-unary-expression:  
pp-primary-expression  
! whitespaceopt pp-unary-expression

pp-primary-expression:  
true  
false  
conditional-symbol  
( whitespaceopt pp-expression whitespaceopt )

Cuando se hace referencia a un símbolo de compilación condicional definido en una expresión de preprocesamiento, este adquiere el valor booleano true, y un símbolo de compilación condicional no definido tiene el valor booleano false.

La evaluación de una expresión de preprocesamiento siempre produce un valor booleano. Las reglas de evaluación de una expresión de preprocesamiento son las mismas que las de una expresión constante (§7.19), excepto en que las únicas entidades definidas por el usuario a las que puede hacerse referencia son símbolos de compilación condicional.

### Directivas de declaración

Las directivas de declaración permiten definir o anular la definición de símbolos de compilación condicional.

pp-declaration:  
whitespaceopt # whitespaceopt define whitespace conditional-symbol pp-new-line  
whitespaceopt # whitespaceopt undef whitespace conditional-symbol pp-new-line

pp-new-line:  
whitespaceopt single-line-commentopt new-line

El procesamiento de una directiva #define causa la definición del símbolo de compilación condicional dado, empezando en la línea del código fuente que sigue a la directiva. De igual manera, el procesamiento de una directiva #undef causa la eliminación de la definición del símbolo de compilación condicional dado, empezando en la línea del código fuente que sigue a la directiva.

Todas las directivas #define y #undef de un archivo de código fuente deben aparecer antes del primer símbolo (token) (§2.4) en el archivo de código fuente; de lo contrario, se producirá un error en tiempo de compilación. En términos intuitivos, las directivas #define y #undef deben preceder a cualquier “código real” en el archivo de código fuente.

El ejemplo:

#define Enterprise

#if Professional || Enterprise  
 #define Advanced  
#endif

namespace Megacorp.Data  
{  
 #if Advanced  
 class PivotTable {...}  
 #endif  
}

es válido porque las directivas #define preceden al primer símbolo (token) (la palabra clave namespace) en el archivo de código fuente.

El siguiente ejemplo daría como resultado un error en tiempo de compilación porque hay una directiva #define después del código real:

#define A  
namespace N  
{  
 #define B  
 #if B  
 class Class1 {}  
 #endif  
}

Puede utilizarse #define para definir un símbolo de compilación condicional que ya esté definido, sin necesidad de que intervenga ningún símbolo #undef para ello. El ejemplo siguiente define un símbolo de compilación condicional A y después lo define otra vez.

#define A  
#define A

#undef puede “anular la definición” de un símbolo de compilación condicional que no esté definido. En el siguiente ejemplo se define un símbolo de compilación condicional A para después eliminar dicha definición dos veces; el segundo #undef, pese a no surtir ningún efecto, sigue siendo válido.

#define A  
#undef A  
#undef A

### Directivas de compilación condicional

Las directivas de compilación condicional permiten incluir o excluir de forma condicional partes de un archivo de código fuente.

pp-conditional:  
pp-if-section pp-elif-sectionsopt pp-else-sectionopt pp-endif

pp-if-section:  
whitespaceopt # whitespaceopt if whitespace pp-expression pp-new-line conditional-sectionopt

pp-elif-sections:  
pp-elif-section  
pp-elif-sections pp-elif-section

pp-elif-section:  
whitespaceopt # whitespaceopt elif whitespace pp-expression pp-new-line conditional-sectionopt

pp-else-section:  
whitespaceopt # whitespaceopt else pp-new-line conditional-sectionopt

pp-endif:  
whitespaceopt # whitespaceopt endif pp-new-line

conditional-section:  
input-section  
skipped-section

skipped-section:  
skipped-section-part  
skipped-section skipped-section-part

skipped-section-part:  
skipped-charactersopt new-line  
pp-directive

skipped-characters:  
whitespaceopt not-number-sign input-charactersopt

not-number-sign:  
Any input-character except #

Como indica la sintaxis, las directivas de compilación condicional deben escribirse como conjuntos formados por (en este orden): una directiva #if, cero o más directivas #elif, cero o más directivas #else y una directiva #endif. Entre las directivas se encuentran secciones condicionales de código fuente. Cada sección está controlada por la directiva inmediatamente precedente. Una sección condicional puede contener directivas de compilación condicional anidadas a condición de que dichas directivas formen conjuntos completos.

Una condicional pp (pp-conditional) selecciona como mucho una de las secciones condicionales (conditional-sections) contenidas para el procesamiento léxico normal:

* Las expresiones pp (pp-expression) de las directivas #if y #elif se evalúan por orden hasta que una produce un resultado true. Si una expresión produce true, se selecciona la sección condicional (conditional-section) de la directiva correspondiente.
* Si el resultado de todas las expresiones pp (pp-expression) es false, y si hay presente una directiva #else, se selecciona la sección condicional (conditional-section) de la directiva #else.
* En caso contrario, no se selecciona la sección condicional (conditional-section).

La sección condicional (conditional-section) seleccionada, si existe, se procesa como una sección entrada (input-section) normal: el código fuente de la sección debe cumplir la gramática léxica; se generan símbolos (tokens) a partir de dicho código y las directivas de preprocesamiento de la sección tienen los efectos prescritos.

Las secciones condicionales restantes (conditional-sections), si existen, se procesan como secciones omitidas (skipped-sections): excepto en lo que respeta a las directivas de preprocesamiento, el código fuente de la sección no tiene por qué cumplir la gramática léxica; no se generan tokens a partir del código fuente de la sección; y las directivas de preprocesamiento de la sección deben ser léxicamente correctas, pero en caso contrario no se procesan. Dentro de una sección condicional (conditional-section) que se ha procesado como sección omitida(skipped-section), cualquier sección condicional (conditional-section) anidada (contenida en construcciones #if...#endif y #region...#endregion anidadas) también se procesa como sección omitida (skipped-section).

En el ejemplo siguiente se ilustra cómo pueden anidarse directivas de compilación condicional:

#define Debug // Debugging on  
#undef Trace // Tracing off

class PurchaseTransaction  
{  
 void Commit() {  
 #if Debug  
 CheckConsistency();  
 #if Trace  
 WriteToLog(this.ToString());  
 #endif  
 #endif  
 CommitHelper();  
 }  
}

Excepto por las directivas de preprocesamiento, el código fuente omitido no se somete al análisis léxico. Por ejemplo, el código siguiente es válido, a pesar del comentario sin terminación de la sección #else:

#define Debug // Debugging on

class PurchaseTransaction  
{  
 void Commit() {  
 #if Debug  
 CheckConsistency();  
 #else  
 /\* Do something else  
 #endif  
 }  
}

No obstante, debe tenerse en cuenta que las directivas de preprocesamiento deben ser léxicamente correctas aunque se encuentren en secciones omitidas del código fuente.

Las directivas de preprocesamiento no se procesan cuando aparecen en elementos de entrada que ocupan varias líneas. Por ejemplo, el programa:

class Hello  
{  
 static void Main() {  
 System.Console.WriteLine(@"hello,   
#if Debug  
 world  
#else  
 Nebraska  
#endif  
 ");  
 }  
}

da como resultado:

hello,  
#if Debug  
 world  
#else  
 Nebraska  
#endif

En casos concretos, el conjunto de directivas de preprocesamiento que se procesa puede depender de la evaluación de la expresión pp (pp-expression). En el ejemplo:

#if X  
 /\*  
#else  
 /\* \*/ class Q { }  
#endif

se muestra siempre la misma secuencia de tokens (class Q { }), independientemente de que X esté definido o no. Si X está definido, las únicas directivas que se procesan son #if y #endif, a causa del comentario en varias líneas. Si X no está definido, las tres directivas (#if, #else, #endif) forman parte del conjunto de directivas.

### Directivas de diagnóstico

Las directivas de diagnóstico permiten generar de forma explícita mensajes de error y advertencias que se notifican de la misma manera que otros errores y advertencias en tiempo de compilación.

pp-diagnostic:  
whitespaceopt # whitespaceopt error pp-message  
whitespaceopt # whitespaceopt warning pp-message

pp-message:  
new-line  
whitespace input-charactersopt new-line

En el ejemplo:

#warning Code review needed before check-in

#if Debug && Retail  
 #error A build can't be both debug and retail  
#endif

class Test {...}

se muestra siempre la advertencia (“Code review needed before check-in”). Además, si están definidos los símbolos condicionales Debug y Retail, se muestra el error en tiempo de compilación “A build can't be both debug and retail”. Observe que en un mensaje pp (pp-message) puede especificarse cualquier texto: no es necesario incluir tokens formados correctamente, como indica el apóstrofo (comilla simple) de la palabra can’t.

### Directivas de región

Las directivas de región marcan de forma explícita regiones del código fuente.

pp-region:  
pp-start-region conditional-sectionopt pp-end-region

pp-start-region:  
whitespaceopt # whitespaceopt region pp-message

pp-end-region:  
whitespaceopt # whitespaceopt endregion pp-message

No se asocia ningún significado semántico a las regiones; los programadores y las herramientas automáticas las utilizan para marcar secciones del código fuente. El mensaje especificado en una directiva #region o #endregion tampoco tiene un significado semántico; sencillamente se usa para identificar la región. Las directivas #region y #endregion correspondientes pueden tener mensajes pp (pp-message) diferentes.

El procesamiento léxico de una región:

#region  
...  
#endregion

se corresponde exactamente con el procesamiento léxico de una directiva de compilación condicional de la forma:

#if true  
...  
#endif

### Directivas de línea

Las directivas de línea pueden usarse para modificar los números de línea y los nombres de archivo de código fuente que el compilador notifica en la salida como advertencias y errores, y que los atributos de información del llamador utilizan (§17.4.4).

Las directivas de línea suelen utilizarse en herramientas de metaprogramación que generan código fuente C# a partir de otras entradas de texto.

pp-line:  
whitespaceopt # whitespaceopt line whitespace line-indicator pp-new-line

line-indicator:  
decimal-digits whitespace file-name   
decimal-digits  
default  
hidden

file-name:  
" file-name-characters "

file-name-characters:  
file-name-character  
file-name-characters file-name-character

file-name-character:  
Any input-character except "

Cuando no aparecen directivas #line, el compilador muestra en su resultado los números de línea y nombres de archivo de código fuente reales. Cuando se procesa una directiva #line que incluye un indicador de línea (line-indicator) que no es default, el compilador trata la línea siguiente a la directiva como si tuviera el número de línea dado (y el nombre de archivo, si se especifica).

Una directiva #line default invierte el efecto de todas las directivas #line anteriores. El compilador comunica la información de línea real para las líneas posteriores, exactamente igual que si no se hubieran procesado directivas #line.

Una directiva #line hidden no produce ningún efecto sobre el archivo ni sobre los números de línea de los que se informa en los mensajes de error, pero sí afecta a la depuración en el nivel de código fuente. Durante la depuración, las líneas entre una directiva #line hidden y la siguiente directiva #line (distinta de #line hidden) carecerán de información de número de línea. Cuando se recorre el código en el depurador, estas líneas se omiten por completo.

Observe que los nombres de archivo (file-name) se diferencian de los literales de cadena ordinarios en que los caracteres de escape no se procesan; en los nombres de archivo, el carácter ‘\’ sólo designa un carácter convencional de barra diagonal inversa.

### Directivas pragma

La directiva de preprocesamiento #pragma se utiliza para especificar información contextual opcional en el compilador. La información suministrada en una directiva #pragma nunca cambiará la semántica del programa.

pp-pragma:  
whitespaceopt # whitespaceopt pragma whitespace pragma-body pp-new-line

pragma-body:  
pragma-warning-body

C# proporciona directivas #pragma para controlar advertencias del compilador. Es posible que en futuras versiones del lenguaje se incluyan directivas #pragma adicionales. Para garantizar la interoperabilidad con otros compiladores de C#, el compilador de Microsoft C# no genera errores de compilación para directivas #pragma conocidas; sin embargo, dichas directivas sí generan advertencias.

#### Pragma warning

La directiva #pragma warning se utiliza para deshabilitar o restaurar todos los mensajes de advertencia o un conjunto en particular durante la compilación del texto del programa subsiguiente.

pragma-warning-body:  
warning whitespace warning-action  
warning whitespace warning-action whitespace warning-list

warning-action:  
disable  
restore

warning-list:  
decimal-digits  
warning-list whitespaceopt , whitespaceopt decimal-digits

Una directiva #pragma warning que omite la lista de advertencias afecta a todas las advertencias. Una directiva #pragma warning que incluye una lista de advertencias afecta únicamente a aquellas advertencias especificadas en la lista.

Una directiva #pragma warning disable deshabilita todo o el conjunto de advertencias especificado.

Una directiva #pragma warning restore restaura todas las advertencias o un conjunto dado de las mismas al estado en el que se encontraban al inicio de la unidad de compilación. Tenga en cuenta que si una advertencia concreta se deshabilitó externamente, una directiva #pragma warning restore (para todas las advertencias o para una específica) no rehabilitará dicha advertencia.

En el siguiente ejemplo se muestra el uso de #pragma warning para deshabilitar temporalmente la advertencia notificada al hacer referencia a miembros obsoletos, con el número de advertencia del compilador de Microsoft C#.

using System;

class Program  
{  
 [Obsolete]  
 static void Foo() {}

static void Main() {  
#pragma warning disable 612  
 Foo();  
#pragma warning restore 612  
 }  
}

# Conceptos básicos

## Inicio de la aplicación

Una aplicación es un ensamblado que contiene un punto de entrada. Cuando se ejecuta una aplicación, se crea un nuevo dominio de aplicación. Pueden existir varias creaciones de instancias diferentes de una aplicación simultáneamente en el mismo equipo, cada una con su propio dominio de aplicación.

Un dominio de aplicación habilita el aislamiento de aplicaciones, al actuar como contenedor del estado de una aplicación. Un dominio de aplicación se comporta como contenedor y límite de los tipos definidos en la aplicación y en las bibliotecas de clases que utiliza. Los tipos cargados en un dominio de aplicación son distintos de los mismos tipos cargados en otro, y las instancias de los objetos no se comparten directamente entre dominios de aplicación. Por ejemplo, cada dominio de aplicación mantiene su propia copia de variables estáticas para estos tipos, y un constructor estático de un tipo se ejecuta como máximo una vez por dominio de aplicación. Las implementaciones tienen libertad para proporcionar mecanismos o directivas específicas de la implementación para la creación y destrucción de los dominios de aplicación.

Un inicio de aplicación se produce cuando el entorno de ejecución llama a un método designado, que se conoce como punto de entrada de la aplicación. Este método de punto de entrada siempre se denomina Main y puede tener una de las signaturas siguientes:

static void Main() {...}

static void Main(string[] args) {...}

static int Main() {...}

static int Main(string[] args) {...}

Como se muestra, el punto de entrada de manera opcional puede devolver un valor int. El valor devuelto se utiliza en la finalización de la aplicación (§3.2).

El punto de entrada puede tener, opcionalmente, un parámetro formal. El parámetro puede tener cualquier nombre, pero el tipo del parámetro debe ser string[]. Si el parámetro formal está presente, el entorno de ejecución crea y pasa un argumento string[] que contiene los argumentos de línea de comandos especificados al iniciar la aplicación. El argumento string[] nunca es null, pero puede tener longitud cero si no se encuentra especificado ningún argumento de línea de comandos.

Dado que C# acepta la sobrecarga de métodos, una clase o un struct puede contener varias definiciones del mismo método, a condición de que todas tengan una firma diferente. No obstante, ninguna clase o struct de un mismo programa podrá contener más de un método denominado Main cuya definición lo califique para ser utilizado como punto de entrada de aplicación. Sin embargo, sí se permiten otras versiones sobrecargadas de Main, a condición de que tengan más de un parámetro o de que su único parámetro no sea de tipo string[].

Una aplicación puede estar compuesta por varias clases o structs. Es posible para más de una de esas clases o structs contener un método denominado Main cuya definición lo califica para ser utilizado como punto de entrada de la aplicación. En tales casos, debe utilizarse un mecanismo externo (por ejemplo, una opción de compilador de línea de comandos) para seleccionar uno de estos métodos Main como punto de entrada.

En C#, todos los métodos deben estar definidos como miembros de una clase o un struct. Normalmente, la accesibilidad declarada (§3.5.1) de un método está determinada por los modificadores de acceso (§10.3.5) especificados en su declaración y, de forma similar, la accesibilidad declarada de un tipo está determinada por los modificadores de acceso especificados en su declaración. Para que un método dado de un tipo concreto sea invocable, tanto el tipo como el miembro deben ser accesibles. No obstante, el punto de entrada de la aplicación es un caso especial. En concreto, el entorno de ejecución puede tener acceso al punto de entrada de la aplicación con independencia de su accesibilidad declarada y de la accesibilidad declarada de sus declaraciones de tipo envolvente.

Es posible que el método de punto de entrada de la aplicación no se encuentre dentro de una declaración de clase genérica.

A todos los demás respectos, los métodos de punto de entrada se comportan como los de punto de no entrada.

## Finalización de la aplicación

La finalización de la aplicación devuelve el control al entorno de ejecución.

Si el tipo del valor devuelto por el método de punto de entrada de la aplicación es int, el valor devuelto cumple la función de código de estado de finalización de la misma. El propósito de este código es permitir la comunicación del éxito o el error al entorno de ejecución.

Si el tipo del valor devuelto del método de punto de entrada es void, cuando se alcanza la llave de cierre (}), que finaliza el método, o cuando se ejecuta una instrucción return que no contiene una expresión, se produce un código de estado de finalización igual a 0.

Antes de la finalización de una aplicación, se llama a destructores para todos los objetos que aún no hayan sido sometidos a la recolección de elementos no utilizados (salvo que se haya suprimido este tipo de limpieza mediante, por ejemplo, una llamada al método de biblioteca GC.SuppressFinalize).

## Declaraciones

Las declaraciones de un programa de C# definen los elementos constituyentes del programa. Los programas de C# se organizan mediante el uso de espacios de nombres (§9), que pueden contener declaraciones de tipo y declaraciones de espacio de nombres anidadas. Las declaraciones de tipo (§9.6) se utilizan para definir clases (§10), structs (§10.14), interfaces (§13), enumeraciones (§14) y delegados (§15). Las categorías de los miembros permitidos en una declaración de tipo dependen de la forma de la declaración. Por ejemplo, las declaraciones de clase pueden contener declaraciones para las constantes (§10.4), campos (§10.5), métodos (§10.6), propiedades (§10.7), eventos (§10.8), indizadores (§10.9), operadores (§10.10), constructores de instancia (§10.11), constructores estáticos (§10.12), destructores (§10.13) y tipos anidados (§10.3.8).

Una declaración define un nombre en el espacio de declaración (declaration space) al que pertenece la declaración. Con la excepción de los miembros sobrecargados (§3.6), es un error en tiempo de compilación contar con dos o más declaraciones que introducen miembros con el mismo nombre en un espacio de declaración. No está permitido que un espacio de declaración contenga categorías de miembros diferentes con el mismo nombre. Por ejemplo, un espacio de declaración no puede contener un campo y un método con el mismo nombre.

Existen varios tipos diferentes de espacios de declaración, como se explica a continuación.

* En todos los archivos de código fuente de un programa, las declaraciones de miembro de espacio de nombres (namespace-member-declaration) sin una declaración de espacio de nombres (namespace-declaration) envolvente son miembros de un solo espacio de declaración combinado, denominado espacio de declaración global.
* Dentro de todos los archivos de código fuente de un programa, las declaraciones de miembros de espacios de nombres (namespace-member-declaration) de las declaraciones de espacio de nombres (namespace-declaration) que tienen el mismo nombre completo del espacio de nombres son miembros de un solo espacio de declaración combinado.
* Cada declaración de clase, struct o interfaz crea un nuevo espacio de declaración. Los nombres se introducen en este espacio de declaración a través de declaraciones de miembro de clase (class-member-declaration), declaraciones de miembro de struct (struct-member-declaration), declaraciones de miembro de interfaz (interface-member-declaration) o parámetros de tipo (type-parameter). Salvo por las declaraciones de constructor de instancia y las declaraciones de constructor estático sobrecargadas, una clase o struct no puede contener una declaración de miembro que tenga el mismo nombre que la clase o struct. Una clase, struct o interfaz permiten la declaración de métodos e indizadores sobrecargados. Asimismo, las clases y structs admiten la declaración de operadores y constructores de instancia sobrecargados. Por ejemplo, una clase, un struct o una interfaz pueden contener varias declaraciones de método con el mismo nombre, a condición de que dichas declaraciones difieran en su signatura (§3.6). Debe tenerse en cuenta que las clases base no contribuyen al espacio de declaración de una clase, y que las interfaces base no contribuyen al espacio de declaración de una interfaz. Por lo tanto, una clase o interfaz derivada tiene permitido declarar un miembro con el mismo nombre que un miembro heredado. Se dice que un miembro como este oculta el miembro heredado.
* Una declaración de delegado crea un espacio de declaración nuevo. Los nombres se introducen en este espacio de declaración a través de parámetros formales (parámetros de tipo fijado (fixed-parameter) y matrices de parámetros (parameter-array)) y parámetros de tipo (type-parameter).
* Una declaración de enumeración crea un espacio de declaración nuevo. Los nombres se introducen en este espacio de declaración mediante declaraciones de miembros de enumeración (enum-member-declarations).
* Cada declaración de método, declaración de indizador, declaración de operador, declaración de constructor de instancia y función anónima crea un nuevo espacio de declaración denominado espacio de declaración de variables locales. Los nombres se introducen en este espacio de declaración a través de parámetros formales (parámetros fijos (fixed-parameter) y matrices de parámetros (parameter-array)) y parámetros de tipo (type-parameter). El cuerpo del miembro de función o de la función anónima, si existe, se considera anidado en el espacio de declaración de variables locales. Que el espacio de declaración de una variable local y el espacio de declaración de una variable local anidado contengan elementos con el mismo nombre supone un error. Por lo tanto, dentro de un espacio de declaración anidado no es posible declarar una variable o constante local con el mismo nombre que una variable o constante local contenida en un espacio de declaración. Es posible que dos espacios de declaración contengan elementos con el mismo nombre siempre y cuando ninguno de los espacios de declaración contenga al otro.
* Cada bloque (block \b ) o bloque switch (switch-block \b \t "See declaration space, block and" \b \t "See declaration space, block and" \b ), así como cada instrucción *for*, *foreach* y *using*, crea un espacio de declaración de variables locales para las variables locales y las constantes locales \b \t "See declaration space, block and" \b \t "See declaration space, switch block" . Los nombres se introducen en este espacio de declaración a través de declaraciones de variables locales (local-variable-declaration) y declaraciones de constantes locales ( local-constant-declaration). Tenga en cuenta que los bloques que sean el cuerpo de un miembro de función o una función anónima, o que se encuentren dentro de él, están anidados en el espacio de declaración de variables locales declarado por esas funciones para sus parámetros. Que, por ejemplo, un método con una variable local y un parámetro tengan el mismo nombre supone un error.
* Cada bloque (block) o bloque switch (switch-block) crea un espacio de declaración independiente para las etiquetas. Los nombres se introducen en este espacio de declaración a través de instrucciones con etiqueta (labeled-statement) y se hace referencia a ellos a través de instrucciones goto (goto-statement). El espacio de declaración de etiquetas de un bloque incluye los bloques anidados. Por lo tanto, dentro de un bloque anidado no es posible declarar una etiqueta del mismo nombre que una etiqueta contenida en un bloque contenedor.

El orden textual de declaración de los nombres generalmente carece de importancia. Así, el orden textual no resulta significativo para la declaración o el uso de espacios de nombres, constantes, métodos, propiedades, eventos, indizadores, operadores, constructores de instancia, destructores, constructores estáticos o tipos. El orden de declaración es significativo de las siguientes formas:

* El orden de declaración de las declaraciones de campos y de variables locales determina el orden en que se ejecutan sus inicializadores (si existen).
* Las variables locales no se pueden usar antes de su definición (§3.7).
* El orden de declaración para las declaraciones de miembros de enumeración (§14.3) es importante cuando se omiten los valores de expresión constante (constant-expression).

El espacio de declaración de un espacio de nombres es “de extremo abierto”, es decir, dos declaraciones de espacio de nombres con un mismo nombre completo contribuirán al mismo espacio de declaración. Por ejemplo:

namespace Megacorp.Data  
{  
 class Customer  
 {  
 ...  
 }  
}

namespace Megacorp.Data  
{  
 class Order  
 {  
 ...  
 }  
}

Las dos declaraciones de espacios de nombres anteriores contribuyen al mismo espacio de declaración, en este caso declarando dos clases con los nombres completos Megacorp.Data.Customer y Megacorp.Data.Order. Como las dos declaraciones contribuyen al mismo espacio de declaración, si cada una contiene una declaración de clase con el mismo nombre se producirá un error en tiempo de compilación.

Como se especificó anteriormente, el espacio de declaración de un bloque incluye cualquier bloque anidado. De este modo, en el siguiente ejemplo, los métodos F y G producen errores en tiempo de compilación porque el nombre i está declarado en el bloque exterior y no se puede volver a declarar en el bloque interior. Sin embargo, los métodos H e I son válidos porque los dos caracteres i se declaran en bloques independientes no anidados.

class A  
{  
 void F() {  
 int i = 0;  
 if (true) {  
 int i = 1;   
 }  
 }

void G() {  
 if (true) {  
 int i = 0;  
 }  
 int i = 1;   
 }

void H() {  
 if (true) {  
 int i = 0;  
 }  
 if (true) {  
 int i = 1;  
 }  
 }

void I() {  
 for (int i = 0; i < 10; i++)  
 H();  
 for (int i = 0; i < 10; i++)  
 H();  
 }  
}

## Miembros

Los espacios de nombres y los tipos tienen miembros. Los miembros de una entidad están generalmente disponibles mediante el uso de un nombre completo compuesto por una referencia a la entidad seguida de un token “.” y el nombre del miembro.

Los miembros de un tipo se declaran en la declaración de tipo o se heredan de la clase base del tipo. Si un tipo se deriva de una clase base, todos los miembros de dicha clase, excepto los constructores de instancia, destructores y constructores estáticos, pasan a ser miembros del tipo derivado. La accesibilidad declarada de un miembro de clase base no controla si el miembro se hereda: la herencia se amplía a cualquier miembro que no sea un constructor de instancia, un constructor estático o un destructor. No obstante, un miembro heredado puede no estar accesible en un tipo derivado, ya sea a causa de su accesibilidad declarada (§3.5.1) o porque está oculto por una declaración del propio tipo (§3.7.1.2).

### Miembros de espacio de nombres

Los espacios de nombres y los tipos que no tienen un espacio de nombres envolvente son miembros del espacio de nombres global. Esto corresponde directamente a los nombres declarados en el espacio de declaración global.

Los espacios de nombres y los tipos declarados en de un espacio de nombres son miembros de dicho espacio. Esto corresponde directamente a los nombres declarados en el espacio de declaración del espacio de nombres.

Los espacios de nombres no presentan restricciones de acceso. No es posible declarar espacios de nombres privados, protegidos o internos, y los nombres de los espacios de nombres siempre son accesibles públicamente.

### Miembros de struct

Los miembros de un struct son los miembros declarados en el struct y los miembros heredados de la clase base directa del struct System.ValueType y la clase base indirecta object.

Los miembros de un tipo simple se corresponden directamente con los del tipo struct con alias por el tipo simple:

* Los miembros de sbyte son los del struct System.SByte.
* Los miembros de byte son los del struct System.Byte.
* Los miembros de short son los del struct System.Int16.
* Los miembros de ushort son los del struct System.UInt16.
* Los miembros de int son los del struct System.Int32.
* Los miembros de uint son los del struct System.UInt32.
* Los miembros de long son los del struct System.Int64.
* Los miembros de ulong son los del struct System.UInt64.
* Los miembros de char son los del struct System.Char.
* Los miembros de float son los del struct System.Single.
* Los miembros de double son los del struct System.Double.
* Los miembros de decimal son los del struct System.Decimal.
* Los miembros de bool son los del struct System.Boolean.

### Miembros de enumeraciones

Los miembros de una enumeración son las constantes declaradas en la enumeración y los miembros heredados de la clase base directa de la enumeración System.Enum y las clases base indirectas System.ValueType y object.

### Miembros de clase

Los miembros de una clase son aquellos que se declaran en la clase y los que se heredan de la clase base (a excepción de la clase object, que no tiene clase base). Entre los miembros heredados de la clase base se incluyen las constantes, los campos, los métodos, las propiedades, los eventos, los indizadores, los operadores y los tipos de la clase base, pero no los constructores de instancia, los destructores ni los constructores estáticos de la clase base. Los miembros de la base clase se heredan con independencia de su accesibilidad.

La declaración de una clase puede contener declaraciones de constantes, campos, métodos, propiedades, eventos, indizadores, operadores, constructores de instancia, destructores, constructores estáticos y tipos.

Los miembros de object y string se corresponden directamente con los miembros de los tipos de clases de los que son alias:

* Los miembros de object son los de la clase System.Object.
* Los miembros de string son los de la clase System.String.

### Miembros de interfaz

Los miembros de una interfaz son los miembros declarados en la interfaz y en todas las interfaces base de la misma. Los miembros de la clase object no son, estrictamente hablando, miembros de ninguna interfaz (§13.2). Sin embargo, los miembros de la clase object están disponibles a través de la búsqueda de miembros en cualquier tipo de interfaz (§7.4).

### Miembros de matriz

Los miembros de una matriz son los miembros heredados de la clase System.Array.

### Miembros de delegados

Los miembros de un delegado son los miembros heredados de la clase System.Delegate.

## Acceso a miembros

Las declaraciones a miembros aportan control sobre el acceso a los miembros. La accesibilidad de un miembro se establece mediante la accesibilidad declarada (§3.5.1) del miembro combinada con la accesibilidad del tipo contenedor inmediato, si existe.

Cuando se permite el acceso a un miembro concreto, se dice que este es accesible. Por otro lado, cuando no se permite el acceso a un miembro concreto, se dice que este es inaccesible. El acceso a un miembro está permitido cuando la ubicación textual en la que tiene lugar el acceso está incluida en el dominio de accesibilidad (§3.5.2) del miembro.

### Accesibilidad declarada

La accesibilidad declarada de un miembro puede ser cualquiera de las siguientes:

* Public, que se selecciona mediante la inclusión de un modificador public en la declaración del miembro. El significado intuitivo de public es “acceso sin restricciones”.
* Protected, que se selecciona mediante la inclusión de un modificador protected en la declaración del miembro. El significado intuitivo de protected es “acceso restringido para la clase contenedora o los tipos derivados de ella”.
* Internal, que se selecciona mediante la inclusión de un modificador internal en la declaración del miembro. El significado intuitivo de internal es “acceso restringido al programa”.
* Protected internal (es decir, protegida o interna), que se selecciona mediante la inclusión de los modificadores protected e internal en la declaración del miembro. El significado intuitivo de protected internal es “acceso restringido al programa actual o a los tipos derivados de la clase contenedora”.
* Private, que se selecciona mediante la inclusión de un modificador private en la declaración del miembro. El significado intuitivo de private es “acceso restringido al tipo contenedor”.

Según el contexto en el que se produce la declaración de un miembro, sólo se permite declarar ciertos tipos de accesibilidad. Además, si una declaración del miembro no incluye modificadores de acceso, el contexto en que tiene lugar la declaración determina la accesibilidad declarada predeterminada.

* Los espacios de nombres tienen implícitamente la accesibilidad declarada public. Los modificadores de acceso no se pueden utilizar en las declaraciones de espacios de nombres.
* Los tipos declarados en las unidades de compilación o los espacios de nombres pueden tener la accesibilidad declarada public o internal y su valor predeterminado puede ser internal.
* Los miembros de clase pueden tener cualquiera de las cinco clases de accesibilidad declarada y el valor predeterminado de la accesibilidad declarada private. (Debe tenerse en cuenta que un tipo declarado como miembro de una clase sólo puede tener una de las cinco clases de accesibilidad declarada, mientras que un tipo declarado como miembro de un espacio de nombres solo puede tener la accesibilidad declarada public o internal).
* Los miembros de struct pueden tener la accesibilidad declarada public, internal o private, siendo el valor predeterminado private, puesto que los structs son implícitamente de tipo sealed. Los miembros de struct introducidos en un struct (es decir, no heredados por ellos) no pueden tener una accesibilidad declarada protected ni protected internal. (Tenga en cuenta que un tipo declarado como miembro de un struct puede tener accesibilidad declarada public, internal o private, mientras que un tipo declarado como miembro de un espacio de nombres solo puede tener la accesibilidad declarada public o internal).
* Los miembros de interfaz tienen implícitamente la accesibilidad declarada public. Los modificadores de acceso no se pueden utilizar en las declaraciones de miembros de interfaz.
* Los miembros de enumeración tienen implícitamente la accesibilidad declarada public. Los modificadores de acceso no pueden utilizarse en las declaraciones de miembros de enumeración.

### Dominios de accesibilidad

El dominio de accesibilidad de un miembro está formado por las secciones (posiblemente separadas) de texto del programa en las que está permitido el acceso al miembro. Para definir el dominio de accesibilidad de un miembro, se dice que un miembro es de nivel superior si no está declarado dentro de un tipo, y se dice que está anidado si se declara dentro de otro tipo. Asimismo, el texto de programa de un programa se define como todo el texto de programa contenido en todos los archivos de código fuente del programa. El texto de programa de un tipo se define como todo el texto del programa contenido en las declaraciones de tipo (type-declaration) de ese tipo (incluidos, posiblemente, los tipos anidados dentro del tipo).

El dominio de accesibilidad de un tipo predefinido (como object, int o double) es ilimitado.

El dominio de accesibilidad de un tipo sin enlazar de nivel superior T (§4.4.3) declarado en un programa P se define como sigue:

* Si la accesibilidad declarada de T es public, el dominio de accesibilidad de T es el texto del programa de P y cualquier programa que haga referencia a P.
* Si la accesibilidad declarada de T es internal, el dominio de accesibilidad de T es el texto de programa de P.

A partir de estas definiciones, se deduce que el dominio de accesibilidad de un tipo sin enlazar de nivel superior es, al menos, el texto del programa en el que se declara ese tipo.

El dominio de accesibilidad para un tipo construido T<A1, ...,AN> es la intersección entre el dominio de accesibilidad del tipo genérico sin enlazar T y los dominios de accesibilidad de los argumentos de tipo A1, ...,AN.

El dominio de accesibilidad de un miembro anidado M declarado en un tipo T dentro de un programa P se define como sigue (teniendo en cuenta que M probablemente sea un tipo):

* Si la accesibilidad declarada de M es public, el dominio de accesibilidad de M es el dominio de accesibilidad de T.
* Si la accesibilidad declarada de M es protected internal, D será la unión del texto de programa de P y del texto de programa de cualquier tipo derivado de T, que está declarado fuera de P. El dominio de accesibilidad de M es la intersección del dominio de accesibilidad de T con D.
* Si la accesibilidad declarada de M es protected, D será la unión del texto de programa de T y del texto de programa de cualquier tipo derivado de T. El dominio de accesibilidad de M es la intersección del dominio de accesibilidad de T con D.
* Si la accesibilidad declarada de M es internal, el dominio de accesibilidad de M es la intersección del dominio de accesibilidad de T con el texto de programa de P.
* Si la accesibilidad declarada de M es private, el dominio de accesibilidad de M es el texto de programa de T.

De estas definiciones se deduce que el dominio de accesibilidad de un tipo anidado es siempre, al menos, el texto de programa del tipo donde el miembro aparece declarado. Asimismo, se concluye que el dominio de accesibilidad de un miembro nunca es más inclusivo que el dominio de accesibilidad del tipo en el que se declara el miembro.

En términos intuitivos, cuando se obtiene acceso a un tipo o un miembro M, se evalúan los pasos siguientes para garantizar que el acceso esté permitido:

* En primer lugar, si M se declara dentro de un tipo (por oposición a una unidad de compilación o un espacio de nombres), se produce un error en tiempo de compilación si el tipo no está accesible.
* Por lo tanto, si M es public, el acceso está permitido.
* O bien, si M es protected internal, el acceso está permitido si se produce dentro del programa en el que está declarado M o si se produce dentro de una clase derivada de la clase en la que está declarado M y tiene lugar por medio del tipo de la clase derivada (§3.5.3).
* O bien, si M es protected, el acceso está permitido si se produce dentro de la clase en la que está declarado M o si se produce dentro de una clase derivada de la clase en la que está declarado M y tiene lugar por medio del tipo de la clase derivada (§3.5.3).
* O bien, si M es internal, el acceso está permitido si tiene lugar dentro del programa en el que está declarado M.
* O bien, si M es private, el acceso está permitido si tiene lugar dentro del tipo en el que está declarado M.
* De lo contrario, el tipo o el miembro es inaccesible y se produce un error en tiempo de compilación.

En el ejemplo

public class A  
{  
 public static int X;  
 internal static int Y;  
 private static int Z;  
}

internal class B  
{  
 public static int X;  
 internal static int Y;  
 private static int Z;

public class C  
 {  
 public static int X;  
 internal static int Y;  
 private static int Z;  
 }

private class D  
 {  
 public static int X;  
 internal static int Y;  
 private static int Z;  
 }  
}

las clases y los miembros tienen los siguientes dominios de accesibilidad:

* El dominio de accesibilidad de A y A.X es ilimitado.
* El dominio de accesibilidad de A.Y, B, B.X, B.Y, B.C, B.C.X y B.C.Y es el texto del programa contenedor.
* El dominio de accesibilidad de A.Z es el texto del programa de A.
* El dominio de accesibilidad de B.Z y B.D es el texto de programa de B, incluido el texto de programa de B.C y de B.D.
* El dominio de accesibilidad de B.C.Z es el texto del programa de B.C.
* El dominio de accesibilidad de B.D.X y B.D.Y es el texto de programa de B, incluido el texto de programa de B.C y de B.D.
* El dominio de accesibilidad de B.D.Z es el texto del programa de B.D.

Como muestra el ejemplo, el dominio de accesibilidad de un miembro nunca es mayor que el de un tipo contenedor. Por ejemplo, aunque todos los miembros X tengan una accesibilidad declarada como public, todos excepto A.X tienen dominios de accesibilidad que están restringidos por un tipo contenedor.

Como se explica en §3.4, todos los miembros de una clase base, a excepción de los constructores de instancia, destructores y constructores estáticos, son heredados por tipos derivados. Esto incluye incluso a los miembros privados de una clase base. No obstante, el dominio de accesibilidad de un miembro privado sólo incluye el texto del programa del tipo donde el miembro aparece declarado. En el ejemplo

class A  
{  
 int x;

static void F(B b) {  
 b.x = 1; // Ok  
 }  
}

class B: A  
{  
 static void F(B b) {  
 b.x = 1; // Error, x not accessible  
 }  
}

la clase B hereda el miembro privado x de la clase A. Dado que el miembro es privado, solo está accesible dentro del cuerpo de clase (class-body) de A. Por lo tanto, el acceso a b.x es correcto en el método A.F, pero no en el método B.F.

### Acceso protegido para miembros de instancia

Cuando se obtiene acceso a un miembro de instancia protected fuera del texto del programa de la clase en la que está declarado, y cuando se obtiene acceso a un miembro de instancia protected internal fuera del texto del programa en el que está declarado, el acceso debe tener lugar dentro de una declaración de clase derivada de la clase en la que está declarada. Es más, es necesario que el acceso se produzca a través de una instancia de ese tipo de clase derivada o un tipo de clase construido a partir de la misma. Esta restricción impide que una clase derivada tenga acceso a miembros protegidos de otras clases derivadas, incluso cuando los miembros son heredados de la misma clase base.

Supongamos que B es una clase base que declara un miembro de instancia M protegido, y que D es una clase que se deriva de B. Dentro del cuerpo de clase (class-body) de D, el acceso a M puede tener uno de los siguientes formatos:

* Un nombre de tipo (type-name) o expresión primaria (primary-expression) sin calificar de la forma M.
* Una expresión primaria (primary-expression) de la forma E.M, siempre que el tipo de E sea T o una clase derivada de T, donde T es el tipo de clase D, o un tipo de clase construido a partir de D.
* Una expresión primaria (primary-expression) de la forma base.M.

Además de estas formas de acceso, una clase derivada puede tener acceso a un constructor de instancia protegido de una clase base en un inicializador de constructor (constructor-initializer) (§10.11.1).

En el ejemplo

public class A  
{  
 protected int x;

static void F(A a, B b) {  
 a.x = 1; // Ok  
 b.x = 1; // Ok  
 }  
}

public class B: A  
{  
 static void F(A a, B b) {  
 a.x = 1; // Error, must access through instance of B  
 b.x = 1; // Ok  
 }  
}

dentro de A, es posible obtener acceso a x mediante instancias de A y de B, puesto que en cualquiera de los dos casos el acceso tiene lugar mediante una instancia de A o una clase derivada de A. No obstante, dentro de B, no es posible el acceso a x por medio de una instancia de A, puesto que A no se deriva de B.

En el ejemplo

class C<T>  
{  
 protected T x;  
}

class D<T>: C<T>  
{  
 static void F() {  
 D<T> dt = new D<T>();  
 D<int> di = new D<int>();  
 D<string> ds = new D<string>();  
 dt.x = default(T);  
 di.x = 123;  
 ds.x = "test";  
 }  
}

se permiten las tres asignaciones a x porque todas se dan gracias a instancias de tipos de clase construidos desde el tipo genérico.

### Restricciones de accesibilidad

Algunas construcciones del lenguaje C# requieren que un tipo sea al menos tan accesible como un miembro u otro tipo. Se dice que un tipo T es por lo menos tan accesible como un miembro o un tipo M si el dominio de accesibilidad de T es un supraconjunto del dominio de accesibilidad de M. Es decir, T es por lo menos tan accesible como M si T es accesible en todos los contextos donde M es accesible.

Existen las siguientes restricciones de la accesibilidad:

* La clase base directa de un tipo de clase debe ser al menos tan accesible como el propio tipo de clase.
* Las interfaces base explícitas de un tipo de interfaz deben ser al menos tan accesibles como el propio tipo de interfaz.
* El tipo de valor devuelto y los tipos de los parámetros de un tipo delegado deben ser al menos tan accesibles como el propio tipo delegado.
* El tipo de una constante debe ser al menos tan accesible como la propia constante.
* El tipo de un campo debe ser al menos tan accesible como el propio campo.
* El tipo de valor devuelto y los tipos de parámetros de un método deben ser al menos tan accesibles como el propio método.
* El tipo de una propiedad debe ser al menos tan accesible como la misma propiedad.
* El tipo de un evento debe ser al menos tan accesible como el propio evento.
* El tipo y los tipos de parámetros de un indizador deben ser al menos tan accesibles como el propio indizador.
* El tipo de valor devuelto y los tipos de parámetros de un operador deben ser al menos tan accesibles como el propio operador.
* Los tipos de parámetros de un constructor de instancia deben ser al menos tan accesibles como el propio constructor de instancia.

En el ejemplo

class A {...}

public class B: A {...}

la clase B produce un error en tiempo de compilación porque A no tiene una capacidad de acceso mínima como B.

Análogamente, en el ejemplo:

class A {...}

public class B  
{  
 A F() {...}

internal A G() {...}

public A H() {...}  
}

el método H de B produce un error en tiempo de compilación porque el tipo de valor devuelto A no es por lo menos tan accesible como el método.

## Firmas y sobrecargas

Los métodos, constructores de instancia, indizadores y operadores están caracterizados por sus signaturas:

* La signatura de un método se compone del nombre del método, el número de parámetros de tipo y el tipo y el modo de pasar los parámetros (valor, referencia o resultado) de cada uno de sus parámetros formales, considerados de izquierda a derecha. Con este fin, todo parámetro de tipo del método que se produce en el tipo de un parámetro formal se identifica no por su nombre, sino por su posición ordinal en la lista de argumentos de tipo del método. La signatura de un método no incluye específicamente el tipo de valor devuelto, el modificador params que puede especificarse para el parámetro situado más a la derecha, ni la restricción de parámetro de tipo opcional.
* La signatura de un constructor de instancias se compone del tipo y el modo de pasar los parámetros (valor, referencia o resultado) de cada uno de sus parámetros formales, considerados de izquierda a derecha. La signatura de un constructor de instancias no incluye específicamente el modificador params, que puede especificarse para el parámetro situado más a la derecha.
* La firma de un indizador está formada por el tipo de sus parámetros formales, considerados de izquierda a derecha. La signatura de un indizador no incluye específicamente el tipo de elemento ni el modificador params que puede estar especificado para el parámetro situado más a la derecha.
* La firma de un operador se compone del nombre del operador y del tipo de sus parámetros formales, considerados de izquierda a derecha. La firma de un operador no incluye específicamente el tipo del resultado.

Dos signaturas del mismo tipo de miembro se consideran **la misma signatura** si tienen el mismo nombre, número de parámetros de tipo y modos de pasar los parámetros, y además existe una conversión de identidad entre sus tipos correspondientes (§6.1.1).

Las signaturas constituyen el mecanismo de habilitación que permite sobrecargar los miembros de clases, structs e interfaces:

* La sobrecarga de los métodos permite que una clase, struct o interfaz declare varios métodos con el mismo nombre, siempre que sus firmas sean únicas dentro de esa clase, struct o interfaz.
* La sobrecarga de los constructores de instancia permite que una clase o un struct declare varios constructores de instancia, a condición de que sus firmas sean únicas dentro de esa clase o struct.
* La sobrecarga de los indizadores permite que una clase, struct o interfaz declare varios indizadores, siempre que sus firmas sean únicas dentro de esa clase, struct o interfaz.
* La sobrecarga de los operadores permite que una clase o un struct declare varios operadores con el mismo nombre, siempre que sus firmas sean únicas dentro de esa clase o struct.

Aunque los modificadores de parámetros out y ref se consideran como parte de una signatura, los miembros declarados en un tipo único no pueden diferir en la signatura únicamente por ref y out. Si dos miembros se declaran en el mismo tipo con signaturas que serían iguales si todos los parámetros de ambos métodos con modificadores out se cambiaran a modificadores ref, se produce un error de compilación. Para otros propósitos de coincidencia de signatura (por ejemplo, para ocultar o invalidar), ref y out se consideran como parte de la signatura y no coinciden entre sí. (Esta restricción se utiliza para permitir traducir con facilidad los programas de C# para que se ejecuten en Common Language Infrastructure (CLI), que no proporciona una manera de definir métodos que solamente se diferencian en ref y out).

El siguiente ejemplo muestra un conjunto de declaraciones de métodos sobrecargados con sus firmas.

interface ITest  
{  
 void F(); // F()

void F(int x); // F(int)

void F(ref int x); // F(ref int)

void F(out int x); // F(out int) error

void F(int x, int y); // F(int, int)

int F(string s); // F(string)

int F(int x); // F(int) error

void F(string[] a); // F(string[])

void F(params string[] a); // F(string[]) error  
}

Se debe tener en cuenta que los modificadores de parámetros ref y out (§10.6.1) forman parte de una signatura. Por lo tanto, F(int) y F(ref int) son signaturas únicas. Sin embargo, F(ref int) y F(out int) no se pueden declarar dentro de la misma interfaz porque sus signaturas se diferencian únicamente en ref y out. Asimismo, el tipo de valor devuelto y el modificador params no forman parte de una signatura, por lo que no es posible sobrecargar basándose exclusivamente en el tipo de valor devuelto o en la inclusión o exclusión del modificador params. Como tales, las declaraciones de los métodos F(int) y F(params string[]) anteriormente identificadas producen un error en tiempo de compilación.

## Ámbitos

El ámbito (scope) de un nombre es la región del texto del programa en la cual es posible referirse a la entidad declarada por el nombre sin la calificación de este. Los ámbitos se pueden anidar y un ámbito interno puede volver a declarar el significado de un nombre de un ámbito exterior (sin embargo, de esta forma no se quita la restricción impuesta por §3.3 de que, dentro de un bloque anidado, no se puede declarar una variable local con el mismo nombre de una variable local de un bloque contenedor). En este caso, se dice que el nombre del ámbito externo está oculto en la región del texto del programa cubierta por el ámbito interno, y el acceso al nombre externo sólo es posible si se califica el nombre.

* El ámbito de un miembro de espacio de nombres declarado por una declaración de miembro de espacio de nombres (namespace-member-declaration) (§9.5) sin declaración envolvente de espacio de nombres (namespace-declaration) es todo el texto del programa.
* El ámbito de un miembro de espacio de nombres declarado mediante una declaración de miembro de espacio de nombres (namespace-member-declaration), dentro de una declaración de espacio de nombres (namespace-declaration) cuyo nombre completo sea N, será el cuerpo de espacio de nombres (namespace-body) de todas las declaraciones de espacio de nombres (namespace-declaration) cuyo nombre completo sea N o empiece por N, seguido por un punto.
* El ámbito de un nombre definido por una directiva de alias extern (extern-alias-directive) se extiende sobre las directivas using (using-directives), los atributos globales (global-attributes) y las declaraciones de miembros de espacios de nombres (namespace-member-declarations) de la unidad de compilación o del cuerpo de espacio de nombres inmediato. Una directiva de alias extern (extern-alias-directive) no contribuye con ningún miembro nuevo al espacio de declaración subyacente. En otras palabras, una directiva de alias extern (extern-alias-directive) no es transitiva, sino que solo afecta a la unidad de compilación o al cuerpo de espacio de nombres en el que se da.
* El ámbito de un nombre definido o importado por una directiva using (using-directive) (§9.4) se amplía a las declaraciones de miembros de espacios de nombres (namespace-member-declarations) de la unidad de compilación (compilation-unit) o del cuerpo de espacio de nombres (namespace-body) en que tiene lugar la directiva using (using-directive). Una directiva using (using-directive) puede ofrecer cero o más espacios de nombres o nombres de tipos disponibles dentro de una unidad de compilación (compilation-unit) o cuerpo de espacio de nombres (namespace-body) en particular, pero no aporta ningún miembro nuevo al espacio de declaración subyacente. Es decir, una directiva using (using-directive) no es transitiva, sino que afecta únicamente a la unidad de compilación (compilation-unit) o al cuerpo de espacio de nombres (namespace-body) en que tiene lugar.
* El ámbito de un parámetro de tipo declarado por una lista de parámetros de tipo (type-parameter-list) en una declaración de clase (class-declaration) (§10.1) es la clase base (class-base), las cláusulas de restricciones de parámetros de tipo (type-parameter-constraints-clauses) y el cuerpo de clase (class-body) de esa declaración de clase ( class-declaration).
* El ámbito de un parámetro de tipo declarado por una lista de parámetros de tipo (type-parameter-list) en una declaración de estructura (struc-declaration) (§11.1) son las interfaces de estructuras (struct-interfaces), las cláusulas de restricciones de parámetros de tipo (type-parameter-constraints-clauses) y el cuerpo de estructura (struct-body) de esa declaración de estructura ( struct-declaration).
* El ámbito de un parámetro de tipo declarado por una lista de parámetros de tipo (type-parameter-list) en una declaración de interfaz (interface-declaration) (§13.1) es la base de interfaz (interface-base), las cláusulas de restricciones de parámetros de tipo (type-parameter-constraints-clauses) y el cuerpo de interfaz (interface-body) de esa declaración de interfaz ( interface-declaration).
* El ámbito de un parámetro de tipo declarado por una lista de parámetros de tipo (type-parameter-list) en una declaración de delegado (delegate-declaration) (§15.1) es el tipo devuelto (return-type), la lista de parámetros formales (formal-parameter-list) y las cláusulas de restricciones de parámetros de tipo (type-parameter-constraints-clauses) de esa declaración de delegado (delegate-declaration).
* El ámbito de un miembro declarado por una declaración de miembro de clase (class-member-declaration) (§10.1.6) es el cuerpo de clase (class-body) en que se produce la declaración. Asimismo, el ámbito de un miembro de clase se amplía al cuerpo de clase (class-body) de las clases derivadas que estén incluidas en el dominio de accesibilidad (§3.5.2) del miembro.
* El ámbito de un miembro declarado por una declaración de miembro de struct (struct-member-declaration) (§11.2) es el cuerpo de struct (struct-body) en el que se produce la declaración.
* El ámbito de un miembro declarado por una declaración de miembro de enumeración (enum-member-declaration) (§14.3) es el cuerpo de enumeración (enum-body) en el que se produce la declaración.
* El ámbito de un parámetro declarado en una declaración de método (method-declaration) (§10.6) es el cuerpo del método (method-body) de dicha declaración de método (method-declaration).
* El ámbito de un parámetro declarado en una declaración de indizador (indexer-declaration) (§10.9) son las declaraciones de descriptor de acceso (accessor-declarations) de dicha declaración de indizador (indexer-declaration).
* El ámbito de un parámetro declarado en una declaración de operador (operator-declaration) (§10.10) es el bloque (block) de dicha declaración de operador (operator-declaration).
* El ámbito de un parámetro declarado en una declaración de constructor (constructor-declaration) (§10.11) es el inicializador de constructor (constructor-initializer) y el bloque (block) de dicha declaración de constructor (constructor-declaration).
* El ámbito de un parámetro declarado en una expresión de lambda (lambda-expression) (§7.15) es el cuerpo de expresión de lambda (lambda-expression-body) de dicha expresión de lambda (lambda-expression).
* El ámbito de un parámetro declarado en una expresión de método anónimo (anonymous-method-expression) (§7.15) es el bloque (block) de dicha expresión de método anónimo (anonymous-method-expression).
* El ámbito de una etiqueta declarada en una instrucción con etiqueta (labeled-statement) (§8.4) es el bloque (block) en el que se produce la declaración.
* El ámbito de una variable local declarada en una declaración de variable local (local-variable-declaration) (§8.5.1) es el bloque donde se produce la declaración.
* El ámbito de una variable local declarada en un bloque switch (switch-block) de una instrucción switch (§8.7.2) es el bloque switch (switch-block).
* El ámbito de una variable local declarada en un inicializador for (for-initializer) de una instrucción for (§8.8.3) es el inicializador for (for-initializer), la condición for (for-condition), el iterador for (for-iterator) y la instrucción (statement) contenida de la instrucción for.
* El ámbito de una constante local declarada en una declaración de constante local (local-constant-declaration) (§8.5.2) es el bloque donde se produce la declaración. Es un error en tiempo de compilación hacer referencia a una constante local en una posición textual que precede a su declarador de constante (constant-declarator).
* El ámbito de una variable declarada como parte de una instrucción foreach (foreach-statement), instrucción using (using-statement), instrucción lock (lock-statement) o expresión de consulta (query-expression) está determinado por la expansión de la construcción determinada.

Dentro del ámbito de un miembro de espacio de nombres, clase, struct o enumeración es posible hacer referencia al miembro en una posición textual que precede a la declaración del miembro. Por ejemplo:

class A  
{  
 void F() {  
 i = 1;  
 }

int i = 0;  
}

Por lo tanto, es válido que F haga referencia a i antes de su declaración.

En el ámbito de una variable local, es un error en tiempo de compilación hacer referencia a la variable local en una posición textual que preceda al declarador de la variable local (local-variable-declarator). Por ejemplo:

class A  
{  
 int i = 0;

void F() {  
 i = 1; // Error, use precedes declaration  
 int i;  
 i = 2;  
 }

void G() {  
 int j = (j = 1); // Valid  
 }

void H() {  
 int a = 1, b = ++a; // Valid  
 }  
}

En el método F anterior, la primera asignación de i no hace ninguna referencia concreta al campo declarado en el ámbito externo. En lugar de ello, hace referencia a la variable local, con lo que se produce un error durante la compilación, porque precede textualmente a la declaración de la variable. En el método G, el uso de j en el inicializador de la declaración de j es válido porque no precede al declarador de variable local (local-variable-declarator). En el método H, un declarador de variable local (local-variable-declarator) posterior hace referencia correctamente a una variable local declarada en un declarador de variable local (local-variable-declarator) anterior con la misma declaración de variable local (local-variable-declaration).

Las reglas que rigen el ámbito de las variables locales pretenden garantizar que el significado de un nombre utilizado en el contexto de una expresión no varíe dentro de un bloque. Si el ámbito de una variable local sólo se extiende desde su declaración hasta el final del bloque, entonces, en el ejemplo anterior, la primera asignación asignará a la variable de instancia y la segunda a la variable local, lo que posiblemente producirá errores en tiempo de compilación si más adelante fuera necesario reorganizar las instrucciones del bloque.

El significado de un nombre contenido en un bloque puede diferir según el contexto en que se utilice el nombre. En el ejemplo

using System;

class A {}

class Test  
{  
 static void Main() {  
 string A = "hello, world";  
 string s = A; // expression context

Type t = typeof(A); // type context

Console.WriteLine(s); // writes "hello, world"  
 Console.WriteLine(t); // writes "A"  
 }  
}

el nombre A se utiliza en el contexto de una expresión para hacer referencia a la variable local A y en un contexto de tipo para hacer referencia a la clase A.

### Ocultar nombres

El ámbito de una entidad habitualmente abarca más texto del programa que el espacio de declaración de la entidad. En concreto, el ámbito de una entidad puede incluir declaraciones que introducen nuevos espacios de declaración que contienen entidades con el mismo nombre. Las declaraciones de este tipo hacen que la entidad original quede oculta (hidden). A la inversa, se dice que una entidad es visible cuando no está oculta.

La ocultación de nombres se produce cuando los ámbitos se superponen a causa del anidamiento y cuando se superponen por herencia. Las características de los dos tipos de ocultación se explican en las próximas secciones.

#### Ocultar mediante anidación

La ocultación de nombres por medio del anidamiento puede ser el resultado del anidamiento de espacios de nombres o tipos contenidos en espacios de nombres, la consecuencia del anidamiento de tipos dentro de clases o structs y el resultado de las declaraciones de parámetros y variables locales.

En el ejemplo

class A  
{  
 int i = 0;

void F() {  
 int i = 1;  
 }

void G() {  
 i = 1;  
 }  
}

dentro del método F, la variable de instancia i queda oculta por la variable local i, pero dentro del método G, i se sigue refiriendo a la variable de instancia.

Cuando un nombre en un ámbito interno oculte otro nombre en el ámbito externo, ocultará también todas sus apariciones sobrecargadas. En el ejemplo

class Outer  
{  
 static void F(int i) {}

static void F(string s) {}

class Inner  
 {  
 void G() {  
 F(1); // Invokes Outer.Inner.F  
 F("Hello"); // Error  
 }

static void F(long l) {}  
 }  
}

la llamada F(1) invoca la F declarada en Inner porque la declaración interna oculta todas las apariciones externas de F. Por el mismo motivo, la llamada F("Hello") produce un error en tiempo de compilación.

#### Ocultar mediante herencia

La ocultación de nombres por medio de la herencia ocurre cuando clases o structs vuelven a declarar nombres que se han heredado de clases base. Este tipo de ocultación de nombres toma una de las siguientes formas:

* Una constante, campo, propiedad, evento o tipo introducido en una clase o struct que oculta todos los miembros de la clase base con el mismo nombre.
* Un método introducido en una clase o struct que oculta todos los miembros de clase base no de método con el mismo nombre, y todos los métodos de clase base con la misma signatura (§3.6).
* Un indizador introducido en una clase o struct que oculta todos los indizadores de la clase base con la misma signatura (§3.6).

Las reglas que gobiernan las declaraciones de operador (§10.10) imposibilitan la declaración por una clase derivada de un operador con la misma signatura que un operador de una clase base. Por lo tanto, los operadores nunca se ocultan mutuamente.

Al contrario que cuando se oculta un nombre de un ámbito externo, cuando se oculta un nombre accesible desde un ámbito heredado se genera una advertencia. En el ejemplo

class Base  
{  
 public void F() {}  
}

class Derived: Base  
{  
 public void F() {} // Warning, hiding an inherited name  
}

la declaración de F en Derived genera una advertencia. Ocultar un nombre heredado no es específicamente un error, puesto que esto impediría la evolución independiente de las clases base. Por ejemplo, la situación anterior podría haberse producido porque una versión posterior de Base introdujo un método F que no estaba presente en una versión anterior de la clase. Si la situación anterior hubiera sido un error, cualquier cambio realizado a una clase base en una versión independiente de la biblioteca de clases podría haber invalidado las clases derivadas.

La advertencia causada por la ocultación de un nombre heredado puede eliminarse mediante uso del modificador new:

class Base  
{  
 public void F() {}  
}

class Derived: Base  
{  
 new public void F() {}  
}

El modificador new indica que F de Derived es “nuevo” y que su propósito real es ocultar el miembro heredado.

Una declaración de un miembro nuevo oculta un miembro heredado sólo dentro del ámbito del nuevo miembro.

class Base  
{  
 public static void F() {}  
}

class Derived: Base  
{  
 new private static void F() {} // Hides Base.F in Derived only  
}

class MoreDerived: Derived  
{  
 static void G() { F(); } // Invokes Base.F  
}

En el ejemplo anterior, la declaración de F en Derived oculta el método F que se heredó de Base, pero, como el nuevo F de Derived tiene acceso de tipo privado, su ámbito no se extiende a MoreDerived. De este modo, la llamada F() de MoreDerived.G es válida e invocará a Base.F.

## Espacios de nombres y nombres de tipos

Algunos contextos de un programa de C# requieren que se especifique un nombre de espacio de nombres (namespace-name) o un nombre de tipo (type-name).

namespace-name:  
namespace-or-type-name

type-name:  
namespace-or-type-name

namespace-or-type-name:  
identifier type-argument-listopt  
namespace-or-type-name . identifier type-argument-listoptqualified-alias-member

Un nombre de espacio de nombres (namespace-name) es un nombre de espacio de nombres o de tipo (namespace-or-type-name) que hace referencia a un espacio de nombres. Conforme a la resolución explicada más adelante, el nombre de espacio de nombres o de tipo (namespace-or-type-name) de un nombre de espacio de nombres (namespace-name) debe hacer referencia a un espacio de nombres o, de lo contrario, se producirá un error en tiempo de compilación. No puede haber argumentos de tipo (§4.4.1) en un nombre de espacio de nombres (namespace-name); solo los tipos pueden tener argumentos de tipo.

Un nombre de tipo (type-name) es un nombre de espacio de nombres o de tipo (namespace-or-type-name) que hace referencia a un tipo. Conforme a la resolución explicada más adelante, el nombre de espacio de nombres o de tipo (namespace-or-type-name) de un nombre de tipo (type-name) debe hacer referencia a un tipo o, de lo contrario, se producirá un error en tiempo de compilación.

Si el nombre de espacio de nombres o de tipo (namespace-or-type-name) es un miembro de alias calificado, su significado es como se describe en §9.7. En caso contrario, un nombre de espacio de nombres o de tipo (namespace-or-type-name) adopta una de las cuatro estructuras siguientes:

* I
* I<A1, ..., AK>
* N.I
* N.I<A1, ..., AK>

donde I es un solo identificador, N es un nombre de espacio de nombres o de tipo (namespace-or-type-name) y <A1, ..., AK> es una lista de argumentos de tipo (type-argument-list). Si no se especifica ninguna lista de argumentos de tipo (type-argument-list), se considera que K es cero.

El significado de un nombre de espacio de nombres o de tipo (namespace-or-type-name) se determina como sigue:

* Si el nombre de espacio de nombres o de tipo (namespace-or-type-name) tiene la estructura I o la forma I<A1, ..., AK>:
* Si K es cero y el nombre de espacio de nombres o de tipo (namespace-or-type-name) aparece dentro de una declaración de método genérico (§10.6) y si dicha declaración incluye un parámetro de tipo (§10.1.3) con el nombre I, el nombre de espacio de nombres o de tipo (namespace-or-type-name) hace referencia a dicho parámetro de tipo.
* De lo contrario, si el nombre de espacio de nombres o de tipo (namespace-or-type-name) aparece dentro de una declaración de tipo, para cada tipo de instancia T (§10.3.1), empezando por el tipo de instancia de dicha declaración tipo y continuando con el tipo de instancia para cada clase envolvente o declaración de structs (si las hubiera):
* Si K es cero y la declaración de T incluye un parámetro de tipo con el nombre I, el nombre de espacio de nombres o de tipo (namespace-or-type-name) hace referencia a dicho parámetro de tipo.
* De lo contrario, si el nombre de espacio de nombres o de tipo (namespace-or-type-name) aparece dentro del cuerpo de a declaración de tipo y T o alguno de sus tipos base contiene un tipo accesible anidado con el nombre I y parámetros de tipo K, el nombre de espacio de nombres o de tipo (namespace-or-type-name) hace referencia a dicho tipo construido con los argumentos de tipo dados. Si hay más de uno de esos tipos, se selecciona el tipo declarado dentro del tipo más derivado. Tenga en cuenta que los miembros que no son de tipo (constantes, campos, métodos, propiedades, indizadores, operadores, constructores de instancia, destructores y constructores estáticos) y miembros de tipo con un número diferente de parámetros de tipo se omiten a la hora de determinar el significado de un nombre de espacio de nombres o de tipo (namespace-or-type-name).
* Si los pasos anteriores no produjeron resultados satisfactorios, para cada espacio de nombres N, empezando por el espacio de nombres en el que se produce el nombre de espacio de nombres o de tipo (namespace-or-type-name), continuando por cada uno de los espacios de nombres envolventes (si los hubiera) y terminando por el espacio de nombres global, se irán evaluando los siguientes pasos hasta que se localice una entidad:
* Si K es cero e I es el nombre de un espacio de nombres en N:
* Si la ubicación donde tiene lugar el nombre de espacio de nombres o de tipo (namespace-or-type-name) tiene una declaración de espacio de nombres para N y la declaración de espacio de nombres contiene una directiva de alias extern (extern-alias-directive) o una directiva de alias using (using-alias-directive) que asocia el nombre I con un nombre de espacio de nombres o de tipo, el nombre de espacio de nombres o de tipo (namespace-or-type-name) es ambiguo y se genera un error en tiempo de compilación.
* De lo contrario, el nombre de espacio de nombres o de tipo (namespace-or-type-name) hace referencia al espacio de nombres denominado I en N.
* De lo contrario, si N contiene un tipo accesible con un nombre I y parámetros de tipo K:
* Si K es cero y la ubicación donde tiene lugar el nombre de espacio de nombres o de tipo (namespace-or-type-name) tiene una declaración de espacio de nombres para N y la declaración de espacio de nombres contiene una directiva de alias extern (extern-alias-directive) o una directiva de alias using (using-alias-directive) que asocia el nombre I con un nombre de espacio de nombres o de tipo, el nombre de espacio de nombres o de tipo (namespace-or-type-name) es ambiguo y se genera un error en tiempo de compilación.
* De lo contrario, el nombre de espacio de nombres o de tipo (namespace-or-type-name) hace referencia al tipo construido con los argumentos de tipo dados.
* De lo contrario, si la ubicación donde tiene lugar el nombre de espacio de nombres o de tipo (namespace-or-type-name) tiene una declaración de espacio de nombres para N:
* Si K es cero y la declaración de espacio de nombres contiene una directiva de alias extern (extern-alias-directive) o una directiva de alias using (using-alias-directive) que asocia el nombre I a un espacio de nombres o tipo importado, entonces el nombre de espacio de nombres o de tipo (namespace-or-type-name) hace referencia a dicho espacio de nombres o tipo.
* De lo contrario, si los espacios de nombres importados por las directivas using de espacio de nombres (using-namespace-directives) de la declaración del espacio de nombres contienen exactamente un único tipo con el nombre I y parámetros de tipo K, entonces el nombre de espacio de nombres o de tipo (namespace-or-type-name) hace referencia al tipo construido con los argumentos de tipo dados.
* De lo contrario, si los espacios de nombres importados por las directivas using de espacio de nombres (using-namespace-directives) de la declaración del espacio de nombres contienen más de un tipo con el nombre I y parámetros de tipo K, entonces el nombre de espacio de nombres o de tipo (namespace-or-type-name) será ambiguo y se producirá un error.
* De lo contrario, el nombre de espacio de nombres o de tipo (namespace-or-type-name) no está definido y se produce un error en tiempo de compilación.
* De lo contrario, el nombre de espacio de nombres o de tipo (namespace-or-type-name) tiene la estructura N.I o la forma N.I<A1, ..., AK>. Se resuelve primero N como nombre de espacio de nombres o de tipo (namespace-or-type-name). Si la resolución de N no es correcta, se produce un error en tiempo de compilación. De lo contrario, N.I o N.I<A1, ..., AK> se resuelve de la siguiente manera:
* Si K es cero y N hace referencia a un espacio de nombres y N contiene un espacio de nombres anidado con el nombre I, el nombre de espacio de nombres o de tipo (namespace-or-type-name) hace referencia a dicho espacio de nombres anidado.
* De lo contrario, si N contiene un espacio de nombres y N contiene un tipo accesible con el nombre I y parámetros de tipo K, el nombre de espacio de nombres o de tipo (namespace-or-type-name) hace referencia a dicho tipo construido con los argumentos de tipo dados.
* De lo contrario, si N hace referencia a una clase (posiblemente construida) o a un tipo struct y N o algunas de sus clases base contiene un tipo accesible anidado con el nombre I y parámetros de tipo K, el nombre de espacio de nombres o de tipo (namespace-or-type-name) hace referencia a dicho tipo construido con los argumentos de tipo dados. Si hay más de uno de esos tipos, se selecciona el tipo declarado dentro del tipo más derivado. Tenga en cuenta que si el significado de N.I se determina como parte de la resolución de la especificación de la clase base de N, la clase base directa de N se considera object (§10.1.4.1).
* De lo contrario, N.I es un nombre de espacio de nombres o de tipo no válido (invalid namespace-or-type-name) y se produce un error en tiempo de compilación.

Se permite que un nombre de espacio de nombres o de tipo (namespace-or-type-name) (§10.1.1.3) haga referencia a una clase estática solo si

* El nombre de espacio de nombres o de tipo (namespace-or-type-name) es T en un nombre de espacio de nombres o de tipo (namespace-or-type-name) con el formato T.I, o bien
* El nombre de espacio de nombres o de tipo (namespace-or-type-name) es T en una expresión typeof (typeof-expression) (§7.5.11) con el formato typeof(T).

### Nombres completos

Todos los espacios de nombres y todos los tipos tienen un nombre completo, que los identifica de forma exclusiva. El nombre completo de un espacio de nombres o un tipo N se determina como sigue:

* Si N es un miembro del espacio de nombres global, su nombre completo es N.
* De lo contrario, su nombre completo es S.N, donde S es el nombre completo del espacio de nombres o el tipo donde N está declarado.

Esto es, el nombre completo de N es la ruta de acceso jerárquica completa de los identificadores que conducen a N, empezando desde el espacio de nombres global. Dado que todos los miembros de un espacio de nombres o un tipo deben tener un nombre único, se deduce que el nombre completo de un espacio de nombres o tipo siempre es único.

En el ejemplo siguiente se muestran varias declaraciones de tipos y espacio de nombres con sus nombres completos asociados.

class A {} // A

namespace X // X  
{  
 class B // X.B  
 {  
 class C {} // X.B.C  
 }

namespace Y // X.Y  
 {  
 class D {} // X.Y.D  
 }  
}

namespace X.Y // X.Y  
{  
 class E {} // X.Y.E  
}

## Administración automática de la memoria

C# usa la administración automática de memoria, que exime a los programadores de la asignación manual y la liberación de la memoria ocupada por objetos. Las directivas de administración automática de la memoria se implementan mediante un recolector de elementos no utilizados. El ciclo de vida de la administración de la memoria de un objeto es:

1. Cuando se crea el objeto, se le asigna memoria, se ejecuta el constructor y el objeto se considera activo.
2. En caso de que no se pueda obtener acceso al objeto o alguna de sus partes por medio de las posibles continuaciones de la ejecución, en lugar de iniciar los destructores, el objeto dejará de considerarse en uso y quedará expuesto al proceso de destrucción. El compilador de C# y el recolector de elementos no utilizados pueden optar por analizar el código para determinar qué referencias a un objeto podrán utilizarse en el futuro. Por ejemplo, si una variable local que se encuentra dentro del ámbito es la única referencia existente a un objeto, pero no se hace ninguna referencia a tal variable local en ninguna continuación posible de la ejecución desde el punto actual de ejecución dentro del procedimiento, el recolector de elementos no utilizados puede tratar los objetos (aunque no necesariamente) como si ya no estuvieran en uso.
3. Cuando el objeto ya es candidato a la destrucción, el destructor del objeto (si existe) se ejecuta en un momento posterior no especificado (§10.13). En circunstancias normales el destructor del objeto solo se ejecuta una vez, aunque puede que las API específicas de la implementación permitan la invalidación de este comportamiento.
4. Si, una vez ejecutado el destructor de un objeto, las posibles continuaciones de la ejecución (incluida la ejecución de los destructores) no pudieran obtener acceso al objeto o a alguna de sus partes, se considerará que ya no se encuentra accesible y quedará expuesto al proceso de recolección.
5. Por último, cuando el objeto ya es candidato para la recolección, en un momento posterior, el recolector de elementos no utilizados libera la memoria asociada al objeto.

El recolector de elementos no utilizados mantiene información acerca de la utilización de objetos y la utiliza para tomar decisiones sobre la administración de memoria. Por ejemplo, contiene datos como en qué parte de la memoria se ubica un objeto recién creado, cuándo se reubica un objeto y cuándo un objeto ya no se utiliza o está inaccesible.

Igual que otros lenguajes que dan por supuesta la existencia de un recolector de elementos no utilizados, C# se ha diseñado para que el recolector de elementos no utilizados pueda implementar una amplia gama de directivas de administración de la memoria. Por ejemplo, C# no exige que se lleve a cabo la ejecución de destructores ni la recolección de objetos en el momento en que pasen a ser candidatos a ella, ni que los destructores se ejecuten en un orden concreto o en un subproceso concreto.

El comportamiento del recolector de elementos no utilizados puede controlarse, en cierta medida, mediante métodos estáticos de la clase System.GC, una clase que puede utilizarse para solicitar la ejecución de la recolección o de los destructores (o su no ejecución), etc.

Dado que el recolector de elementos no utilizados tiene una amplia libertad para decidir cuándo debe recolectar objetos y ejecutar destructores, una implementación compatible puede producir resultados diferentes de los mostrados en el código siguiente. El programa

using System;

class A  
{  
 ~A() {  
 Console.WriteLine("Destruct instance of A");  
 }  
}

class B  
{  
 object Ref;

public B(object o) {  
 Ref = o;  
 }

~B() {  
 Console.WriteLine("Destruct instance of B");  
 }  
}

class Test  
{  
 static void Main() {  
 B b = new B(new A());  
 b = null;  
 GC.Collect();  
 GC.WaitForPendingFinalizers();  
 }  
}

crea una instancia de la clase A y otra de la clase B. Estos objetos se convierten en candidatos a la recolección de elementos no utilizados cuando se asigna a la variable b el valor null, puesto que, después de ello, no se puede obtener acceso a ellos mediante ningún código escrito por el usuario. El resultado podría ser

Destruct instance of A  
Destruct instance of B

o

Destruct instance of B  
Destruct instance of A

puesto que el lenguaje no impone restricciones al orden en que se lleva a cabo la recolección de elementos de los objetos no utilizados.

En casos concretos, la distinción entre “candidato a la destrucción” y “candidato a la recolección” puede ser importante. Por ejemplo,

using System;

class A  
{  
 ~A() {  
 Console.WriteLine("Destruct instance of A");  
 }

public void F() {  
 Console.WriteLine("A.F");  
 Test.RefA = this;  
 }  
}

class B  
{  
 public A Ref;

~B() {  
 Console.WriteLine("Destruct instance of B");  
 Ref.F();  
 }  
}

class Test  
{  
 public static A RefA;  
 public static B RefB;

static void Main() {  
 RefB = new B();  
 RefA = new A();  
 RefB.Ref = RefA;  
 RefB = null;  
 RefA = null;

// A and B now eligible for destruction  
 GC.Collect();  
 GC.WaitForPendingFinalizers();

// B now eligible for collection, but A is not  
 if (RefA != null)  
 Console.WriteLine("RefA is not null");  
 }  
}

En el programa anterior, si el recolector de elementos no utilizados decide ejecutar el destructor de A antes que el de B, el resultado de este programa puede ser:

Destruct instance of A  
Destruct instance of B  
A.F  
RefA is not null

Debe tenerse en cuenta que, aunque la instancia de A no estaba en uso cuando se ejecutó el destructor de A, todavía es posible llamar a los métodos de A (en este caso, F) desde otro destructor. Asimismo, no debe olvidarse que la ejecución de un destructor puede hacer que un objeto vuelva a ser utilizable desde el programa principal. En este caso, la ejecución del destructor de B hace que una instancia de A que no estaba anteriormente en uso esté ahora accesible desde la referencia activa Test.RefA. Tras la llamada a WaitForPendingFinalizers, la instancia de B pasa a quedar expuesta a la recolección, pero no así la instancia de A, debido a la referencia Test.RefA.

Para evitar confusiones y comportamientos inesperados, generalmente es recomendable que los destructores solamente realicen la limpieza de los datos almacenados en los campos propios de sus objetos, y que no realicen acciones en los objetos a los que se hace referencia o en campos estáticos.

Una alternativa al uso de destructores consiste en permitir que una clase implemente la interfaz System.IDisposable. Así se permite al cliente del objeto determinar cuándo liberar los recursos del objeto, normalmente mediante el acceso al objeto como recurso en una instrucción using (§8.13).

## Orden de ejecución

La ejecución de los programas de C# se realiza de tal modo que los efectos secundarios de cada subproceso en ejecución se van conservando en puntos críticos de ejecución. Un efecto secundario (side effect) se define como la lectura o escritura de un campo volátil, la escritura en una variable no volátil, la escritura en un recurso externo o el inicio de una excepción. Los puntos de ejecución críticos en los que deben conservarse tales efectos secundarios son las referencias a campos volátiles (§10.5.3), las instrucciones lock (§8.12) y la creación y terminación de subprocesos. El entorno de ejecución puede cambiar el orden de ejecución de un programa de C#, con las siguientes restricciones:

* Se conservará la dependencia de datos dentro de un subproceso de ejecución. Es decir, el valor de cada variable se calculará como si todas las instrucciones del subproceso se ejecutaran en el orden del programa original.
* Se conservan las reglas de orden de inicialización (§10.5.4 y §10.5.5).
* Se conserva el orden de los efectos secundarios respecto a las lecturas y escrituras volátiles (§10.5.3). Además, el entorno de ejecución no necesitará evaluar parte de una expresión si puede deducir que el valor de esa expresión no se utiliza y que no se producen efectos secundarios necesarios (incluidos los causados por la llamada a un método o el acceso a un campo volátil). Cuando un evento asincrónico (tal como una excepción iniciada por otro subproceso) interrumpe la ejecución de un programa, no habrá garantías de que los efectos secundarios observables resulten visibles en el orden del programa original.

# Tipos

Los tipos del lenguaje C# se dividen en dos categorías principales: tipos de valor y tipos de referencia. Ambos, los tipos de valor y los tipos de referencia, pueden ser tipos genéricos, que adoptan uno o más parámetros de tipo. Los parámetros de tipo pueden designar tanto a tipos de valor como a tipos de referencia.

type:  
value-type  
reference-type   
type-parameter

Una tercera categoría de tipos, los punteros, sólo está disponible en el código no seguro. Esta categoría se explica con más detalle en la sección §18.2.

Los tipos de valor se diferencian de los tipos de referencia en que sus variables contienen directamente sus datos, mientras que las variables de los tipos de referencia contienen referencias a sus datos, que se conocen como objetos. En el caso de los tipos de referencia, es posible que dos variables hagan referencia al mismo objeto y, por tanto, que las operaciones en una variable afecten al objeto al que hace referencia la otra variable. En el caso de los tipos de valor, cada variable tiene su propia copia de los datos, de manera que no es posible que las operaciones de una afecten a la otra.

El sistema de tipos de C# está unificado, de manera que un valor de cualquier tipo puede tratarse como un objeto. Todos los tipos de C# se derivan directa o indirectamente del tipo de clase object, que es la clase base definitiva de todos los tipos. Los valores de los tipos de referencia se tratan como objetos considerándolos sencillamente como del tipo object. Los valores de los tipos de valor se tratan como objetos mediante la realización de operaciones de conversión boxing y unboxing (§4.3).

## Tipos de valor

Un tipo de valor es un tipo struct o un tipo enum. C# proporciona un conjunto de tipos struct predefinidos denominados tipos simples. Los tipos simples se identifican mediante palabras reservadas.

value-type:  
struct-type  
enum-type

struct-type:  
type-name  
simple-type   
nullable-type

simple-type:  
numeric-type  
bool

numeric-type:  
integral-type  
floating-point-type  
decimal

integral-type:  
sbyte  
byte  
short  
ushort  
int  
uint  
long  
ulong  
char

floating-point-type:  
float  
double

nullable-type:  
non-nullable-value-type ?

non-nullable-value-type:  
type

enum-type:  
type-name

A diferencia de las variables de un tipo de referencia, las variables de los tipos de valor pueden contener el valor null sólo si el tipo de valor acepta valores NULL. Para cada tipo de valor que no acepta valores NULL hay un tipo de valor correspondiente que sí acepta valores NULL y que denota el mismo conjunto de valores más el valor null.

La asignación a una variable de un tipo de valor crea una copia del valor que se asigna. lo cual difiere de la asignación a una variable de un tipo de referencia, que copia la referencia, pero no el objeto identificado por ella.

### Tipo System.ValueType

Todos los tipos de valor se heredan implícitamente de la clase System.ValueType, que, a su vez, se hereda de la clase object. No es posible que cualquier tipo se derive de un tipo de valor y, por lo tanto, los tipos de valor son implícitamente tipos sealed (§10.1.1.2).

Observe que System.ValueType no es en sí ningún tipo de valor (value-type). Más bien, es un tipo de clase (class-type) del que se derivan automáticamente todos los tipos de valores (value-types).

### Constructores predeterminados

Todos los tipos de valor declaran implícitamente un constructor de instancia público sin parámetros denominado constructor predeterminado. El constructor predeterminado devuelve una instancia inicializada en cero conocida como el valor predeterminado del tipo de valor:

* Para todos los tipos simples (simple-types), el valor predeterminado es el generado por un modelo de bits de todo ceros:
* Para sbyte, byte, short, ushort, int, uint, long y ulong, el valor predeterminado es 0.
* Para char, el valor predeterminado es '\x0000'.
* Para float, el valor predeterminado es 0.0f.
* Para double, el valor predeterminado es 0.0d.
* Para decimal, el valor predeterminado es 0.0m.
* Para bool, el valor predeterminado es false.
* Para un tipo enum (enum-type) E, el valor predeterminado es 0, convertido en el tipo E.
* Para un tipo struct (struct-type), el valor predeterminado es el que se genera al configurar todos los campos de tipos de valores en su valor predeterminado y todos los tipos de referencia en null.
* Para un tipo que acepta valores NULL (nullable-type) el valor predeterminado es una instancia para la que la propiedad HasValue es falsa y la propiedad Value no está definida. El valor predeterminado también se conoce como el valor NULL del tipo que acepta valores NULL.

Igual que con cualquier otro constructor de instancia, se llama al constructor predeterminado de un tipo de valor mediante el operador new. Por razones de eficacia, el objetivo de este requisito no es lograr que la implementación genere una llamada de constructor. En el ejemplo siguiente, las dos variables i y j se inicializan en cero.

class A  
{  
 void F() {  
 int i = 0;  
 int j = new int();  
 }  
}

Dado que todos los tipos de valor implícitamente tienen un constructor de instancias público sin parámetros, un tipo struct no puede contener una declaración explícita de un constructor sin parámetros. No obstante, en un tipo struct se pueden declarar constructores de instancia con parámetros (§11.3.8).

### Tipos struct

Un tipo struct es un tipo de valor que puede declarar constantes, campos, métodos, propiedades, indizadores, operadores, constructores de instancia, constructores estáticos y tipos anidados. La declaración de los tipos struct se describe en §11.1.

### Tipos simples

C# proporciona un conjunto de tipos struct predefinidos denominados tipos simples. Los tipos simples se identifican mediante palabras reservadas, pero éstas son sencillamente alias de tipos struct predefinidos del espacio de nombres System, como se explica en la tabla siguiente.

|  |  |
| --- | --- |
| **Palabra reservada** | **Tipo con alias** |
| sbyte | System.SByte |
| byte | System.Byte |
| short | System.Int16 |
| ushort | System.UInt16 |
| int | System.Int32 |
| uint | System.UInt32 |
| long | System.Int64 |
| ulong | System.UInt64 |
| char | System.Char |
| float | System.Single |
| double | System.Double |
| bool | System.Boolean |
| decimal | System.Decimal |

Como un tipo simple equivale a un tipo struct, todos los tipos simples tienen miembros. Por ejemplo, int tiene los miembros declarados en System.Int32 y los miembros heredados de System.Object, y se permiten las siguientes instrucciones:

int i = int.MaxValue; // System.Int32.MaxValue constant  
string s = i.ToString(); // System.Int32.ToString() instance method  
string t = 123.ToString(); // System.Int32.ToString() instance method

Los tipos simples se diferencian de otros tipos struct en que permiten determinadas operaciones adicionales:

* La mayoría de los tipos simples permiten la creación de valores mediante la escritura de literales (§2.4.4). Por ejemplo, 123 es un literal del tipo int y 'a' es un literal del tipo char. C# no proporciona literales de tipos struct en general, y los valores no predeterminados de otros tipos struct se crean siempre en último término mediante constructores de instancia de dichos tipos struct.
* Cuando todos los operandos de una expresión son constantes de tipo simple, es posible que el compilador evalúe la expresión en tiempo de compilación. Dicha expresión se conoce por el nombre de expresión constante (constant-expression) (§7.19). Las expresiones que incluyen operadores definidos por otros tipos struct no se consideran expresiones constantes.
* Mediante las declaraciones const es posible declarar constantes de los tipos simples (§10.4). No es posible tener constantes de otros tipos struct, pero se consigue un efecto similar mediante campos static readonly.
* Las conversiones que involucran tipos simples pueden participar en la evaluación de los operadores de conversión definidos por otros tipos struct, pero un operador de conversión definido por el usuario no puede participar en la evaluación de otros operadores definidos por el usuario (§6.4.3).

### Tipos enteros

C# admite los nueve tipos enteros: sbyte, byte, short, ushort, int, uint, long, ulong y char. Los tipos enteros tienen los siguientes tamaños e intervalos de valores:

* El tipo sbyte, que representa enteros de 8 bits con signo con valores comprendidos entre –128 y 127.
* El tipo byte, que representa enteros de 8 bits sin signo con valores comprendidos entre 0 y 255.
* El tipo short, que representa enteros de 16 bits con signo con valores comprendidos entre –32768 y 32767.
* El tipo ushort, que representa enteros de 16 bits sin signo con valores comprendidos entre 0 y 65535.
* El tipo int, que representa enteros de 32 bits con signo con valores comprendidos entre –2147483648 y 2147483647.
* El tipo uint, que representa enteros de 32 bits sin signo con valores comprendidos entre 0 y 4294967295.
* El tipo long, que representa enteros de 64 bits con signo con valores comprendidos entre –9223372036854775808 y 9223372036854775807.
* El tipo ulong, que representa enteros de 64 bits sin signo con valores comprendidos entre 0 y 18446744073709551615.
* El tipo char representa enteros sin signo de 16 bits con valores entre 0 y 65535. El conjunto de valores posibles para el tipo char corresponde al juego de caracteres Unicode. Aunque char tiene la misma representación que ushort, no todas las operaciones permitidas en un tipo están permitidas en el otro.

Los operadores unarios y binarios de tipos integrales siempre trabajan con precisión de 32 bits con signo, precisión de 32 bits sin signo, precisión de 64 bits con signo o precisión de 64 bits sin signo:

* Para el operador unario + y los operadores ~, el operando se convierte al tipo T, donde T es el primero de int, uint, long y ulong que puede representar completamente todos los valores posibles del operando. Después se ejecuta la operación con la precisión del tipo T y el tipo del resultado es T.
* Para el operador unario –, el operando se convierte al tipo T, donde T es el primero de int y long que puede representar completamente todos los valores posibles del operando. Después se ejecuta la operación con la precisión del tipo T y el tipo del resultado es T. El operador unario – no se puede aplicar a operandos del tipo ulong.
* Para los operadores binarios +, –, \*, /, %, &, ^, |, ==, !=, >, <, >= y <=, los operandos se convierten al tipo T, donde T es el primero de int, uint, long y ulong que puede representar completamente todos los valores posibles de los dos operandos. Después se ejecuta la operación con la precisión del tipo T, y el tipo del resultado es T (o bool para los operadores relacionales). Los operadores binarios no permiten que un operando sea de tipo long y el otro de tipo ulong.
* Para el operador binario << y los operadores >>, el operando izquierdo se convierte al tipo T, donde T es el primero de int, uint, long y ulong que puede representar completamente todos los valores posibles del operando. Después se ejecuta la operación con la precisión del tipo T y el tipo del resultado es T.

El tipo char se clasifica como un tipo entero, pero difiere de otros tipos enteros en dos aspectos:

* No existen conversiones implícitas desde otros tipos al tipo char. En concreto, aunque los tipos sbyte, byte y ushort tienen intervalos de valores que son totalmente representables mediante el tipo char, las conversiones implícitas de sbyte, byte o ushort a char no existen.
* Las constantes de tipo char deben escribirse como literales de caracteres (character-literals) o como literales de enteros (integer-literals) en combinación con una conversión al tipo char. Por ejemplo, (char)10 es igual que '\x000A'.

Los operadores e instrucciones checked y unchecked se utilizan para controlar la comprobación del desbordamiento para las operaciones y conversiones aritméticas de tipos enteros (§7.6.12). En un contexto checked, un desbordamiento produce un error de tiempo de compilación o causa una excepción System.OverflowException. En un contexto unchecked, los desbordamientos no se tienen en cuenta y los bits de orden superior que no son aceptables para el tipo de destino se descartan.

### Tipos de punto flotante

C# admite dos tipos de punto flotante: float y double. Los tipos float y double se representan mediante los formatos IEEE 754 de 32 bits de precisión sencilla y de 64 bits de precisión doble, que proporcionan los siguientes conjuntos de valores:

* Cero positivo y cero negativo. En la mayoría de las situaciones, cero positivo y cero negativo tienen un comportamiento idéntico al del valor simple cero, pero algunas operaciones distinguen entre los dos (§7.8.2).
* Infinito positivo e infinito negativo. Los infinitos son generados por operaciones como dividir por cero un número distinto de cero. Por ejemplo, el resultado de 1.0 / 0.0 es infinito positivo, y el de –1.0 / 0.0 es infinito negativo.
* El valor no numérico (Not-a-Number), abreviado normalmente como NaN. Los valores NaN se generan por operaciones de punto flotante no válidas, como dividir cero por cero.
* El conjunto finito de valores distintos de cero con el formato s × m × 2e, donde s es 1 o −1, y m y e están determinados por el tipo de punto flotante concreto: para float, 0 < m < 224 y −149 ≤ e ≤ 104, y para double, 0 < m < 253 y −1075 ≤ e ≤ 970. Los números de punto flotante sin normalizar se consideran valores válidos distintos de cero.

El tipo float puede representar valores comprendidos entre aproximadamente 1,5 × 10−45 y 3,4 × 1038 con una precisión de 7 dígitos.

El tipo double puede representar valores comprendidos entre aproximadamente 5,0 × 10−324 y 1,7 × 10308 con una precisión de 15-16 dígitos.

Si uno de los operandos de un operador binario es un tipo de punto flotante, el otro debe ser un tipo entero o un tipo de punto flotante, y la operación se evalúa como sigue:

* Si uno de los operandos es un tipo entero, se convierte al tipo de punto flotante del otro operando.
* Después, si uno de los operandos es de tipo double, el otro se convierte a double, la operación se ejecuta utilizando por lo menos el intervalo y la precisión double, y el tipo del resultado es double (o bool para los operadores relacionales).
* O bien, la operación se ejecuta utilizando por lo menos el intervalo y la precisión float y el tipo del resultado es float (o bool para los operadores relacionales).

Los operadores de punto flotante, incluidos los operadores de asignación, nunca producen excepciones. En lugar de ello, en situaciones excepcionales, las operaciones de punto flotante producen cero, infinito o NaN, como se explica a continuación:

* Si el resultado de una operación de punto flotante es demasiado pequeño para el formato de destino, el resultado de la operación es cero positivo o cero negativo.
* Si el resultado de una operación de punto flotante es demasiado grande para el formato de destino, el resultado de la operación es infinito positivo o infinito negativo.
* Si una operación de punto flotante no es válida, el resultado de la operación es NaN.
* Si uno o los dos operandos de una operación de punto flotante es NaN, el resultado de la operación es NaN.

Las operaciones de punto flotante pueden realizarse con mayor precisión que el tipo de resultado de la operación. Por ejemplo, algunas arquitecturas de hardware admiten el tipo de punto flotante “extended” o “long double” con un intervalo y precisión mayores que el tipo double, e implícitamente realizan todas las operaciones de punto flotante utilizando este tipo de mayor precisión. Se puede conseguir que las arquitecturas de hardware de esta clase realicen operaciones de punto flotante con menor precisión solo a cambio de un costo excesivo en el rendimiento; en vez de requerir una implementación que penalice tanto el rendimiento como la precisión, C# permite utilizar un tipo de mayor precisión para todas las operaciones de punto flotante. Aparte de proporcionar resultados más precisos, esto rara vez tiene efectos medibles. No obstante, en expresiones con la forma x \* y / z, donde la multiplicación genera un resultado que sale del intervalo de double, pero la siguiente división devuelve el resultado temporal al intervalo de double, el hecho de que la expresión se evalúe en un formato de intervalo mayor puede producir un resultado finito en lugar de infinito. Para forzar un valor de un tipo de punto flotante a la precisión exacta de su tipo, se puede usar una conversión explícita.

### Tipo decimal

El tipo decimal es un tipo de datos de 128 bits apto para cálculos financieros y monetarios. El tipo decimal puede representar valores comprendidos entre aproximadamente 1,0 × 10−28 a aproximadamente 7,9 × 1028 con 28-29 dígitos significativos.

El conjunto finito de valores de tipo decimal tiene la forma (–1)s× c × 10-e, donde el signo s es 0 o 1, el coeficiente c viene dado por 0 ≤ c < 296 y la escala e es 0 ≤ e ≤ 28. El tipo decimal no admite ceros con signo, valores infinitos ni valores NaN. Un decimal se representa como un entero de 96 bits elevado a diez. Para valores decimal con un valor absoluto menor que 1.0m, el valor es exacto sólo hasta la posición decimal 28a. Para valores decimal con un valor absoluto mayor o igual que 1.0m, el valor es exacto hasta el dígito 28 o 29. Al contrario que los tipos de datos float y double, los números fraccionarios decimales como 0,1 pueden representarse exactamente en la representación decimal. En las representaciones float y double, estos números con frecuencia son fracciones infinitas, lo que las hace más susceptibles de errores de redondeo.

Si uno de los operandos de un operador binario es de tipo decimal, el otro debe ser un tipo entero o un tipo decimal. Si está presente un operando de tipo entero, se convertirá a decimal antes de realizar la operación.

El resultado de una operación con valores de tipo decimal es el que resultaría de calcular un resultado exacto (conservando la escala, según lo definido por cada operador) y, a continuación, redondearlo para ajustarse a la representación. El resultado se redondea hasta el valor representable más próximo y, en caso de que un resultado tenga la misma proximidad con dos valores representables, se redondea al valor que tenga un número par en la posición del dígito menos significativo (esto se conoce como “redondeo de banca”). Un resultado cero siempre tiene un signo 0 y una escala 0.

Si una operación aritmética decimal genera un valor menor o igual que 5 × 10-29 en valor absoluto, el resultado de la operación es cero. Si una operación aritmética decimal produce un resultado que es demasiado largo para el formato decimal, se inicia una excepción System.OverflowException.

El tipo decimal tiene una precisión mayor, pero un intervalo menor, que los tipos de punto flotante. Por lo tanto, las conversiones de los tipos de punto flotante a decimal pueden producir excepciones de desbordamiento, y las conversiones de decimal a los tipos de punto flotante pueden causar una pérdida de precisión. Por estos motivos, no existen conversiones implícitas entre los tipos de punto flotante y decimal, y sin conversiones explícitas, no es posible combinar operandos de punto flotante y decimal en la misma expresión.

### El tipo booleano

El tipo bool representa cantidades lógicas booleanas. Los valores posibles del tipo bool son true y false.

No existen conversiones estándar entre bool y otros tipos. En concreto, el tipo bool es único y exclusivo de los tipos enteros, y un valor bool no puede sustituir a un valor integral, ni viceversa.

En los lenguajes C y C++, un valor integral o de punto flotante cero, o un puntero de valor null, pueden convertirse al valor booleano false, y un valor integral o de punto flotante distinto de cero o un puntero con un valor distinto de null pueden convertirse al valor booleano true. En C#, las conversiones de esta categoría se llevan a cabo mediante la comparación explícita de un valor integral o de punto flotante con cero, o mediante la comparación explícita de una referencia de objeto con null.

### Tipos de enumeración

Un tipo de enumeración es un tipo exclusivo con constantes con nombre. Cada tipo de enumeración tiene un tipo subyacente, que debe ser byte, sbyte, short, ushort, int, uint, long o ulong. El conjunto de valores del tipo de enumeración es el mismo que el del conjunto de valores del tipo subyacente. Los valores del tipo de la enumeración no se restringen a los valores de las constantes nombradas. Los tipos de enumeración se definen mediante declaraciones de enumeración (§14.1).

### Tipos que aceptan valores null

Un tipo que acepta valores NULL puede representar todos los valores de su tipo subyacente más un valor NULL adicional. Un tipo que acepta valores NULL se escribe como T?, donde T es el tipo subyacente. Esta sintaxis es la forma abreviada de System.Nullable<T> y las dos formas se pueden utilizar indistintamente.

Un tipo de valor que no admite valores NULL es cualquier tipo de valor distinto de System.Nullable<T> y se abrevia como T? (para cualquier T), más cualquier parámetro de tipo que esté restringido a ser un tipo de valor que no admite valores null (es decir, cualquier parámetro de tipo con una restricción struct). El tipo System.Nullable<T> especifica la restricción de tipo de valor para T (§10.1.5), lo que significa que el tipo de valor subyacente de un tipo que acepta valores NULL puede ser cualquier valor que no acepte valores NULL. El tipo subyacente de un tipo que no acepte valores NULL no puede ser un tipo que acepte valores NULL ni un tipo de referencia. Por ejemplo, int?? y string? no son tipos válidos.

Una instancia de un tipo T? que acepta valores NULL tiene dos propiedades de solo lectura públicas:

* Propiedad HasValue de tipo bool
* Propiedad Value de tipo T

Una instancia para la que HasValue es verdadera no es NULL. Una instancia que no es NULL contiene un valor conocido y Value devuelve dicho valor.

Una instancia para la que HasValue es falso se dice que es NULL. Una instancia NULL tiene un valor no definido. Un intento de leer el objeto Value de una instancia nula hace que no se genere un a excepción System.InvalidOperationException. El proceso de tener acceso a la propiedad Value de una instancia que acepta valores NULL se conoce como desajustar.

Además del constructor predeterminado, cada tipo T? que acepta valores NULL tiene un constructor público que adopta un argumento único del tipo T. Dado un valor x del tipo T, una invocación de constructor con la estructura

new T?(x)

crea una instancia T? que no acepta valores NULL para los que la propiedad Value es x. El proceso de crear una instancia que no acepta valores NULL de un tipo que sí acepta valores NULL para un valor dado se conoce como ajustar.

Existen conversiones implícitas desde el literal null a T? (§6.1.5) y de T a T? (§6.1.4).

## Tipos de referencia

Un tipo de referencia es un tipo de clase, un tipo de interfaz, un tipo de matriz o un tipo delegado.

reference-type:  
class-type  
interface-type  
array-type  
delegate-type  
dynamic

class-type:  
type-name  
object  
string

interface-type:  
type-name

array-type:  
non-array-type rank-specifiers

non-array-type:  
type

rank-specifiers:  
rank-specifier  
rank-specifiers rank-specifier

rank-specifier:  
[ dim-separatorsopt ]

dim-separators:  
,  
dim-separators ,

delegate-type:  
type-name

Un valor de tipo de referencia es una referencia a una instancia del tipo, que se conoce como objeto. El valor especial null es compatible con todos los tipos de referencia e indica la ausencia de una instancia.

### Tipos de clase

Un tipo de clase define una estructura de datos que contiene miembros de datos (constantes y campos), miembros de función (métodos, propiedades, eventos, indizadores, operadores, constructores de instancia, destructores y constructores estáticos) y tipos anidados. Los tipos de clase admiten la herencia, un mecanismo mediante el cual una clase derivada puede extender y especializar a las clases base. Las instancias de los tipos de clase se crean mediante expresiones de creación de objetos (object-creation-expressions) (§7.6.10.1).

Los tipos de clase se describen en §10.

Ciertos tipos de clases predefinidos tienen un significado especial en el lenguaje C#, tal como se describe en la siguiente tabla.

|  |  |
| --- | --- |
| **Tipo de clase** | **Descripción** |
| System.Object | Clase base definitiva de todos los demás tipos. Vea §4.2.2. |
| System.String | El tipo de cadena del lenguaje de C#. Vea §4.2.4. |
| System.ValueType | La clase base de todos los tipos de valor. Vea §4.1.1. |
| System.Enum | La clase base de todos los tipos enum. Vea §14. |
| System.Array | La clase base de todos los tipos de matriz. Vea §12. |
| System.Delegate | La clase base de todos los tipos de delegado. Vea §15. |
| System.Exception | La clase base de todos los tipos de excepción. Vea §16. |

### Tipo object

El tipo de clase object es la clase base definitiva de todos los demás tipos. Todos los tipos de C# se derivan directa o indirectamente del tipo de clase object.

La palabra clave object es simplemente un alias para la clase predefinida System.Object.

### Tipo dinámico

El tipo dynamic, al igual que object, puede hacer referencia a cualquier objeto. Cuando los operadores se aplican a expresiones de tipo dynamic, su resolución se pospone hasta la ejecución del programa. Así, si el operador no puede aplicarse legalmente al objeto al que se hace referencia, no se produce error durante la compilación. Se producirá una excepción cuando la resolución del operador dé error en tiempo de ejecución.

El tipo dinámico se describe con más detalle en §4.7, y el enlace dinámico en §7.2.2.

### Tipo string

El tipo string es un tipo de clase sealed que hereda directamente de object. Las instancias de la clase string representan cadenas de caracteres Unicode.

Los valores del tipo string pueden escribirse como literales de cadena (§2.4.4.5).

La palabra clave string es simplemente un alias para la clase predefinida System.String.

### Tipos de interfaz

Una interfaz define un contrato. Una clase o struct que implementa una interfaz debe adherirse a su contrato. Una interfaz puede heredar de varias interfaces base, y una clase o struct puede implementar varias interfaces.

Los tipos de interfaz se describen en §13.

### Tipos de matriz

Una matriz es una estructura de datos que contiene cero o más variables, a las que se obtiene acceso a través de índices calculados. Las variables contenidas en una matriz, también conocidas como elementos de la matriz, son todas del mismo tipo, denominado tipo de elemento de la matriz.

Los tipos de matriz se describen en §12.

### Tipos delegados

Un delegado es una estructura de datos que hace referencia a uno o más métodos. Para los métodos de instancia, se refiere también a sus correspondientes instancias de objeto.

El equivalente más cercano de un delegado en C o C++ es un puntero a una función pero, mientras que éste sólo puede hacer referencia a funciones estáticas, un delegado puede hacer referencia tanto a métodos estáticos como a métodos de instancia. En este caso, el delegado no sólo almacena una referencia al punto de entrada del método, sino que también almacena una referencia a la instancia de objeto para el que se invoca el método.

Los tipos delegados se describen en §15.

## Conversiones boxing y unboxing

El concepto de boxing y unboxing desempeña un papel central en el sistema de tipos de C#. Sirven como enlace entre los tipos de valor (value-type) y los tipos de referencia (reference-type), al permitir la conversión de cualquier valor de un tipo de valor (value-type) al tipo object y viceversa. Las conversiones boxing y unboxing habilitan una visión unificada del sistema de tipos en la que un valor de cualquier tipo puede tratarse básicamente como un objeto.

### Conversiones boxing

Una conversión boxing permite que un tipo de valor (value-type) se convierta implícitamente en un tipo de referencia. Existen los siguientes tipos de conversiones boxing:

* De cualquier tipo de valor (value-type) al tipo object.
* De cualquier tipo de valor (value-type) al tipo System.ValueType.
* De cualquier tipo de valor que no acepta valores NULL (non-nullable-value-type) a cualquier tipo de interfaz (interface-type) implementada por el tipo de valor (value-type).
* De cualquier tipo que acepta valores NULL (nullable-type) a cualquier tipo de interfaz (interface-type) implementada por el tipo subyacente del tipo que acepta valores NULL (nullable-type).
* De cualquier tipo enum (enum-type) al tipo System.Enum.
* De cualquier tipo que acepta valores NULL (nullable-type) con un tipo enum (enum-type) subyacente al tipo System.Enum.

Tenga en cuenta que una conversión implícita desde un parámetro de tipo se ejecutará como una conversión boxing si, en tiempo de ejecución, termina convirtiéndose desde un tipo de valor a un tipo de referencia (§6.1.10).

La conversión boxing de un tipo de valor que no acepta valores NULL (non-nullable-value-type) consiste en asignar una instancia del objeto y después copiar el tipo de valor que no acepta valores NULL (non-nullable-value-type) en esa instancia.

La conversión boxing de un valor de un tipo que acepta valores NULL (nullable-type) produce una referencia NULL si se trata de un valor null (HasValue es false), o el resultado de desajustar y realizar una conversión boxing en el valor subyacente de lo contrario.

El proceso real de conversión boxing del valor de un tipo de valor que no acepta valores NULL (non-nullable-value-type) se entiende mejor si uno se imagina la existencia de una clase boxing genérica, que se comporta como si estuviera declarada de la siguiente forma:

sealed class Box<T>: System.ValueType  
{  
 T value;

public Box(T t) {  
 value = t;  
 }  
}

La conversión boxing de un valor v de tipo T consiste ahora en ejecutar la expresión new Box<T>(v) y devolver la instancia resultante como un valor de tipo object. Por lo tanto, las instrucciones

int i = 123;  
object box = i;

conceptualmente se corresponden con

int i = 123;  
object box = new Box<int>(i);

La conversión boxing de una clase como Box<T> no existe en realidad y el tipo dinámico de un valor al que se ha aplicado la conversión boxing no es realmente un tipo de clase. En lugar de ello, un valor convertido mediante boxing de tipo T tiene el tipo dinámico T, y una comprobación tipo dinámica que usa el operador is sencillamente puede hacer referencia al tipo T. Por ejemplo,

int i = 123;  
object box = i;  
if (box is int) {  
 Console.Write("Box contains an int");  
}

devolverá la cadena “Box contains an int” en la consola.

Una conversión boxing implica la creación de una copia del valor al que se aplica la conversión. Esto es distinto de la conversión de un tipo de referencia (reference-type) a un tipo object, en la cual el valor sigue haciendo referencia a la misma instancia y sencillamente se considera como el tipo object menos derivado. Por ejemplo, dada la declaración

struct Point  
{  
 public int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

las siguientes instrucciones

Point p = new Point(10, 10);  
object box = p;  
p.x = 20;  
Console.Write(((Point)box).x);

muestran en la consola el valor 10, porque la operación boxing implícita que ocurre en la asignación de p a box causa la copia del valor de p. En cambio, si Point se hubiera declarado como class, el resultado sería 20, puesto que p y box harían referencia a la misma instancia.

### Conversiones Unboxing

Una conversión unboxing permite que un tipo de referencia (reference-type) se convierta explícitamente en un tipo de valor (value-type). Existen los siguientes tipos de conversiones unboxing:

* Del tipo object a cualquier tipo de valor (value-type).
* Del tipo System.ValueType a cualquier tipo de valor (value-type).
* De cualquier tipo de interfaz (interface-type) a cualquier tipo de valor que no acepta valores NULL (non-nullable-value-type) que implementa el tipo de interfaz (interface-type).
* De cualquier tipo de interfaz (interface-type) a cualquier tipo que acepta valores NULL (nullable-type) cuyo tipo subyacente implementa el tipo de interfaz (interface-type).
* Del tipo System.Enum a cualquier tipo enum (enum-type).
* De cualquier tipo System.Enum a cualquier tipo que acepta valores NULL (nullable-type) con un tipo enum (enum-type) subyacente.

Tenga en cuenta que una conversión explícita a un parámetro de tipo se ejecutará como una conversión unboxing si, en tiempo de ejecución, termina convirtiéndose desde un tipo de referencia a un tipo de valor (§6.2.6).

Una operación unboxing a un tipo de valor que no acepta valores NULL (non-nullable-value-type) consiste en comprobar primero que la instancia del objeto es un valor al que se ha aplicado la conversión boxing de un tipo de valor determinado que no acepta valores NULL (non-nullable-value-type), y copiar después el valor fuera de la instancia.

La conversión unboxing a un tipo que acepta valores NULL (nullable-type) produce el valor NULL de dicho tipo si el operando de origen es null o el resultado desajustado de realizar una conversión unboxing en la instancia de objeto al tipo de valor subyacente desde el tipo que acepta valores NULL (nullable-type) en caso contrario.

En cuanto a la clase boxing imaginaria descrita en la sección anterior, una conversión unboxing de un objeto box a un tipo de valor T consiste en ejecutar la expresión ((Box<T>)box).value. Por lo tanto, las instrucciones

object box = 123;  
int i = (int)box;

conceptualmente se corresponden con

object box = new Box<int>(123);  
int i = ((Box<int>)box).value;

Para que una conversión unboxing a un tipo de valor dado que no acepta valores NULL (non-nullable-value-type) se ejecute correctamente en tiempo de ejecución, el valor del operando de origen debe ser una referencia a un objeto creado mediante una conversión boxing de ese tipo de valor que no acepta valores NULL (non-nullable-value-type). Si el operando del código fuente es null, se produce una excepción System.NullReferenceException. Si el operando de origen es una referencia a un objeto incompatible, se producirá una excepción System.InvalidCastException.

Para que una conversión unboxing a un tipo de valor dado que acepta valores NULL se ejecute correctamente en tiempo de ejecución, el valor del operando de origen debe ser null o una referencia a un valor creado mediante la conversión boxing del tipo de valor que no acepta valores NULL subyacente del tipo que acepta valores NULL. Si el operando de origen es una referencia a un objeto incompatible, se producirá una excepción System.InvalidCastException.

## Tipos construidos

Una declaración de tipo genérico, por sí misma, denota un tipo genérico sin enlazar que se utiliza como “huella” para formar muchos tipos diferentes aplicando argumentos de tipo. Los argumentos de tipo se escriben entre corchetes angulares (< y >) inmediatamente después del nombre de la declaración de tipo genérico. Un tipo que incluye al menos un argumento de tipo se llama un tipo construido. Un tipo construido se puede utilizar en la mayoría de lugares en el idioma en el que puede aparecer el nombre del tipo. Un tipo genérico independiente sólo se puede utilizar dentro de una expresión typeof (typeof-expression) (§7.6.11).

Los tipos construidos también se pueden utilizar en expresiones como nombres simples (§7.6.2) o cuando se obtiene acceso a un miembro (§7.6.4).

Al evaluar un nombre de espacio de nombres o de tipo (namespace-or-type-name) solo se tienen en cuenta los tipos genéricos con el número correcto de parámetros tipo. Por lo tanto, es posible utilizar el mismo identificador para identificar tipos diferentes siempre y cuando los tipos tengan diferente número de parámetros de tipo. Esto resulta útil al mezclar clases genéricas y no genéricas en el mismo programa:

namespace Widgets  
{  
 class Queue {...}  
 class Queue<TElement> {...}  
}

namespace MyApplication  
{  
 using Widgets;

class X  
 {  
 Queue q1; // Non-generic Widgets.Queue  
 Queue<int> q2; // Generic Widgets.Queue  
 }  
}

Es posible que un nombre de tipo (type-name) identifique un tipo construido aunque éste no especifique directamente parámetros de tipo. Esto puede ocurrir si un tipo está anidado dentro de una declaración de clase genérica y el tipo de instancia de la declaración contenedor se utiliza de manera implícita para la búsqueda de nombres (§10.3.8.6):

class Outer<T>  
{  
 public class Inner {...}

public Inner i; // Type of i is Outer<T>.Inner  
}

En código no seguro, un tipo construido no se puede utilizar como un tipo no administrado (unmanaged-type) (§18.2).

### Argumentos de tipo

Cada argumento de una lista de argumentos de tipo es simplemente un tipo (type).

type-argument-list:  
< type-arguments >

type-arguments:  
type-argument  
type-arguments , type-argument

type-argument:  
type

En código no seguro (§18), es posible que un argumento de tipo (type-argument) no sea un tipo de puntero. Cada argumento de tipo debe respetar todas las restricciones del parámetro tipo correspondiente (§10.1.5).

### Tipos cerrados y abiertos

Todos los tipos se pueden clasificar como tipos abiertos o tipos cerrados. Un tipo abierto es un tipo que implica parámetros de tipo. Concretamente:

* Un parámetro de tipo define un tipo abierto.
* Un tipo de matriz es un tipo abierto solamente si su tipo de elemento es un tipo abierto.
* Un tipo construido es un tipo abierto sólo si uno o más de sus argumentos de tipo es un tipo abierto. Un tipo anidado construido es un tipo abierto sólo si uno o más de sus argumentos de tipo o los argumentos de tipo de sus tipos contenedores son un tipo abierto.

Un tipo cerrado es un tipo que no es abierto.

En tiempo de ejecución, todos los códigos de una declaración de tipo genérico se ejecutan en el contexto de un tipo construido cerrado creado al aplicar argumentos de tipo a la declaración genérica. Cada parámetro de tipo dentro del tipo genérico está enlazado a un tipo concreto en tiempo de ejecución. El proceso en tiempo de ejecución de todas las instrucciones y expresiones siempre tiene lugar con tipos cerrados; los tipos abiertos sólo tienen lugar durante el proceso en tiempo de compilación.

Cada tipo construido cerrado tiene su propio conjunto de variables estáticas, que no se comparten con otros tipos construidos cerrados. Los tipos abiertos no se dan en tiempo de ejecución y, por lo tanto, no tienen variables estáticas asociadas. Dos tipos construidos cerrados son el mismo tipo si están construidos desde el mismo tipo genérico independiente y sus argumentos de tipo correspondientes son el mismo tipo.

### Tipos enlazados y sin enlazar

El término tipo sin enlazar hace referencia a un tipo no genérico o a un tipo genérico sin enlazar. El término tipo enlazado hace referencia a un tipo no genérico o a un tipo construido.

Un tipo sin enlazar hace referencia a la entidad declarada por una declaración de tipo. Un tipo genérico sin enlazar no es un tipo en sí mismo, y no se puede usar como tipo de una variable, argumento o valor devuelto, ni como un tipo base. La única construcción en la que se puede hacer referencia a un tipo genérico sin enlazar es la expresión typeof (§7.6.11).

### Cumplimiento de las restricciones

Siempre que se haga referencia a un tipo construido o a un método genérico, los argumentos de tipo suministrados se cotejan con las restricciones de parámetro de tipo declaradas en el tipo o método genéricos (§10.1.5). Para cada cláusula where, el argumento de tipo A que corresponde al parámetro de tipo denominado se coteja con cada una de las restricciones de la siguiente manera:

* Si la restricción es un tipo de clase o de interfaz, o un parámetro de tipo, C debe representar dicha restricción con los argumentos de tipo suministrados sustituidos por cualquier parámetro de tipo que aparezca en la restricción. Para cumplir la restricción se debe dar el caso de que el tipo A se pueda convertir en el tipo C por uno de los siguientes métodos:
* Una conversión de identidad (§6.1.1)
* Una conversión de referencia implícita (§6.1.6)
* Una conversión boxing (§6.1.7), siempre y cuando el tipo A sea un tipo de valor que no acepta valores NULL.
* Una conversión de referencia implícita, boxing o de parámetro de tipo a partir de un parámetro de tipo A a C.
* Si la restricción es la restricción de tipo de referencia (class), el tipo A debe cumplir las siguientes condiciones:
* A es un tipo de interfaz, de clase, de delegado o de matriz. Tenga en cuenta que System.ValueType y System.Enum son tipos de referencia que cumplen esta restricción.
* A es un parámetro de tipo que se sabe que es un tipo de referencia (§10.1.5).
* Si la restricción es la restricción de tipo de valor (struct), el tipo A debe cumplir las siguientes condiciones:
* A es un tipo struct o un tipo enum, pero no un tipo que acepte valores NULL. Tenga en cuenta que System.ValueType y System.Enum son tipos de referencia que no cumplen esta restricción.
* A es un tipo de parámetro que tiene la restricción de tipo de valor (§10.1.5).
* Si la restricción es la restricción del constructor new(), el tipo A no debe ser abstract y debe tener un constructor público sin parámetros. Esta condición se cumple si una de las siguientes afirmaciones es verdadera:
* A es un tipo de valor puesto que todos los tipos de valor tienen un constructor público predeterminado (§4.1.2).
* A es un parámetro de tipo que tiene la restricción de constructor (§10.1.5).
* A es un tipo de parámetro que tiene la restricción de tipo de valor (§10.1.5).
* A es una clase que no es abstract y contiene un constructor public explícitamente declarado sin parámetros.
* A no es abstract y tiene un constructor predeterminado (§10.11.4).

Se genera un error en tiempo de compilación si los argumentos de tipo dados no cumplen una o más de las restricciones de un parámetro de tipo.

Puesto que los parámetros tipo no se heredan, tampoco se heredan las restricciones. En el siguiente ejemplo, D debe especificar la restricción en su parámetro de tipo T para que T cumpla con la restricción impuesta por la clase base B<T>. Por el contrario, no es necesario que E especifique una restricción porque List<T> implementa IEnumerable para cualquier T.

class B<T> where T: IEnumerable {...}

class D<T>: B<T> where T: IEnumerable {...}

class E<T>: B<List<T>> {...}

## Parámetros de tipo

Un parámetro de tipo es un identificador que designa un tipo de valor o tipo de referencia al que el parámetro está enlazado en tiempo de ejecución.

type-parameter:  
identifier

Puesto que se puede crear una instancia para un parámetro de tipo con muchos argumentos de tipo reales diferentes, los parámetros de tipo tienen operaciones y restricciones ligeramente diferentes a las de otros tipos. A continuación se enumeran algunas de ellas:

* Un parámetro de tipo no se puede utilizar directamente para declarar o una clase base (§10.2.4) o una interfaz (§13.1.3).
* Las reglas para la búsqueda de miembros en parámetros de tipo dependen de las restricciones, si existen, que se aplican al parámetro de tipo. En la sección §7.4 se proporcionan más detalles.
* Las conversiones disponibles para un parámetro de tipo dependen de las restricciones, si existen, que se apliquen al parámetro de tipo. En las secciones §6.1.10 y §6.2.6 se proporcionan más detalles.
* El literal null no se puede convertir a un tipo dado por un parámetro de tipo a menos que el parámetro de tipo sea un tipo de referencia (§6.1.10). Sin embargo, se puede utilizar en su lugar una expresión default (§7.6.13). Además, un valor con un tipo dado por un parámetro de tipo se puede comparar con null utilizando == y != (§7.10.6), a menos que el parámetro de tipo tenga la restricción de tipo de valor.
* Una expresión new (§7.6.10.1) solo se puede utilizar con un parámetro de tipo si el parámetro de tipo está restringido por una restricción de constructor (constructor-constraint) o la restricción del tipo de valor (§10.1.5).
* Un parámetro de tipo no se puede utilizar en cualquier parte dentro de un atributo.
* Un parámetro de tipo no se puede utilizar en un acceso a miembros (§7.6.4) ni en un nombre de tipo (§3.8) para identificar un miembro static o un tipo anidado.
* En código no seguro, un parámetro de tipo no se puede utilizar como un tipo no administrado (unmanaged-type) (§18.2).

Al igual que un tipo, los parámetros de tipo son una construcción en tiempo de compilación. En tiempo de ejecución, cada parámetro de tipo está enlazado a un tipo en tiempo de ejecución que se especifica al proporcionar un argumento de tipo a la declaración de tipo genérico. Por lo tanto, el tipo de una variable declarado con un parámetro de tipo será un tipo construido cerrado en tiempo de ejecución (§4.4.2). La ejecución en tiempo de ejecución de todas las instrucciones y expresiones que implican los parámetros de tipo utiliza el tipo real que se proporcionó como argumento de tipo para dicho parámetro.

## Tipos de árbol de expresiones

Los árboles de expresión permiten representar las expresiones lambda como estructuras de datos en lugar de como código ejecutable. Los árboles de expresiones son valores de tipos de árbol de expresiones con el formato System.Linq.Expressions.Expression<D>, donde D es cualquier tipo delegado. En adelante haremos referencia a estos tipos mediante la forma abreviada Expression<D>.

Si existe una conversión desde una expresión lambda a un tipo delegado D, también existe una conversión al tipo de árbol de expresión Expression<D>. Mientras que la conversión de una expresión lambda a un tipo delegado genera un delegado que hace referencia a código ejecutable para la expresión lambda, la conversión a un tipo de árbol de expresión crea una representación de árbol de expresión de la expresión lambda.

Los árboles de expresión son representaciones eficaces de datos en memoria de expresiones lambda y hacen que la estructura de la expresión lambda sea transparente y explícita.

Al igual que un tipo delegado D, se considera que Expression<D> tiene tipos devueltos y de parámetro, que son los mismos de D.

En el ejemplo siguiente se representa una expresión lambda como código ejecutable y como árbol de expresión. Debido a que existe una conversión a Func<int,int>, también existe una conversión a Expression<Func<int,int>>:

Func<int,int> del = x => x + 1; // Code

Expression<Func<int,int>> exp = x => x + 1; // Data

Siguiendo estas asignaciones, el delegado del hace referencia a un método que devuelve x + 1, y el árbol de expresiones exp hace referencia a la estructura de datos que describe la expresión x => x + 1.

Tanto la definición exacta del tipo genérico Expression<D> como las reglas precisas para construir un árbol de expresión cuando una expresión lambda se convierte a un tipo de árbol de expresión superan el ámbito de este documento.

Es importante señalar dos cosas:

* No todas las expresiones lambda se pueden convertir a árboles de expresión. Por ejemplo, no se pueden representar expresiones lambda con cuerpos de instrucción, ni expresiones lambda que contengan expresiones de asignaciones. Incluso en estos casos existe una conversión, pero no se producirá correctamente en tiempo de compilación. Estas excepciones se indican en §6.5.
* Expression<D> ofrece un método de instancia Compile que produce un delegado de tipo D:

Func<int,int> del2 = exp.Compile();

Al invocar a este delegado se provoca la ejecución del código representado por el árbol de expresiones. Así, dadas las definiciones anteriores, del y del2 son equivalentes, y las dos instrucciones siguientes tendrán el mismo efecto:

int i1 = del(1);

int i2 = del2(1);

Después de ejecutar este código, tanto i1 como i2 tendrán el valor 2.

## Tipo dinámico

El tipo dynamic tiene un significado especial en C#. Su objetivo es permitir el enlace dinámico, que se describe en detalle en §7.2.2.

dynamic se considera idéntico a object excepto en los siguientes aspectos:

* Las operaciones en expresiones de tipo dynamic pueden enlazarse dinámicamente (§7.2.2).
* La inferencia de tipos (§7.5.2) prefiere dynamic antes que object si ambos son candidatos.

Debido a esta equivalencia, lo siguiente es cierto:

* Hay una conversión de identidad implícita entre object y dynamic, y entre los tipos construidos que son iguales cuando reemplazan dynamic con object
* Las conversiones implícitas y explícitas de y en object también se aplican de y en dynamic.
* Las signaturas de método que son iguales cuando se reemplaza dynamic con object se consideran la misma signatura.

El tipo dynamic no se distingue de object en tiempo de ejecución.

Una expresión de tipo dynamic se denomina expresión dinámica.

# Variables

Las variables representan ubicaciones de almacenamiento. Toda variable tiene un tipo, que determina los valores que pueden almacenarse en ella. C# es un lenguaje con seguridad de tipos, y el compilador de C# garantiza que los valores almacenados en variables siempre son del tipo apropiado. El valor de una variable puede modificarse mediante una asignación o con el uso de los operadores ++ y ‑‑.

Una variable debe estar definitivamente asignada (§5.3) para que pueda obtenerse su valor.

Como se explica en las secciones siguientes, las variables pueden ser asignadas inicialmente o no asignadas inicialmente. Una variable asignada inicialmente tiene un valor inicial bien definido y siempre se considera asignada definitivamente. Una variable no asignada inicialmente no tiene un valor inicial. Para que una variable no asignada inicialmente se considere asignada definitivamente en una ubicación determinada, debe ocurrir una asignación de la variable en todas las rutas de ejecución posibles que conduzcan a dicha ubicación.

## Categorías de variables

C# define siete categorías de variables: variables estáticas, variables de instancia, elementos de matriz, parámetros de valores, parámetros de referencia, parámetros de salida y variables locales. En las próximas secciones se describen estas categorías.

En el ejemplo

class A  
{  
 public static int x;  
 int y;

void F(int[] v, int a, ref int b, out int c) {  
 int i = 1;  
 c = a + b++;  
 }  
}

x es una variable estática, y es una variable de instancia, v[0] es un elemento de matriz, a es un parámetro de valores, b es un parámetro de referencias, c es un parámetro de salida e i es una variable local.

### Variables estáticas

Un campo declarado con el modificador static se denomina variable estática. Una variable estática se genera antes de la ejecución del constructor estático (§10.12) de su tipo contenedor, y deja de existir a la vez que el dominio de aplicación asociado.

El valor inicial de una variable estática es el valor predeterminado (§5.2) del tipo de la variable.

Para los fines de comprobación de asignación definitiva, una variable estática se considera asignada inicialmente.

### Variables de instancia

Un campo declarado sin el modificador static se denomina variable de instancia.

#### Variables de instancia en clases

Una variable de instancia de una clase se genera cuando se crea una nueva instancia de esa clase, y deja de existir cuando no hay referencias a esa instancia y se ha ejecutado el destructor de la instancia (si existe).

El valor inicial de una variable de instancia de una clase es el valor predeterminado (§5.2) del tipo de la variable.

Para los fines de comprobación de asignación definitiva, una variable de instancia de una clase se considera asignada inicialmente.

#### Variables de instancia en structs

Una variable de instancia de un struct tiene exactamente el mismo período de duración que la variable de struct a la que pertenece. Es decir, cuando se genera o deja de existir una variable de tipo struct, le ocurre lo mismo a las variables de instancia del struct.

El estado de asignación inicial de una variable de instancia de un struct es el mismo que el de la variable que contiene el struct. Es decir, si una variable de struct se considera asignada inicialmente, lo mismo le ocurre a sus variables de instancia, y si una variable de struct se considera no asignada inicialmente, sus variables de instancia tampoco lo están.

### Elementos matriciales

Los elementos de una matriz comienzan a existir cuando se crea una instancia de la matriz y cesa su existencia cuando dejan de existir referencias a dicha instancia.

El valor inicial de los elementos de una matriz es el valor predeterminado (§5.2) del tipo de los elementos de la matriz.

Para los fines de comprobación de asignación definitiva, un elemento de matriz se considera asignado inicialmente.

### Parámetros de valor

Un parámetro de valor es un parámetro que se declara sin ref o out.

Un parámetro de valor se genera al invocar el miembro de función (método, constructor de instancia, descriptor de acceso u operador) o la función anónima a la que pertenece el parámetro, y se inicializa con el valor del argumento especificado en la invocación. Un parámetro de valores normalmente deja de existir tras la devolución del miembro de función o la función anónima. Sin embargo, si una función anónima captura el parámetro de valor (§7.15), su duración se extiende al menos hasta que el árbol de expresiones o el delegado creados a partir de esa función anónima son susceptibles de recolección de elementos no utilizados.

Para los fines de comprobación de asignación definitiva, un parámetro de valores se considera asignado inicialmente.

### Parámetros de referencia

Un parámetro de referencia es un parámetro que se declara con un modificador ref.

Un parámetro de referencia no crea una nueva ubicación de almacenamiento. En lugar de ello, un parámetro de referencia representa la misma ubicación de almacenamiento que la variable especificada como argumento en la invocación de la función anónima o del miembro de función. Por lo tanto, el valor de un parámetro de referencia siempre es el mismo que el de la variable subyacente.

Se aplican las siguientes reglas de asignación definitiva a los parámetros de referencia. Deben tenerse en cuenta las distintas reglas de los parámetros de salida explicadas en §5.1.6.

* Una variable debe estar asignada definitivamente (§5.3) para que pueda pasarse como parámetro de referencia en la invocación de un miembro de función o un delegado.
* Dentro de un miembro de función o una función anónima, un parámetro de referencia se considera asignado inicialmente.

Dentro de un método de instancia o un descriptor de acceso de instancia de tipo struct, la palabra clave this tiene exactamente el mismo comportamiento que un parámetro de referencia de tipo struct (§7.6.7).

### Parámetros de salida

Un parámetro de salida es un parámetro que se declara con un modificador out.

Un parámetro de salida no crea una nueva ubicación de almacenamiento. En lugar de ello, un parámetro de salida representa la misma ubicación de almacenamiento que la variable especificada como argumento en la invocación del delegado o del miembro de función. Por lo tanto, el valor de un parámetro de salida siempre es el mismo que el de la variable subyacente.

Se aplican las siguientes reglas de asignación definitiva a los parámetros de salida. Deben tenerse en cuenta las distintas reglas de los parámetros de referencia explicadas en §5.1.5.

* Una variable no tiene por qué estar asignada definitivamente para poder pasarla como parámetro de salida en una invocación de delegado o miembro de función.
* Tras la finalización normal de la invocación de un delegado o miembro de función, cada variable que se pase como parámetro de salida se considera asignada en esa ruta de ejecución.
* Dentro de un miembro de función o una función anónima, un parámetro de salida se considera asignado inicialmente.
* Cada parámetro de salida de un miembro de función o de una función anónima se debe estar asignado definitivamente (§5.3) para poder obtener la salida del miembro de función o la función anónima con normalidad.

Dentro de un constructor de instancia del tipo struct, la palabra clave this tiene exactamente el mismo comportamiento que un parámetro de salida de tipo struct (§7.6.7).

### Variables locales

Una variable local se declara mediante una declaración de variable local (local-variable-declaration), lo cual puede ocurrir en un bloque (block), una instrucción for (for-statement), una instrucción switch (switch-statement) o una instrucción using (using-statement); o mediante una instrucción foreach (foreach-statement) o una cláusula catch específica (specific-catch-clause) en una instrucción try (try-statement).

La duración de una variable local representa la parte de la ejecución del programa durante la cual se garantiza que hay un espacio de almacenamiento reservado para ella. Esta duración se prolonga al menos desde la entrada en el bloque (block), instrucción for (for-statement), instrucción switch (switch-statement), instrucción using (using-statement), instrucción foreach (foreach-statement), o cláusula catch específica (specific-catch-clause) a los que está asociada, hasta que la ejecución de dicho bloque, instrucción for, instrucción switch, instrucción using, instrucción foreach, o cláusula catch específica finaliza de cualquier modo. (La entrada en un bloque contenedor o la llamada a un método suspende, aunque no finaliza, la ejecución del bloque, instrucción for, instrucción switch, instrucción using, instrucción foreach, o cláusula catch específica actuales.) Si una función anónima captura la variable local (§7.15.5.1), su duración se extiende al menos hasta que el árbol de expresiones o el delegado creados a partir de esa función anónima, junto con cualquier otro objeto que haga referencia a la variable capturada, sean susceptibles de recolección de elementos no utilizados.

Si se entra de manera recursiva en el bloque primario, la instrucción for, la instrucción switch, la instrucción using, la instrucción foreach, o la cláusula catch específica, se creará una instancia nueva de la variable local cada vez, y su inicializador de variable local (local-variable-initializer), si existe, se evaluará en cada ocasión.

Una variable local introducida por una declaración de variable local (local-variable-declaration) no se inicializa automáticamente y, por lo tanto, no tiene un valor predeterminado. Para los fines de comprobación de asignación definitiva, una variable local introducida por una declaración de variable local (local-variable-declaration) se considera no asignada inicialmente. Una declaración de variable local (local-variable-declaration) puede incluir un inicializador de variable local (local-variable-initializer), en cuyo caso la variable se considera definitivamente asignada solamente después de la expresión de inicialización (§5.3.3.4).

En el ámbito de una variable local introducida por una declaración de variable local (local-variable-declaration), es un error en tiempo de compilación hacer referencia a esa variable local en una posición textual que preceda al declarador de la variable local (local-variable-declarator). Si la declaración de variable local está implícita (§8.5.1), también se considera un error hace referencia a la variable local dentro de un declarador de variable local (local-variable-declarator).

Una variable local introducida por una instrucción foreach (foreach-statement) o una cláusula catch específica (specific-catch-clause) se considera asignada de forma definitiva en todo su ámbito.

La duración real de una variable local depende de la implementación. Por ejemplo, un compilador puede determinar estáticamente que una variable local de un bloque sólo se utiliza para una pequeña parte del bloque, y usar este análisis para que el compilador genere un código que dé como resultado que el almacenamiento de la variable tenga una duración más corta que la de su bloque contenedor.

El almacenamiento a que se refiere una variable de referencia local se reclama independientemente de la duración de esa variable de referencia local (§3.9).

## Valores predeterminados

Las categorías de variables siguientes se inicializan de forma automática a sus valores predeterminados:

* Variables estáticas.
* Variables de instancia de instancias de clase.
* Elementos matriciales.

El valor predeterminado de una variable depende de su tipo y se determina como sigue:

* Para una variable de un tipo de valor (value-type), el valor predeterminado es el mismo que el valor calculado por el constructor predeterminado del tipo de valor (value-type) (§4.1.2).
* Para una variable de tipo de referencia (reference-type), el valor predeterminado es null.

La inicialización a valores predeterminados, la suelen llevar a cabo el administrador de la memoria o el recolector de elementos no utilizados, que inicializan la memoria a todos los bits cero antes de asignarla para su uso. Por este motivo, es conveniente utilizar todos los bits cero para representar la referencia nula.

## Estado de asignación definitiva

En una ubicación dada dentro del código ejecutable de un miembro de función, se dice que una variable está asignada definitivamente si el compilador puede probar, mediante un análisis de flujo estático concreto (§5.3.3), que la variable se ha inicializado automáticamente o ha sido el destino de por lo menos una asignación. Dicho de una manera informal, las reglas de la asignación definitiva son:

* Una variable asignada inicialmente (§5.3.1) siempre se considera asignada definitivamente.
* Una variable no asignada inicialmente (§5.3.2) se considera asignada definitivamente en una ubicación dada si todas las rutas de ejecución posibles que llevan a ella contienen por lo menos uno de los siguientes elementos:
* Una asignación simple (§7.17.1) en la cual la variable es el operando izquierdo.
* Una expresión de invocación (§7.6.5) o una expresión de creación de objeto (§7.6.10.1) que pasa la variable como parámetro de salida.
* Para una variable local, una declaración de variable local (§8.5.1) que incluye un inicializador de variable.

La especificación formal subyacente a las reglas informales anteriores se describe en los apartados §5.3.1, §5.3.2 y §5.3.3.

El estado de asignación definitiva de las variables de instancia de una variable de tipo struct (struct-type) puede tener un seguimiento individual y también colectivo. Además de las reglas anteriores, se aplican las reglas siguientes a las variables de tipo struct (struct-type) y a sus variables de instancia:

* Una variable de instancia se considera asignada definitivamente si la variable de tipo struct (struct-type) que la contiene se considera asignada definitivamente.
* Una variable de tipo struct (struct-type) se considera asignada definitivamente si todas sus variables de instancia se consideran asignadas definitivamente.

La asignación definitiva es un requisito obligatorio en los contextos siguientes:

* Una variable debe estar asignada definitivamente en cada ubicación donde se obtenga su valor. Esto asegura que nunca puedan ocurrir valores no definidos. Se considera que la ocurrencia de una variable en una expresión obtiene el valor de la variable, excepto cuando:
* la variable es el operando izquierdo de una asignación simple,
* la variable se pasa como un parámetro de salida, o
* la variable es de tipo struct (struct-type) y opera como el operando izquierdo de un acceso a miembros.
* Una variable debe estar asignada definitivamente en cada ubicación donde se pase como un parámetro de referencia. Esto asegura que el miembro de función que se invoca puede considerar al parámetro de referencias como asignado inicialmente.
* Todos los parámetros de salida de un miembro de función deben estar asignados definitivamente en cada ubicación donde se producen devoluciones del miembro de función (mediante una instrucción return o cuando la ejecución llega al final del cuerpo del miembro de función). Esto asegura que los miembros de función no devuelven valores no definidos en los parámetros de salida, lo que permite al compilador considerar la invocación de un miembro de función que llame a una variable como un parámetro de salida equivalente a una asignación de la variable.
* La variable this de un constructor de instancia de tipo struct (struct-type) debe estar asignada definitivamente en cada ubicación donde se produzcan devoluciones del constructor de instancia.

### Variables asignadas inicialmente

Las siguientes categorías de variables se clasifican como asignadas inicialmente:

* Variables estáticas.
* Variables de instancia de instancias de clase.
* Variables de instancia de variables de struct asignadas inicialmente.
* Elementos matriciales.
* Parámetros de valor.
* Parámetros de referencia.
* Variables declaradas en una cláusula catch o en una instrucción foreach.

### Variables no asignadas inicialmente

Las siguientes categorías de variables se clasifican como no asignadas inicialmente:

* Variables de instancia de variables de struct no asignadas inicialmente.
* Parámetros de salida, incluida la variable this de constructores de instancia de struct.
* Variables locales, excepto aquellas declaradas en una cláusula catch o en una instrucción foreach.

### Reglas precisas para determinar asignaciones definitivas

Para determinar que cada variable utilizada está asignada definitivamente, el compilador debe utilizar un proceso equivalente al descrito en esta sección.

El compilador procesa el cuerpo de cada miembro de función que tenga una o más variables no asignadas inicialmente. El compilador determina un estado de asignación definitiva para cada variable v no asignada inicialmente, en cada uno de los siguientes puntos del miembro de función:

* Al principio de cada instrucción
* En el punto final (§8.1) de cada instrucción
* En cada arco que transfiere el control a otra instrucción o al punto final de una instrucción
* Al principio de cada expresión
* Al final de cada expresión

El estado de asignación definitiva de v puede ser:

* Asignada definitivamente. Indica que en todos los posibles flujos de control para este punto, a v se le ha asignado un valor.
* No asignada definitivamente. Para el estado de una variable al final de una expresión de tipo bool, el estado de una variable que no se ha asignado definitivamente puede caer (aunque no necesariamente) en uno de los siguientes subestados:
* Asignada definitivamente después de una expresión true. Este estado indica que si la expresión booleana se evalúa como true, v estará asignada definitivamente, pero si la expresión booleana se evalúa como false, no tiene por qué estar asignada.
* Asignada definitivamente después de una expresión false. Este estado indica que si la expresión booleana se evalúa como false v estará asignada definitivamente, pero si la expresión booleana se evalúa como true, no tiene por qué estar asignada.

Las reglas siguientes rigen la forma en la que se determina el estado de una variable v en cada posición.

#### Reglas generales para instrucciones

* v no se asigna definitivamente al principio del cuerpo de un miembro de función.
* v se asigna definitivamente al principio de cualquier instrucción inalcanzable.
* El estado de asignación definitiva de v al principio de cualquier otra instrucción se determina mediante la comprobación del estado de asignación definitiva de v en todas las transferencias de flujo de control que apunten al principio de esa instrucción. Si (y sólo en este caso) v se asigna definitivamente en todas estas transferencias de flujo de control, v se asigna también definitivamente al principio de la instrucción. El conjunto de posibles transferencias de flujo de control se determina del mismo modo que en la comprobación del alcance de la instrucción (§8.1).
* El estado de asignación definitiva de v en el punto final de una instrucción block, checked, unchecked, if, while, do, for, foreach, lock, using o switch se determina mediante la comprobación del estado de asignación definitiva de v en todas las transferencias de flujo de control que apunten al punto final de la instrucción. Si v se asigna definitivamente en todas estas transferencias de flujo de control, entonces v se asigna definitivamente en el punto final de la instrucción. De lo contrario, v no se asigna definitivamente en el punto final de la instrucción. El conjunto de posibles transferencias de flujo de control se determina del mismo modo que en la comprobación del alcance de la instrucción (§8.1).

#### Instrucciones de bloques e instrucciones checked y unchecked

El estado de asignación definitiva de v en la transferencia de control a la primera instrucción de la lista de instrucciones del bloque (o al punto final del bloque, si la lista de instrucciones está vacía), es el mismo que el de la instrucción de asignación definitiva de v antes de la instrucción de bloque checked o unchecked.

#### Instrucciones de expresiones

Para una instrucción de expresión stmt que consta de la expresión expr:

* v tiene el mismo estado de asignación definitiva al principio de expr que al principio de stmt.
* Si v se asigna definitivamente al final de expr, también lo hará en el punto final de stmt; de lo contrario, este punto no recibirá tal asignación.

#### Instrucciones de declaración

* Si stmt es una instrucción de declaración sin inicializadores, entonces v tiene el mismo estado de asignación definitiva en el punto final de stmt que al principio de stmt.
* Si stmt es una instrucción de declaración con inicializadores, el estado de asignación definitiva de v se determina como si stmt fuera una lista de instrucciones, con una instrucción de asignación para cada declaración con un inicializador (en el orden de la declaración).

#### Instrucciones If

Para una instrucción stmt if con el formato:

if ( expr ) then-stmt else else-stmt

* v tiene el mismo estado de asignación definitiva al principio de expr que al principio de stmt.
* Si v está definitivamente asignada al final de expr, también lo estará en la transferencia de flujo de control a then-stmt y a else-stmt, o bien al punto final de stmt si no hay cláusula else.
* Si v está “definitivamente asignada después de una expresión true” al final de expr, entonces también lo estará en la transferencia de flujo de control a then-stmt; si no hay cláusula else, no estará definitivamente asignada en la transferencia de flujo de control a else-stmt o al punto final de stmt.
* Si v tiene el estado “definitivamente asignada después de expresión false” al final de expr, entonces está definitivamente asignada en la transferencia de flujo de control a else-stmt y no lo está a then-stmt en la transferencia de flujo de control. Está definitivamente asignada al punto final de stmt solamente si está también definitivamente asignada al punto final de then-stmt.
* De lo contrario, v se considera no asignada definitivamente a then-stmt o a else-stmt en la transferencia de flujo de control o, si no hay cláusula else, al punto final de stmt.

#### Instrucciones Switch

En una instrucción switch, stmt con una expresión de control expr:

* El estado de asignación definitiva de v al principio de expr es el mismo que el estado de v al principio de stmt.
* El estado de asignación definitiva de v en la transferencia de flujo de control a una lista de instrucciones de bloque switch alcanzable es el mismo que al final de expr.

#### Instrucciones While

Para una instrucción while stmt con el formato:

while ( expr ) while-body

* v tiene el mismo estado de asignación definitiva al principio de expr que al principio de stmt.
* Si v está asignada definitivamente al final de expr, entonces está definitivamente asignada en la transferencia de flujo de control a while-body y al punto final de stmt.
* Si el estado de v al final de expr es “definitivamente asignada después de una expresión true”, está definitivamente asignada en la transferencia de flujo de control a while-body, pero no en el punto final de stmt.
* Si el estado de v al final de expr es “definitivamente asignada después de una expresión false”, está definitivamente asignada en la transferencia de flujo de control al punto final de stmt, pero no a while-body en la transferencia de flujo de control.

#### Instrucciones Do

Para una instrucción do stmt con el formato:

do do-body while ( expr ) ;

* v tiene el mismo estado de asignación definitiva en la transferencia de flujo de control, desde el principio de stmt a do-body, que al principio de stmt.
* v tiene el mismo estado de asignación definitiva al principio de expr que en el punto final de do-body.
* Si v está asignada definitivamente al final de expr, también lo está en la transferencia de flujo de control al punto final de stmt.
* Si el estado de v al final de expr es “asignada definitivamente después de una expresión false”, entonces está definitivamente asignada en la transferencia de flujo de control al punto final de stmt.

#### Instrucciones For

La comprobación de asignación definitiva de una instrucción for con el formato:

for ( for-initializer ; for-condition ; for-iterator ) embedded-statement

se realiza como si la instrucción estuviera escrita de la siguiente forma:

{  
 for-initializer ;  
 while ( for-condition ) {  
 embedded-statement ;  
 for-iterator ;  
 }  
}

Si se omite la condición for (for-condition) en la instrucción for, la evaluación o la asignación definitiva continuará como si se sustituyera la condición for (for-condition) por true en la expansión anterior.

#### Instrucciones Break, Continue y Goto

El estado de asignación definitiva de v en la transferencia de flujo de control causada por una instrucción break, continue o goto es el mismo que el de v al principio de la instrucción.

#### Instrucciones Throw

Para una instrucción stmt con el formato:

throw expr ;

El estado de asignación definitiva de v al principio de expr es el mismo que el de v al principio de stmt.

#### Instrucciones Return

Para una instrucción stmt con el formato:

return expr ;

* El estado de asignación definitiva de v al principio de expr es el mismo que el de v al principio de stmt.
* Si v es un parámetro de salida, debe asignarse definitivamente:
* después de expr
* o al final del bloque finally de un try-finally o try-catch-finally que incluya la instrucción return.

Para una instrucción stmt con el formato:

return ;

* Si v es un parámetro de salida, debe asignarse definitivamente:
* antes de stmt
* o al final del bloque finally de un try-finally o try-catch-finally que incluya la instrucción return.

#### Instrucciones Try-catch

Para una instrucción stmt con el formato:

try try-block  
catch(...) catch-block-1  
...  
catch(...) catch-block-n

* El estado de asignación definitiva de v al principio de try-block es igual al de v al principio de stmt.
* El estado de asignación definitiva de v al principio de catch-block-i (para cualquier i) es el mismo que el de v al principio de stmt.
* El estado de asignación definitiva de v en el punto final de stmt está asignado definitivamente si (y sólo si) v está definitivamente asignada en el punto final de try-block y de todos los catch-block-i (para cada i de 1 a n).

#### Instrucciones try-finally

Para una instrucción try stmt con el formato:

try try-block finally finally-block

* El estado de asignación definitiva de v al principio de try-block es igual al de v al principio de stmt.
* El estado de asignación definitiva de v al principio de finally-block es igual al de v al principio de stmt.
* v estará definitivamente asignada en el punto final de stmt si (y solo si) al menos una de las siguientes afirmaciones es verdadera:
* v está asignada definitivamente en el punto final de try-block
* v está asignada definitivamente en el punto final de finally-block

Si se realiza una transferencia de flujo de control (por ejemplo, una instrucción goto) que se inicia dentro de un bloque try (try-block) y termina fuera del bloque try, entonces v también se considerará asignada definitivamente en esa transferencia de flujo de control si también lo está en el punto final del bloque finally (finally-block). (Esta condición no es exclusiva: si v está asignada definitivamente por otra razón en esta transferencia de flujo de control, se considerará asignada definitivamente).

#### Instrucciones Try-finally-catch

El análisis de asignación definitiva de una instrucción try-catch-finally con el formato:

try try-block   
catch(...) catch-block-1  
...  
catch(...) catch-block-n  
finally finally-block

se realiza como si la instrucción fuera una instrucción try-finally que incluyera una instrucción try-catch:

try {  
 try try-block   
 catch(...) catch-block-1  
 ...  
 catch(...) catch-block-n  
}  
finally finally-block

El ejemplo siguiente demuestra cómo afectan los diferentes bloques de una instrucción try (§8.10) a la asignación definitiva. \r \h

class A  
{  
 static void F() {  
 int i, j;  
 try {  
 goto LABEL;  
 // neither i nor j definitely assigned  
 i = 1;  
 // i definitely assigned  
 }

catch {  
 // neither i nor j definitely assigned  
 i = 3;  
 // i definitely assigned  
 }

finally {  
 // neither i nor j definitely assigned  
 j = 5;  
 // j definitely assigned  
 }  
 // i and j definitely assigned  
 LABEL:;  
 // j definitely assigned  
  
 }  
}

#### Instrucciones Foreach

Para una instrucción foreach stmt con el formato:

foreach ( type identifier in expr ) embedded-statement

* El estado de asignación definitiva de v al principio de expr es el mismo que el estado de v al principio de stmt.
* El estado de asignación definitiva de v en la transferencia de flujo de control a una instrucción incrustada (embedded-statement) o al punto final de stmt es el mismo que el estado de v al final de expr.

#### Instrucciones Using

Para una instrucción using stmt con el formato:

using ( resource-acquisition ) embedded-statement

* El estado de asignación definitiva de v al principio de adquisición de recurso (resource-acquisition) es igual al estado de v al principio de stmt.
* El estado de asignación definitiva de v en la transferencia de flujo de control a una instrucción incrustada (embedded-statement) es igual al de v al final de la adquisición de recurso (resource-acquisition).

#### Instrucciones Lock

Para una instrucción lock stmt con el formato:

lock ( expr ) embedded-statement

* El estado de asignación definitiva de v al principio de expr es el mismo que el estado de v al principio de stmt.
* El estado de asignación definitiva de v en la transferencia de flujo de control a una instrucción incrustada (embedded-statement) es igual que el estado de v al final de la expresión (expr).

#### Instrucciones yield

Para una instrucción yield return stmt con el formato:

yield return expr ;

* El estado de asignación definitiva de v al principio de expr es el mismo que el estado de v al principio de stmt.
* El estado de asignación definitiva de v al principio de stmt es el mismo que el estado de v al principio de expr.

Una instrucción yield break no tiene ningún efecto sobre el estado de asignación definitiva.

#### Reglas generales para expresiones simples

La siguiente regla se aplica a estos tipos de expresiones: literales (§7.6.1), nombres simples (§7.6.2), expresiones de acceso a miembros (§7.6.4), expresiones de acceso a bases no indizadas (§7.6.8), expresiones typeof (§7.6.11) y expresiones de valor predeterminadas (§7.6.13).

* El estado de asignación definitiva de v al final de una expresión de este tipo es igual que el de v al principio de la expresión.

#### Reglas generales para expresiones con expresiones incrustadas

Las reglas siguientes se aplican a estos tipos de expresiones: expresiones entre paréntesis (§7.6.3), expresiones de acceso a elementos (§7.6.6), expresiones de acceso a bases con indización (§7.6.8), expresiones de incremento y decremento (§7.6.9, §7.7.5), expresiones de conversión de tipos (§7.7.6), expresiones unarias +, -, ~, \*, expresiones binarias +, -, \*, /, %, <<, >>, <, <=, >, >=, ==, !=, is, as, &, |, ^ (§7.8, §7.9, §7.10, §7.11), expresiones de asignación compuestas (§7.17.2), expresiones checked y unchecked (§7.6.12), y expresiones de creación de matrices y de delegados (§7.6.10).

Cada una de estas expresiones tiene una o más subexpresiones que se evalúan incondicionalmente en un orden fijo. Por ejemplo, el operador binario % evalúa el lado izquierdo del operador y, a continuación, el lado derecho. Una operación de indización evalúa la expresión indizada y, a continuación, cada una de las expresiones de índice por orden, de izquierda a derecha. Para una expresión expr que tiene las subexpresiones expr1, expr2, ..., exprn, evaluadas en ese orden:

* El estado de asignación definitiva de v al principio de expr1 es igual que el estado de asignación definitiva al principio de expr.
* El estado de asignación definitiva de v al principio de expri (siendo i mayor que uno) es igual que el estado al final de expri-1.
* El estado de asignación definitiva de v al principio de expr es igual que el estado al final de exprn.

#### Expresiones de invocación y expresiones de creación de objetos

Para una expresión de invocación expr con el formato:

primary-expression ( arg1 , arg2 , … , argn )

o una expresión de creación de objeto con la estructura:

new type ( arg1 , arg2 , … , argn )

* Para una expresión de invocación, el estado de asignación definitiva de v antes de la expresión primaria (primary-expression) es igual que el estado de v antes de expr.
* Para una expresión de invocación, el estado de asignación definitiva de v antes de arg1 es igual que el estado de v después de la expresión primaria (primary-expression).
* Para una expresión de creación de objeto, el estado de asignación definitiva de v antes de arg1 es igual que el estado de v antes de expr.
* Para cada argumento argi, el estado de asignación definitiva de v después de argi está determinado por las reglas de expresión normales omitiendo los modificadores ref u out.
* Para cada argumento argi, siendo i mayor que uno, el estado de asignación definitiva de v antes de argi es el mismo que el de v después de argi-1.
* Si se pasa la variable v como un argumento out (es decir, un argumento de forma "out v") en cualquiera de los argumentos, el estado de v después de expr se asigna definitivamente. De lo contrario, el estado de v después de expr es el mismo que el estado de v después de argn.
* Para los inicializadores de matriz (§7.6.10.4), inicializadores de objetos (§7.6.10.2), inicializadores de colección (§7.6.10.3) e inicializadores de objetos anónimos (§7.6.10.6), el estado de asignación definitiva está determinado por la expansión en cuyos términos están definidos estos constructores.

#### Expresiones de asignación simples

Para una expresión expr con el formato w = expr-rhs:

* El estado de asignación definitiva de v antes de expr-rhs es igual que el de v antes de expr.
* Si w es la misma variable que v, el estado de asignación definitiva de v después de expr se asigna definitivamente. De lo contrario, el estado de asignación definitiva de v después de expr será igual que el de v después de expr-rhs.

#### Expresiones &&

Para una expresión expr con el formato expr-first && expr-second:

* El estado de asignación definitiva de v antes de expr-first es igual que el de v antes de expr.
* El estado de asignación definitiva de v antes de la segunda expresión (expr-second) se asigna definitivamente, si el estado de v después de la primera expresión (expr-first) se asigna definitivamente o es “asignado definitivamente después de una expresión true”. En cualquier otro caso, no se asigna definitivamente.
* El estado de asignación definitiva de v después de expr se determina de la siguiente forma:
* Si la primera expresión (expr-first) es una expresión constante con el valor false, el estado de asignación definitiva de v después de expr es el mismo que el estado de asignación definitiva de v después de expr-first.
* De lo contrario, si el estado de v después de expr-first se asigna definitivamente, también lo hará el estado de v después de expr.
* De lo contrario, si el estado de v después de la segunda expresión (expr-second) se asigna definitivamente y el estado de v después de la primera expresión (expr-first) es “asignado definitivamente después de una expresión false”, entonces el estado de v después de la expresión (expr) se asigna definitivamente.
* De lo contrario, si el estado de v después de la segunda expresión (expr-second) se asigna definitivamente o es “asignado definitivamente después de expresión true”, entonces el estado de v después de la expresión (expr) es “asignado definitivamente después de una expresión true”.
* De lo contrario, si el estado de v después de la primera expresión (expr-first) es “asignado definitivamente después de una expresión false” y el estado de v después de segunda expresión (expr-second) es “asignado definitivamente después de una expresión false”, entonces el estado de v después de la expresión (expr) es “asignado definitivamente después de una expresión false”.
* De lo contrario, el estado de v después de expr no se asignará definitivamente.

En el ejemplo

class A  
{  
 static void F(int x, int y) {  
 int i;  
 if (x >= 0 && (i = y) >= 0) {  
 // i definitely assigned  
 }  
 else {  
 // i not definitely assigned  
 }  
 // i not definitely assigned  
 }  
}

la variable i se considera asignada definitivamente en una de las instrucciones incrustadas de una instrucción if pero no en la otra. En la instrucción if del método F, la variable i está asignada definitivamente en la primera instrucción incrustada porque la ejecución de la expresión (i = y) siempre precede a la ejecución de esta instrucción incrustada. Por el contrario, la variable i no está asignada definitivamente en la segunda instrucción incrustada, puesto que x >= 0 podría dar como resultado false y hacer que la variable i no se asignara.

#### Expresiones ||

Para una expresión expr con el formato expr-first || expr-second:

* El estado de asignación definitiva de v antes de expr-first es igual que el de v antes de expr.
* El estado de asignación definitiva de v antes de la segunda expresión (expr-second) se asigna definitivamente si el estado de v después de la primera expresión (expr-first) se asigna definitivamente o es “asignado definitivamente después de una expresión false”. En cualquier otro caso, no se asigna definitivamente.
* La instrucción de asignación definitiva de v después de expr se determina de la siguiente forma:
* Si la primera expresión (expr-first) es una expresión constante con el valor true, el estado de asignación definitiva de v después de expr es el mismo que el estado de asignación definitiva de v después de expr-first.
* De lo contrario, si el estado de v después de expr-first se asigna definitivamente, también lo hará el estado de v después de expr.
* De lo contrario, si el estado de v después de la segunda expresión (expr-second) se asigna definitivamente, y el estado de v después de la primera expresión (expr-first) es “asignado definitivamente después de una expresión true”, entonces el estado de v después de expr se asignará definitivamente.
* De lo contrario, si el estado de v después de la segunda expresión (expr-second) se asigna definitivamente o es “asignado definitivamente después de una expresión false”, entonces el estado de v después de la expresión (expr) es “asignado definitivamente después de una expresión false”.
* De lo contrario, si el estado de v después de la primera expresión (expr-first) es “asignado definitivamente después de una expresión true”, y el estado de v después de la segunda expresión (expr-second) es “asignado definitivamente después de una expresión true”, entonces el estado de v después de expr es “asignado definitivamente después de una expresión true”.
* De lo contrario, el estado de v después de expr no se asignará definitivamente.

En el ejemplo

class A  
{  
 static void G(int x, int y) {  
 int i;  
 if (x >= 0 || (i = y) >= 0) {  
 // i not definitely assigned  
 }  
 else {  
 // i definitely assigned  
 }  
 // i not definitely assigned  
 }  
}

la variable i se considera asignada definitivamente en una de las instrucciones incrustadas de una instrucción if pero no en la otra. En la instrucción if del método G, la variable i está asignada definitivamente en la segunda instrucción incrustada porque la ejecución de la expresión (i = y) siempre precede a la ejecución de esta instrucción incrustada. Por el contrario, la variable i no está asignada definitivamente en la primera instrucción incrustada, puesto que x >= 0 podría dar como resultado true y hace que la variable i no se asigne.

#### ! expresiones

Para una expresión expr con el formato ! expr-operand:

* El estado de asignación definitiva de v antes de expr-operand es igual que el estado de asignación definitiva de v antes de expr.
* El estado de asignación definitiva de v después de expr se determina de la siguiente forma:
* Si el estado de v después de expr-operand se asigna definitivamente, el estado de v después de expr se asigna definitivamente.
* Si el estado de v después de expr-operand no se asigna definitivamente, el estado de v después de expr no se asigna definitivamente.
* Si el estado de v después del operando de la expresión (expr-operand) es “asignado definitivamente después de una expresión false”, entonces el estado de v después de la expresión (expr) es “asignado definitivamente después de una expresión true”.
* Si el estado de v después del operando de la expresión (expr-operand) es “asignado definitivamente después de una expresión true, entonces el estado de v después de la expresión (expr) es “asignado definitivamente después de una expresión false”.

#### Expresiones ??

Para una expresión expr con el formato expr-first ?? expr-second:

* El estado de asignación definitiva de v antes de expr-first es igual que el de v antes de expr.
* El estado de asignación definitiva de v antes de expr-second es el mismo que el de v después de expr-first.
* La instrucción de asignación definitiva de v después de expr se determina de la siguiente forma:
* Si expr-first es una expresión constante (§7.19) con valor null, entonces el estado de v después de la expresión (expr) es el mismo que el estado de v después de expr-second.
* De lo contrario, el estado v después de expr será igual que el estado de asignación definitiva de v después de expr-first.

#### Expresiones ?:

Para una expresión expr con el formato expr-cond ? expr-true : expr-false:

* El estado de asignación definitiva de v antes de expr-cond es igual al estado de v antes de expr.
* v antes de una expresión true (expr-true) estará definitivamente asignada si (y solo si) se cumple al menos una de las siguientes afirmaciones:
* Una expresión condicional (expr-cond) es una expresión constante con el valor false
* el estado de v después de que la expresión condicional (expr-cond) se asigne definitivamente o se “asigne definitivamente después de una expresión true”.
* v antes de una expresión false (expr-false) estará definitivamente asignada si (y solo si) se cumple al menos una de las siguientes afirmaciones:
* Una expresión condicional (expr-cond) es una expresión constante con el valor true
* el estado de v después de que la expresión condicional (expr-cond) se asigne definitivamente o se “asigne definitivamente después de una expresión false”.
* El estado de asignación definitiva de v después de expr se determina de la siguiente forma:
* Si la expresión condicional (expr-cond) es una expresión constante (§7.19) con valor true, entonces el estado de v después de la expresión (expr) es el mismo que el estado de v después de la expresión para true (expr-true).
* De lo contrario, si la expresión condicional (expr-cond) es una expresión constante (§7.19) con valor false, entonces el estado de v después de la expresión (expr) es el mismo que el estado de v después de la expresión para false (expr-false).
* De lo contrario, si el estado de v después de expr-true y el estado de v después de expr-false se asignan definitivamente, entonces el estado de v después de expr también se asignará definitivamente.
* De lo contrario, el estado de v después de expr no se asignará definitivamente.

#### Funciones anónimas

Para una expresión lambda (lambda-expression) o una expresión de método anónimo (anonymous-method-expression expr) con un cuerpo (ya sea block o expression) body:

* El estado de asignación definitiva de una variable externa v antes del cuerpo (body) es igual que el estado de v antes de expr. Es decir, el estado de asignación definitiva de variables externas se hereda desde el contexto de la función anónima.
* El estado de asignación definitiva de una variable externa v después de expr será igual que el estado de v antes de expr.

El ejemplo

delegate bool Filter(int i);

void F() {  
 int max;

// Error, max is not definitely assigned  
 Filter f = (int n) => n < max;

max = 5;  
 DoWork(f);  
}

genera un error en tiempo de compilación ya que max no se asigna de manera definitiva donde se declara la función anónima. El ejemplo

delegate void D();

void F() {  
 int n;  
 D d = () => { n = 1; };

d();

// Error, n is not definitely assigned  
 Console.WriteLine(n);  
}

también genera un error en tiempo de compilación puesto que la asignación a n en la función anónima no afecta al estado de asignación definitivo de n fuera de la función anónima.

## Referencias de variables

Una referencia de variable (variable-reference) es una expresión (expression) que se clasifica como una variable. Una referencia de variable (variable-reference) es una ubicación de almacenamiento a la que se puede obtener acceso, tanto para obtener el valor actual, como para almacenar un valor nuevo.

variable-reference:  
expression

En C y C++, una referencia de variable (variable-reference) se conoce como lvalue.

## Atomicidad de las referencias de variable

Las lecturas y escrituras de los tipos de datos siguientes son atómicas: bool, char, byte, sbyte, short, ushort, uint, int, float y tipos de referencia. Además, las lecturas y escrituras de tipos enum cuyo tipo subyacente está en la lista anterior son también atómicas. No se garantiza que sean atómicas las lecturas y escrituras de otros tipos, como long, ulong, double y decimal, así como los tipos definidos por el usuario. Aparte de las funciones de biblioteca diseñadas para este propósito, no hay garantía de lectura, modificación ni escritura atómicas, como en el caso de incrementos o decrementos.

# Conversiones

Una conversión permite que una expresión sea tratada como parte de un tipo determinado. Una conversión puede hacer que una expresión de un tipo dado sea tratada como si tuviera un tipo diferente, o que una expresión sin un tipo lo consiga. Las conversiones pueden ser implícitas o explícitas, y esto determina si se requiere una conversión de tipo explícita. Por ejemplo, la conversión del tipo int al tipo long es implícita, por lo que las expresiones de tipo int pueden tratarse implícitamente como del tipo long. La conversión opuesta, del tipo long en int es explícita y, por ello, se requiere una conversión explícita.

int a = 123;  
long b = a; // implicit conversion from int to long  
int c = (int) b; // explicit conversion from long to int

Algunas conversiones están definidas por el lenguaje. Los programas también pueden definir sus propias conversiones (§6.4).

Algunas conversiones del lenguaje se definen de expresiones a tipos y otras de tipos a tipos. La conversión a partir de un tipo se aplica a todas las expresiones que tengan dicho tipo.

enum Color { Red, Blue, Green }

Color c0 = 0; // The expression 0 converts implicitly to enum types  
Color c1 = (Color)1; // other int expressions need explicit conversion

## Conversiones implícitas

Las siguientes conversiones se clasifican como conversiones implícitas:

* Conversiones de identidad
* Conversiones numéricas implícitas
* Conversiones de enumeración implícitas
* Conversiones implícitas que aceptan valores NULL
* Conversiones del literal NULL
* Conversiones de referencia implícitas
* Conversiones boxing
* Conversiones dinámicas implícitas
* Conversiones implícitas de expresión constante
* Conversiones definidas por el usuario implícitas
* Conversiones de función anónima
* Conversiones de grupo de métodos

Las conversiones implícitas pueden ocurrir en distintas situaciones, incluyendo las invocaciones de miembros de función (§7.5.4), las expresiones de conversión (§7.7.6) y las asignaciones (§7.17).

Las conversiones implícitas predefinidas siempre tienen éxito y nunca causan excepciones. Las conversiones implícitas definidas por el usuario correctamente diseñadas también deben exhibir estas características.

Para fines de conversión, los tipos object y dynamic se consideran equivalentes.

Sin embargo, las conversiones dinámicas (§6.1.8 y §6.2.6) se aplican solo a expresiones de tipo dynamic (§4.7).

### Conversiones de identidad

Una conversión de identidad convierte de cualquier tipo al mismo tipo. Esta conversión existe para que una entidad que ya sea de un tipo requerido pueda considerarse convertible en dicho tipo.

Como object y dynamic se consideran equivalentes, hay una conversión de identidad entre object y dynamic, y entre los tipos construidos que son iguales cuando reemplazan las apariciones de dynamic por object.

In most cases an identity conversion has no effect at runtime. However, since floating point operations may be performed at higher precision than prescribed by their type (§4.1.6), assignment of their results may result in a loss of precision, and explicit casts are guaranteed to reduce precision to what is prescribed by the type.

### Conversiones numéricas implícitas

Las conversiones implícitas numéricas son:

* De sbyte en short, int, long, float, double o decimal.
* De byte en short, ushort, int, uint, long, ulong, float, double o decimal.
* De short en int, long, float, double o decimal.
* De ushort en int, uint, long, ulong, float, double o decimal.
* De int en long, float, double o decimal .
* De uint en long, ulong, float, double o decimal.
* De long en float, double o decimal.
* De ulong en float, double o decimal.
* De char en ushort, int, uint, long, ulong, float, double o decimal.
* De float en double.

Las conversiones de int, uint, long o ulong en float y de long o ulong en double pueden producir una pérdida de precisión, pero nunca una pérdida de magnitud. En las demás conversiones implícitas numéricas nunca se pierde información.

No hay conversiones implícitas al tipo char, por tanto los valores de otros tipos enteros no se convierten automáticamente al tipo char.

### Conversiones de enumeración implícitas

Una conversión de enumeración implícita permite convertir el literal entero decimal (decimal-integer-literal) 0 (o 0L etc.) al tipo enum (enum-type) y a cualquier tipo que acepte valores NULL (nullable-type) cuyo tipo subyacente sea enum (enum-type). En este último caso, la conversión se evalúa mediante la conversión al tipo enum (enum-type) subyacente y el ajuste del resultado (§4.1.10).

### Conversiones implícitas que aceptan valores NULL

Las conversiones implícitas predefinidas que funcionan en tipos de valores que no aceptan valores NULL también se pueden utilizar con formas que aceptan valores NULL de dichos tipos. Para cada una de las conversiones numéricas y de identidad implícitas que se convierten de un tipo de valor S que no acepta valores NULL a un tipo de valor T que no acepta valores NULL, existen las siguientes conversiones implícitas que aceptan valores NULL:

* Una conversión implícita de S? a T?.
* Una conversión implícita de S a T?.

La evaluación de una conversión implícita que acepta valores NULL basada en una conversión subyacente de S a T se realiza de la siguiente manera:

* Si la conversión que acepta valores NULL se realiza de S? a T?:
* Si el valor de origen es NULL (la propiedad HasValue es falsa), el resultado es el valor NULL del tipo T?.
* De lo contrario, la conversión se evalúa como un desajuste de S? a S, seguido de la conversión subyacente de S a T, seguida de un ajuste (§4.1.10) de T a T?.
* Si la conversión que acepta valores NULL se realiza de S a T?, la conversión se evalúa como la conversión subyacente de S a T seguida de un ajuste de T a T?.

### Conversiones del literal NULL

Existe una conversión implícita desde el literal null a cualquier tipo que acepte valores NULL. Esta conversión genera el valor NULL (§4.1.10) del tipo que acepte valores NULL.

### Conversiones de referencia implícitas

Las conversiones implícitas de referencia son:

* De cualquier tipo de referencia (reference-type) en object y dynamic.
* De cualquier tipo de clase (class-type) S en cualquier tipo de clase T, a condición de que S se derive de T.
* De cualquier tipo de clase (class-type) S en cualquier tipo de interfaz (interface-type) T, a condición de que S implemente T.
* De cualquier tipo de interfaz (interface-type) S a cualquier tipo de interfaz T, a condición de que S se derive de T.
* De un tipo de matriz (array-type) S con un tipo de elemento SE a un tipo de matriz T con un tipo de elemento TE, siempre que se cumpla todo lo siguiente:
* S y T difieren solamente en el tipo de elemento. Esto es, S y T tienen el mismo número de dimensiones.
* Tanto SE como TE son tipos de referencia (reference-type).
* Existe una conversión implícita de referencia de SE a TE.
* De cualquier tipo de matriz (array-type) en System.Array y las interfaces que implementa.
* Desde un tipo de matriz unidimensional S[] a System.Collections.Generic.IList<T> y sus interfaces base, siempre que haya una identidad implícita o una conversión de referencia de S a T.
* De cualquier tipo delegado (delegate-type) en System.Delegate y las interfaces que implementa.
* Del literal nulo a cualquier tipo de referencia (reference-type).
* De cualquier tipo de referencia (reference-type) en un reference-type T si tiene una identidad implícita o conversión de referencia en un reference-type T0 y T0 tiene una conversión de identidad en T.
* De cualquier tipo de referencia (reference-type) en un tipo de delegado o de interfaz T si tiene una identidad implícita o conversión de referencia en un tipo de delegado o de interfaz T0 y T0 admite la conversión de varianza (§13.1.3.2) en T.
* Conversiones implícitas que impliquen parámetros de tipo que se sabe que son tipos de referencia. Vea la sección §6.1.10 para obtener más datos acerca de las conversiones implícitas que implican parámetros de tipo.

Las conversiones implícitas de referencia son aquellas entre tipos de referencia (reference-types) que se puede demostrar que siempre se realizan correctamente y, por lo tanto, no requieren comprobaciones en tiempo de ejecución.

Las conversiones de referencia, tanto implícitas como explícitas, nunca cambian la identidad referencial del objeto que se convierte. Es decir, si bien una conversión de referencia puede cambiar el tipo de la referencia, nunca cambia el tipo o el valor del objeto al que se refiere.

### Conversiones boxing

Una conversión boxing permite que un tipo de valor (value-type) se convierta implícitamente en un tipo de referencia. Existe una conversión boxing de cualquier tipo de valor que no acepta valores NULL (non-nullable-value-type) en object y dynamic, en System.ValueType y en cualquier tipo de interfaz (interface-type) implementado por el tipo de valor que no acepta valores NULL (non-nullable-value-type). Es más, un tipo enum (enum-type) se puede convertir al tipo System.Enum.

Existe una conversión boxing de cualquier tipo de valor que acepta valores NULL (nullable-value-type) a un tipo de referencia, si y sólo si existe una conversión boxing del tipo de valor que no acepta valores NULL (non-nullable-value-type) subyacente al tipo de referencia.

Un tipo de valor tiene una conversión boxing en un tipo de interfaz I si tiene una conversión boxing en un tipo de interfaz I0 e I0 tiene una identidad de conversión en I.

Un tipo de valor tiene una conversión boxing en un tipo de interfaz I si tiene una conversión boxing en un tipo de interfaz o de delegado I0 e I0 es de varianza convertible (§13.1.3.2) en I.

La conversión boxing de un tipo de valor que no acepta valores NULL (non-nullable-value-type) consiste en asignar una instancia del objeto y después copiar el tipo de valor (value-type) en esa instancia. Se puede aplicar a un struct la conversión boxing al tipo System.ValueType, puesto que ésa es una clase base para todas las structs (§11.3.2).

Una conversión boxing desde un tipo que acepta valores NULL (nullable-type) se procesa de la siguiente manera:

* Si el valor de origen es NULL (la propiedad HasValue es falsa), el resultado es una referencia NULL del tipo de destino.
* De lo contrario, el resultado es una referencia a un tipo T producido por el desajuste y la conversión boxing del valor de origen.

Las conversiones boxing se describen más detalladamente en §4.3.1.

### Conversiones dinámicas implícitas

Existe una conversión dinámica implícita de una expresión de tipo dynamic en cualquier tipo T. La conversión está enlazada dinámicamente (§7.2.2), lo que significa que se buscará una conversión implícita en tiempo de ejecución del tipo en tiempo de ejecución de la expresión en T. Si no se encuentra una conversión, se produce una excepción en tiempo de ejecución,

Observe que esta conversión implícita incumple el consejo dado al comienzo de §6.1 sobre que una conversión implícita nunca debería provocar una excepción. Sin embargo, no es la conversión en sí misma, sino buscar la conversión lo que causa la excepción. El riesgo de que se produzcan excepciones en tiempo de ejecución es inherente al uso del enlace dinámico. Si no se desea usar el enlace dinámico de la conversión, se puede convertir la expresión primero en object y después en el tipo deseado.

En el siguiente ejemplo se muestran conversiones dinámicas implícitas:

object o = “object”  
dynamic d = “dynamic”;

string s1 = o; // Fails at compile-time – no conversion exists  
string s2 = d; // Compiles and succeeds at run-time  
int i = d; // Compiles but fails at run-time – no conversion exists

Las asignaciones en s2 e i emplean ambas conversiones dinámicas implícitas, donde el enlace de las operaciones se suspende hasta el tiempo de ejecución. En tiempo de ejecución, se buscan las conversiones implícitas desde el tipo de tiempo de ejecución de d – string – en el tipo de destino. Se encuentra una conversión en string pero no en int.

### Conversiones implícitas de expresión constante

Una conversión implícita de expresión constante permite las siguientes conversiones:

* Una expresión constante (constant-expression) (§7.19) de tipo int se puede convertir en el tipo sbyte, byte, short, ushort, uint o ulong, siempre que el valor de la expresión constante (constant-expression) quede dentro del intervalo del tipo de destino.
* Una expresión constante (constant-expression) de tipo long se puede convertir al tipo ulong, a condición de que el valor de la expresión constante no sea negativo.

### Conversiones implícitas con parámetros de tipo

Para un parámetro de tipo T dado, existen las siguientes conversiones implícitas:

* Desde T a su clase base efectiva C, desde T a cualquier clase base C y desde T a cualquier interfaz implementada por C. En tiempo de ejecución, si T es un tipo de valor, la conversión se ejecuta como una conversión boxing. De lo contrario, la conversión se ejecuta como una conversión de referencia implícita o una conversión de identidad.
* De T a un tipo de interfaz I en el conjunto de interfaces efectivas de T, y de T a cualquier interfaz base de I. En tiempo de ejecución, si T es un tipo de valor, la conversión se ejecuta como una conversión boxing. De lo contrario, la conversión se ejecuta como una conversión de referencia implícita o una conversión de identidad.
* De T en un parámetro de tipo U, siempre que T dependa de U (§10.1.5). En tiempo de ejecución, si U es un tipo de valor, entonces T y U son necesariamente el mismo tipo y no se realiza una conversión boxing. De lo contrario, si T es un tipo de valor, la conversión se ejecuta como una conversión boxing. De lo contrario, la conversión se ejecuta como una conversión de referencia implícita o una conversión de identidad.
* Desde el literal nulo a T, siempre y cuando T sea un tipo de referencia.
* De T en un tipo de referencia I si tiene conversión implícita en un tipo de referencia S0 y S0 tiene una conversión de identidad en S. En tiempo de ejecución, la conversión se ejecuta igual que la conversión en S0.
* De T en un tipo de interfaz I si tiene una conversión implícita a un tipo de delegado o de interfaz I0 e I0 admite la conversión de varianza en I (§13.1.3.2). En tiempo de ejecución, si T es un tipo de valor, la conversión se ejecuta como una conversión boxing. De lo contrario, la conversión se ejecuta como una conversión de referencia implícita o una conversión de identidad.

Si T es un tipo de referencia (§10.1.5), todas las conversiones anteriores se clasifican como conversiones de referencia implícitas (§6.1.6). Si T no es un tipo de referencia, las conversiones anteriores se clasifican como conversiones boxing (§6.1.7).

### Conversiones definidas por el usuario implícitas

Una conversión implícita definida por el usuario consta de una conversión implícita opcional estándar, seguida por la ejecución de un operador de conversión implícita definido por el usuario, seguido por otra conversión implícita opcional estándar. Las reglas exactas para la evaluación de conversiones implícitas definidas por el usuario se describen en §6.4.4.

### Conversiones de función anónima y conversiones de grupo de métodos

Las funciones anónimas y los grupos de métodos no tienen tipos de por sí, pero se pueden convertir implícitamente en tipos delegados o tipos de árbol de expresiones. Las conversiones de función anónima se describen en más detalle en la sección §6.5 y las conversiones de grupo de métodos en §6.6.

## Conversiones explícitas

Las siguientes conversiones se clasifican como conversiones explícitas:

* Todas las conversiones implícitas.
* Conversiones explícitas numéricas.
* Conversiones explícitas de enumeración.
* Conversiones explícitas que aceptan valores NULL.
* Conversiones explícitas de referencia.
* Conversiones explícitas de interfaz.
* Conversiones Unboxing.
* Conversiones dinámicas explícitas
* Conversiones explícitas definidas por el usuario.

Las conversiones explícitas pueden producirse en las expresiones de conversión de tipos (§7.7.6).

El conjunto de conversiones explícitas incluye todas las conversiones implícitas. Esto quiere decir que están permitidas las expresiones de conversión de tipos redundantes.

Las conversiones explícitas son conversiones que no se puede demostrar que siempre se realizan correctamente, conversiones en las que se sabe que se puede producir pérdida de información y conversiones en dominios de tipos que, por sus diferencias, merecen una mención explícita.

### Conversiones explícitas numéricas

Las conversiones numéricas explícitas son las conversiones de un tipo numérico (numeric-type) en otro tipo numérico para el que no existe ya una conversión de tipo numérico implícita (§6.1.2):

* De sbyte en byte, ushort, uint, ulong o char.
* De byte en sbyte y char.
* De short en sbyte, byte, ushort, uint, ulong o char.
* De ushort en sbyte, byte, short o char .
* De int en sbyte, byte, short, ushort, uint, ulong o char.
* De uint en sbyte, byte, short, ushort, int o char.
* De long en sbyte, byte, short, ushort, int, uint, ulong o char.
* De ulong en sbyte, byte, short, ushort, int, uint, long o char.
* De char en sbyte, byte o short.
* De float en sbyte, byte, short, ushort, int, uint, long, ulong, char o decimal.
* De double en sbyte, byte, short, ushort, int, uint, long, ulong, char, float o decimal.
* De decimal en sbyte, byte, short, ushort, int, uint, long, ulong, char, float o double.

Dado que las conversiones explícitas incluyen todas las conversiones numéricas implícitas y explícitas, siempre es posible convertir de cualquier tipo numérico (numeric-type) a cualquier otro tipo numérico mediante una expresión de conversión de tipos (§7.7.6).

Las conversiones explícitas numéricas pueden producir pérdida de información o incluso provocar excepciones. Una conversión explícita numérica se procesa como se explica a continuación:

* Para una conversión de un tipo entero a otro tipo entero, el procesamiento depende del contexto de comprobación de desbordamiento (§7.6.12) en el que tiene lugar la conversión:
* En un contexto checked, la conversión termina correctamente si el valor del operando de origen queda dentro del intervalo del tipo de destino, pero provoca una excepción System.OverflowException si queda fuera de dicho intervalo.
* En un contexto unchecked, la conversión siempre termina correctamente y se procesa como sigue.
* Si el tipo de origen es mayor que el de destino, el valor de origen se trunca descartando sus bits “extra” más significativos. Después el resultado se trata como un valor del tipo de destino.
* Si el tipo de origen es menor que el tipo de destino, el valor de origen se amplía con un signo o con ceros, de forma que tenga el mismo tamaño que el tipo de destino. La ampliación con signo se utiliza si el tipo de origen tiene signo; se utiliza la ampliación con ceros si el tipo de origen no lleva signo. Después el resultado se trata como un valor del tipo de destino.
* Si el tipo del origen tiene el mismo tamaño que el tipo de destino, el valor de origen se trata como un valor del tipo de destino.
* Cuando se convierte un valor de tipo decimal a un tipo entero, el valor de origen se redondea hacia cero hasta el valor entero más próximo, y este valor integral pasa a ser el resultado de la conversión. Si el valor entero resultante queda fuera del intervalo del tipo de destino, se produce una excepción System.OverflowException.
* Para la conversión de un tipo float o double a otro tipo entero, el procesamiento depende del contexto de comprobación de desbordamiento (§7.6.12) en el que tiene lugar la conversión:
* En un contexto checked, la conversión sigue este procedimiento:
* Si el valor del operando es NaN o infinito, se produce una excepción System.OverflowException.
* De lo contrario, el operando de origen se redondea hacia cero, hasta el valor integral más cercano. Si dicho valor se encuentra dentro del intervalo del tipo de destino, pasará a ser el resultado de la conversión.
* De lo contrario, se inicia una excepción System.OverflowException.
* En un contexto unchecked, la conversión siempre termina correctamente y se procesa como sigue.
* Si el valor del operando es NaN o infinito, el resultado de la conversión es un valor no especificado del tipo de destino.
* De lo contrario, el operando de origen se redondea hacia cero, hasta el valor integral más cercano. Si dicho valor se encuentra dentro del intervalo del tipo de destino, pasará a ser el resultado de la conversión.
* De no ser así, el resultado de la conversión será un valor no especificado del tipo de destino.
* Para una conversión de double a float, el valor double se redondea al valor float más próximo. Si el valor double es demasiado pequeño para representarlo como float, el resultado se convierte en cero positivo o cero negativo. Si el valor double es demasiado grande para representarlo como float, el resultado se convierte en infinito positivo o infinito negativo. Si el valor double es NaN, el resultado también es NaN.
* Para una conversión de float o double a decimal, el valor de origen se convierte a la representación decimal y se redondea hasta el número más próximo después de la posición decimal 28a si es necesario (§4.1.7). Si el valor de origen es demasiado pequeño para representarlo como decimal, el resultado es cero. Si el valor de origen es NaN, infinito o demasiado grande para representarlo como decimal, se produce una excepción System.OverflowException.
* Para una conversión de decimal a float o double, el valor decimal se redondea al valor double o float más próximo. Aunque esta conversión puede perder precisión, nunca produce una excepción.

### Conversiones de enumeración explícitas

Las conversiones explícitas de enumeración son:

* De sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double o decimal en cualquier tipo enum (enum-type).
* De cualquier tipo enum (enum-type) en sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double o decimal.
* De un tipo enum (enum-type) a cualquier otro tipo enum.

Una conversión explícita de enumeración entre dos tipos se procesa tratando a uno de los tipos enum (enum-type) como tipo subyacente del otro, y realizando, a continuación, una conversión numérica implícita o explícita entre los tipos resultantes. Por ejemplo, dado un tipo enum (enum-type) E con un tipo subyacente de int, una conversión de E a byte se trata como una conversión numérica explícita (§6.2.1) de int a byte, y una conversión de byte a E se procesa como una conversión implícita numérica (§6.1.2) de byte a int.

### Conversiones explícitas que aceptan valores NULL

Las conversiones explícitas que aceptan valores NULL permiten predefinir conversiones explícitas que funcionan en tipos de valor que no aceptan valores NULL para que sean utilizados también con estructuras que aceptan valores NULL de dichos tipos. Para cada una de las conversiones predefinidas que se convierten de un tipo de valor S que no acepta valores NULL a un tipo de valor T que no acepta valores NULL (§6.1.1, §6.1.2, §6.1.3, §6.2.1 y §6.2.2), existen las siguientes conversiones que aceptan valores NULL:

* Una conversión explícita de S? a T?.
* Una conversión explícita de S a T?.
* Una conversión explícita de S? a T.

La evaluación de una conversión que acepta valores NULL se basa en una conversión subyacente de S a T se realiza de la siguiente manera:

* Si la conversión que acepta valores NULL se realiza de S? a T?:
* Si el valor de origen es NULL (la propiedad HasValue es falsa), el resultado es el valor NULL del tipo T?.
* De lo contrario, la conversión se evalúa como un desajuste de S? a S, seguido de la conversión subyacente de S a T, seguida de un ajuste de T a T?.
* Si la conversión que acepta valores NULL se realiza de S a T?, la conversión se evalúa como la conversión subyacente de S a T seguida de un ajuste de T a T?.
* Si la conversión que acepta valores NULL se realiza de S? a T, la conversión se evalúa como un desajuste de S? a S seguido de la conversión subyacente de S a T.

Tenga en cuenta que cualquier intento por desajustar un valor que acepta valores NULL producirá una excepción si el valor es null.

### Conversiones explícitas de referencia

Las conversiones explícitas de referencia son:

* De object y dynamic a cualquier otro tipo de referencia (reference-type).
* De cualquier tipo de clase (class-type) S a cualquier tipo de clase (class-type) T, siempre que S sea una clase base de T.
* De cualquier tipo de clase (class-type) S a cualquier tipo de interfaz (interface-type) T, siempre que S no sea sealed y que S no implemente T.
* De cualquier tipo de interfaz (interface-type) S a cualquier tipo de clase (class-type) T, siempre que T no sea sealed ni implemente a S.
* De cualquier tipo de interfaz (interface-type) S a cualquier tipo de interfaz T, a condición de que S no se derive de T.
* De un tipo de matriz (array-type) S con un tipo de elemento SE a un tipo de matriz T con un tipo de elemento TE, siempre que se cumpla todo lo siguiente:
* S y T difieren solamente en el tipo de elemento. Esto es, S y T tienen el mismo número de dimensiones.
* Tanto SE como TE son tipos de referencia (reference-type).
* Existe una conversión explícita de referencia de SE a TE.
* De System.Array y las interfaces que implementa a cualquier tipo de matriz (array-type).
* Desde un tipo de matriz unidimensional S[] a System.Collections.Generic.IList<T> y sus interfaces base, siempre que haya una identidad explícita o una conversión de referencia de S a T.
* Desde System.Collections.Generic.IList<S> y sus interfaces base a un tipo de matriz unidimensional T[], siempre que haya una identidad explícita o una conversión de referencia de S a T.
* De System.Delegate y las interfaces que implementa a cualquier tipo delegado (delegate-type).
* De un tipo de referencia en un tipo de referencia T si tiene conversión de referencia explícita en un tipo de referencia T0 y T0 tiene una conversión de identidad en T.
* De un tipo de referencia en un tipo de delegado o de interfaz T si tiene una conversión de referencia explícita en un tipo de delegado o de interfaz T0 y T0 es de varianza convertible en T o T es de varianza convertible en T0 (§13.1.3.2).
* De D<S1…Sn> a D<T1…Tn>, donde D<X1…Xn> es un tipo de delegado genérico, D<S1…Sn> no es compatible con o idéntico a D<T1…Tn>, y para cada parámetro de tipo Xi de D se cumple una de las condiciones siguientes:
* Si Xi es invariante, Si es idéntico a Ti.
* Si Xi es covariante, entonces existe una identidad implícita o explícita o una conversión de referencia de Si en Ti.
* Si Xi es contravariante, Si y Ti son idénticos o ambos son tipos de referencia.
* Conversiones explícitas que impliquen parámetros de tipo que se sabe que son tipos de referencia. Para obtener más datos acerca de las conversiones explícitas que implican parámetros de tipo, vea la sección §6.2.7.

Las conversiones explícitas de referencia son aquellas conversiones entre tipos de referencias que requieren comprobaciones en tiempo de ejecución para asegurar que son correctas.

Para que una conversión explícita de referencia se realice correctamente en tiempo de ejecución, el valor del operando de origen debe ser una referencia null, o el tipo real del objeto al que hace referencia el operando de origen debe ser un tipo convertible al tipo de destino mediante una conversión implícita de referencia (§6.1.6) o conversión boxing (§6.1.7). Si se produce un error en una conversión de referencia explícita, se genera una excepción System.InvalidCastException.

Las conversiones de referencia, tanto implícitas como explícitas, nunca cambian la identidad referencial del objeto que se convierte. Es decir, si bien una conversión de referencia puede cambiar el tipo de la referencia, nunca cambia el tipo o el valor del objeto al que se refiere.

### Conversiones Unboxing

Una conversión unboxing permite que un tipo de referencia (reference-type) se convierta explícitamente en un tipo de valor (value-type). Existe una conversión unboxing de los tipos object, dynamic y System.ValueType en cualquier tipo de valor que no acepta valores NULL (non-nullable-value-type), y de cualquier tipo de interfaz (interface-type) en cualquier tipo de valor que no acepta valores NULL (non-nullable-value-type) que implementa el tipo de interfaz (interface-type). Es más, se puede aplicar una conversión unboxing en un tipo System.Enum para convertirlo en un tipo enum (enum-type).

Existe una conversión unboxing de cualquier tipo de referencia a un tipo que acepta valores NULL (nullable-type), si existe una conversión unboxing del tipo de referencia al tipo de valor que no acepta valores NULL (non-nullable-value-type) subyacente del tipo que acepta valores NULL (nullable-type).

Un tipo de valor S tiene una conversión unboxing de un tipo de interfaz I si tiene una conversión unboxing de un tipo de interfaz I0 e I0 tiene una identidad de conversión en I.

Un tipo de valor S tiene una conversión unboxing de un tipo de interfaz I si tiene una conversión unboxing de un tipo de delegado o de interfaz I0 y bien I0 es de varianza convertible en I o I es de varianza convertible en I0 (§13.1.3.2).

Una operación unboxing consiste en comprobar primero que la instancia del objeto es un valor al que se ha aplicado la conversión boxing del tipo de valor (value-type) dado, y copiar después el valor fuera de la instancia. Aplicar la conversión unboxing a una referencia NULL para que sea un tipo que admite valores NULL (nullable-type) genera el valor NULL del tipo de acepta valores NULL. Se puede aplicar a un struct la conversión unboxing desde el tipo System.ValueType, porque ésa es una clase base para todas los structs (§11.3.2).

Las conversiones unboxing se explican más detalladamente en in §4.3.2.

### Conversiones dinámicas explícitas

Existe una conversión dinámica explícita de una expresión de tipo dynamic en cualquier tipo T. La conversión está enlazada dinámicamente (§7.2.2), lo que significa que se buscará una conversión explícita en tiempo de ejecución del tipo en tiempo de ejecución de la expresión en T. Si no se encuentra una conversión, se produce una excepción en tiempo de ejecución,

Si no se desea usar el enlace dinámico de la conversión, se puede convertir la expresión primero en object y después en el tipo deseado.

Supongamos que se define la clase siguiente:

class C  
{  
 int i;

public C(int i) { this.i = i; }

public static explicit operator C(string s)   
 {  
 return new C(int.Parse(s));  
 }  
}

En el siguiente ejemplo se muestran conversiones dinámicas explícitas:

object o = "1";  
dynamic d = "2";

var c1 = (C)o; // Compiles, but explicit reference conversion fails  
var c2 = (C)d; // Compiles and user defined conversion succeeds

La mejor conversión de o en C en tiempo de compilación es una conversión de referencia explícita. Esto da error en tiempo de ejecución porque “1” no es en realidad C. La conversión de d en C sin embargo, como conversión dinámica explícita, se suspende en tiempo de ejecución, donde se encuentra una conversión definida por el usuario del tipo en tiempo de ejecución de d – string – en C y se ejecuta correctamente.

### Conversiones explícitas con parámetros de tipo

Para un parámetro de tipo T dado, existen las siguientes conversiones explícitas:

* Desde la clase base efectiva C de T a T y desde cualquier clase base de C a T. En tiempo de ejecución, si T es un tipo de valor, la conversión se ejecuta como una conversión unboxing. De lo contrario, la conversión se ejecuta como una conversión de referencia explícita o una conversión de identidad.
* Desde cualquier tipo de referencia a T. En tiempo de ejecución, si T es un tipo de valor, la conversión se ejecuta como una conversión unboxing. De lo contrario, la conversión se ejecuta como una conversión de referencia explícita o una conversión de identidad.
* Desde T a cualquier de tipo de interfaz (interface-type) I siempre y cuando no exista ya una conversión implícita de T a I. En tiempo de ejecución, si T es un tipo de valor, la conversión se ejecuta como una conversión boxing seguida de una conversión de referencia explícita. De lo contrario, la conversión se ejecuta como una conversión de referencia explícita o una conversión de identidad.
* De un parámetro de tipo U a T, siempre que T dependa de U (§10.1.5). En tiempo de ejecución, si U es un tipo de valor, entonces T y U son necesariamente el mismo tipo y no se realiza una conversión boxing. De lo contrario, si T es un tipo de valor, la conversión se ejecuta como una conversión unboxing. De lo contrario, la conversión se ejecuta como una conversión de referencia explícita o una conversión de identidad.

Si T es un tipo de referencia, las conversiones anteriores se clasifican como conversiones de referencia explícitas (§6.2.4). Si T no es un tipo de referencia, las conversiones anteriores se clasifican como conversiones unboxing (§6.2.5).

Las reglas anteriores no permiten una conversión explícita directa desde un parámetro de tipo sin restricciones a un tipo de interfaz, lo que puede resultar sorprendente. El motivo es evitar la confusión y hacer más fácil la semántica de dichas conversiones. Por ejemplo, en la siguiente declaración:

class X<T>  
{  
 public static long F(T t) {  
 return (long)t; // Error   
 }  
}

Si se permitiese la conversión explícita directa de t a int, el resultado esperado de X<int>.F(7) sería 7L. Sin embargo, no es así porque las conversiones numéricas estándar sólo se consideran cuando los tipos son numéricos en tiempo de compilación. Para que la semántica sea más clara, el ejemplo anterior se debe escribir de la siguiente manera:

class X<T>  
{  
 public static long F(T t) {  
 return (long)(object)t; // Ok, but will only work when T is long  
 }  
}

Este código se compilará, pero si se ejecuta X<int>.F(7) se producirá una excepción en tiempo de ejecución, puesto que int no se puede convertir directamente en long.

### Conversiones explícitas definidas por el usuario

Una conversión explícita definida por el usuario consta de una conversión explícita opcional estándar, seguida por la ejecución de un operador de conversión implícita o explícita definida por el usuario, seguido por otra conversión explícita opcional estándar. Las reglas exactas para la evaluación de conversiones explícitas definidas por el usuario se describen en §6.4.5.

## Conversiones estándar

Las conversiones estándar son conversiones predefinidas que pueden ocurrir como parte de una conversión definida por el usuario.

### Conversiones implícitas estándar

Las siguientes conversiones implícitas se clasifican como conversiones implícitas estándar:

* Conversiones de identidad (§6.1.1)
* Conversiones numéricas implícitas (§6.1.2)
* Conversiones implícitas que aceptan valores NULL (§6.1.4)
* Conversiones del literal NULL (§6.1.5)
* Conversiones implícitas de referencia (§6.1.6)
* Conversiones boxing (§6.1.7)
* Conversiones implícitas de expresión constante (§6.1.8)
* Conversiones implícitas con parámetros de tipo (§6.1.10)

Las conversiones implícitas estándar excluyen de forma específica las conversiones implícitas definidas por el usuario.

### Conversiones explícitas estándar

Las conversiones explícitas estándar son todas las conversiones implícitas estándar más el subconjunto de las conversiones explícitas para las cuales existe una conversión implícita estándar opuesta. Es decir, si existe una conversión implícita estándar de un tipo A a un tipo B, eso significa que existe una conversión explícita estándar del tipo A al tipo B y del tipo B al tipo A.

## Conversiones definidas por el usuario

C# permite la ampliación de las conversiones explícitas e implícitas predefinidas mediante conversiones definidas por el usuario. Las conversiones definidas por el usuario se introducen mediante la declaración de operadores de conversión (§10.10.3) en tipos de clase y struct.

### Conversiones permitidas definidas por el usuario

C# sólo permite la declaración de algunas conversiones definidas por el usuario. En concreto, no es posible redefinir una conversión explícita o implícita ya existente.

Para un tipo de origen S y de destino T dados, si S o T son tipos que aceptan valores NULL, S0 y T0 hacen referencia a sus tipos subyacentes, en caso contrario S0 y T0 son iguales que S y T respectivamente. Una clase o struct tiene permitido declarar una conversión de un tipo de origen S a un tipo de destino T solamente si son verdaderos todos los puntos siguientes:

* S0 y T0 son tipos diferentes.
* S0 o T0 es el tipo de clase o estructura en el que tiene lugar la declaración del operador.
* Ni S0 ni T0 es un tipo de interfaz (interface-type).
* Excluyendo las conversiones definidas por el usuario, no existe una conversión de S a T ni de T a S.

Las restricciones aplicables a las conversiones definidas por el usuario se explican en §10.10.3.

### Operadores de conversión de elevación

Dado un operador de conversión definida por el usuario que realiza conversiones de un tipo de valor que no acepta valores NULL S a un tipo de valor que no acepta valores NULL T, existe un operador de conversión de elevación que convierte de S? a T?. Este operador de conversión de elevación realiza un desajuste de S? a S seguido de una conversión definida por el usuario de S a T seguida de un ajuste de T a T?, con la excepción de que S? con valor NULL se convierte directamente en T? con valor NULL.

Un operador de conversión de elevación tiene la misma clasificación implícita o explícita que su operador de conversión definido por el usuario. El término “conversión definida por el usuario” se aplica al uso tanto de operadores de conversión definida por el usuario como a operadores de conversión de elevación.

### Evaluación de conversiones definidas por el usuario

Una conversión definida por el usuario convierte un valor de su tipo, denominado tipo de origen, a otro tipo, denominado tipo de destino. La evaluación de una conversión definida por el usuario se centra en descubrir el operador de conversión definida por el usuario más específico para los tipos de origen y de destino concretos. Esta determinación se divide en varios pasos:

* Buscar el conjunto de clases y structs a partir del cual se consideran los operadores de conversión definida por el usuario. Este conjunto consta del tipo de origen y sus clases base y el tipo de destino y sus clases base (con los supuestos implícitos de que sólo las clases y structs pueden declarar operadores definidos por el usuario y de que los tipos no de clase no tienen clases base). Para este procedimiento, si el tipo de origen o el de destino es un tipo que acepta valores NULL (nullable-type), se usa su tipo subyacente en su lugar.
* A partir del conjunto de tipos, determinar qué operadores de conversión definida por el usuario o de elevación son aplicables. Para que un operador de conversión sea aplicable, debe ser posible realizar una conversión estándar (§6.3) del tipo de origen al tipo de operando del operador, y debe ser posible realizar una conversión estándar del tipo del resultado del operador al tipo de destino.
* A partir del conjunto de operadores definidos por el usuario que puedan aplicarse, determinar qué operador es el más específico sin ninguna ambigüedad. En términos generales, el operador más específico es aquél cuyo tipo de operando es el “más próximo” al tipo de origen y cuyo tipo de resultado es el “más próximo” al tipo de destino. Se prefieren los operadores de conversión definida por el usuario a los operadores de conversión de elevación. En las próximas secciones se definen las reglas exactas para establecer el operador de conversión definido por el usuario más específico.

Una vez identificado un operador de conversión definido por el usuario más específico, la ejecución de la conversión definida por el usuario implica hasta tres pasos:

* Primero, si se requiere, realizar una conversión estándar del tipo de origen al tipo de operando del operador de conversión definida por el usuario o de elevación.
* Después, invocar al operador de conversión definida por el usuario o de elevación para que realice la conversión.
* Por último, si se requiere, realizar una conversión estándar del tipo del resultado del operador de conversión definida por el usuario o de elevación al tipo de destino.

La evaluación de una conversión definida por el usuario nunca necesita más de un operador de conversión definida por el usuario o de elevación. Esto es, una conversión del tipo S al tipo T nunca ejecuta en primer lugar una conversión definida por el usuario de S a X y después una conversión definida por el usuario de X a T.

En las próximas secciones se ofrecen las definiciones exactas de la evaluación de conversiones implícitas o explícitas definidas por el usuario. En las definiciones se usan los siguientes términos:

* Si existe una conversión implícita estándar (§6.3.1) de un tipo A a un tipo B, y si ni A ni B son tipos de interfaz (interface-types), entonces se dice que A está abarcado por B, y que B abarca a A.
* Si existe una conversión implícita estándar (§6.3.1) de una expresión E en un tipo B, y si ni B ni el tipo de E (si lo tiene) son tipos de interfaz (interface-type), entonces se dice que E está abarcado por B y que B abarca a E.
* El tipo que más abarca de un conjunto de tipos es aquel que abarca a todos los demás tipos del conjunto. Si ninguno de los tipos abarca a todos los demás, entonces el conjunto no tiene tipo que más abarca. En términos más intuitivos, el tipo que más abarca es el “mayor” del conjunto, el tipo al que pueden convertirse implícitamente todos los demás tipos.
* El tipo más abarcado de un conjunto de tipos es aquel al que abarcan todos los demás tipos del conjunto. Si ninguno de los tipos es abarcado por todos los demás, entonces el conjunto no tiene un tipo más abarcado. En términos más intuitivos, el tipo más abarcado es el “menor” del conjunto, aquél que puede convertirse implícitamente a todos los demás tipos.

### Conversiones explícitas definidas por el usuario

Una conversión implícita definida por el usuario de una expresión E en un tipo T se procesa como sigue:

* Determine los tipos S, S0 y T0.
* Si E tiene un tipo, supongamos que este es S.
* Si S o T son tipos que aceptan valores NULL, SU y TU serán sus tipos subyacentes; de lo contrario, SU y TU serán S y T respectivamente.
* Si SU o TU son parámetros de tipo, supongamos que S0 y T0 son sus clases base efectivas; de lo contrario, S0 y T0 serán SU y TU respectivamente.
* Se busca un conjunto de tipos, D, a partir del cual se consideran los operadores de conversión definida por el usuario. Este conjunto está formado por S0 (si S0 existe y es una clase o struct), las clases base de S0 (si S0 existe y es una clase) y T0 (si T0 es una clase o struct). Solamente se agrega un tipo al conjunto D si no existe una conversión de identidad a otro tipo ya incluido en dicho conjunto.
* Se busca el conjunto de operadores de conversión elevada definida por el usuario aplicables, U. Este conjunto consta de los operadores de conversión implícita elevada y definida por el usuario declarados por las clases o structs de D que convierten de un tipo incluyente E a un tipo abarcado por T. Si U está vacía, la conversión no estará definida y se producirá un error en tiempo de ejecución.
* Se busca el tipo de origen más específico, SX, de los operadores de U:
* Si S existe y cualquiera de los operadores de U se convierte desde S, entonces SX es S.
* De lo contrario, SX es el tipo más abarcado del conjunto combinado de tipos de origen de los operadores de U. Si no se puede encontrar exactamente el tipo más abarcado, la conversión es ambigua y se genera un error en tiempo de compilación.
* Se busca el tipo de destino más específico, TX, de los operadores de U:
* Si alguno de los operadores de U se convierte en T, entonces TX es T.
* De lo contrario, TX es el tipo más que más abarca del conjunto combinado de tipos de destino de los operadores de U. Si no se puede encontrar exactamente el tipo más abarcado, la conversión es ambigua y se genera un error en tiempo de compilación.
* Se buscar el operador de conversión más específico:
* Si U contiene exactamente un operador de conversión definida por el usuario que convierte de SX a TX, este es el operador de conversión más específico.
* De lo contrario, si U contiene exactamente un operador de conversión de elevación que realiza la conversión de SX a TX, este es el operador de conversión más específico.
* De lo contrario, la conversión es ambigua y se genera un error en tiempo de compilación.
* Por último, se aplica la conversión:
* Si E no tiene el tipo SX, se realiza una conversión implícita estándar de E a SX.
* El operador de conversión más específico se invoca para realizar la conversión de SX a TX.
* Si TX no es T, se realiza una conversión implícita estándar de TX a T.

Si ya existe una conversión implícita definida por el usuario de una variable de tipo S a T, existe una conversión implícita definida por el usuario de un tipo S a un tipo T.

### Conversiones explícitas definidas por el usuario

Una conversión explícita definida por el usuario de una expresión E en un tipo T se procesa como sigue:

* Determine los tipos S, S0 y T0.
* Si E tiene un tipo, supongamos que este es S.
* Si S o T son tipos que aceptan valores NULL, supongamos que SU y TU son sus tipos subyacentes; de lo contrario, SU y TU serán S y T respectivamente.
* Si SU o TU son parámetros de tipo, supongamos que S0 y T0 son sus clases base efectivas; de lo contrario, S0 y T0 serán SU y TU respectivamente.
* Se busca un conjunto de tipos, D, a partir del cual se consideran los operadores de conversión definida por el usuario. Este conjunto está formado por S0 (si S0 existe y es una clase o struct), las clases base de S0 (si S0 existe y es una clase), T0 (si T0 es una clase o struct) y las clases base de T0 (si T0 es una clase). Solamente se agrega un tipo al conjunto D si no existe una conversión de identidad a otro tipo ya incluido en dicho conjunto.
* Se busca el conjunto de operadores de conversión elevada definida por el usuario aplicables, U. Este conjunto consta de los operadores de conversión elevada implícita o explícita definidos por el usuario, declarados por las clases o structs de D que convierten de un tipo que abarca E o abarcado por S (si existe) a un tipo que abarca o es abarcado por T. Si U está vacía, la conversión no estará definida y se producirá un error en tiempo de ejecución.
* Se busca el tipo de origen más específico, SX, de los operadores de U:
* Si S existe y cualquiera de los operadores de U se convierte desde S, entonces SX es S.
* De lo contrario, si uno de los operadores de U convierte de los tipos que abarca E, entonces SX es el tipo más abarcado del conjunto combinado de tipos de origen de estos operadores. Si no se encuentra un tipo más abarcado, la conversión será ambigua y se producirá un error en tiempo de compilación.
* De lo contrario, SX es el tipo que más abarca del conjunto combinado de tipos de origen de los operadores de U. Si no se puede encontrar exactamente el tipo más abarcado, la conversión es ambigua y se genera un error en tiempo de compilación.
* Se busca el tipo de destino más específico, TX, de los operadores de U:
* Si alguno de los operadores de U se convierte en T, entonces TX es T.
* De lo contrario, si uno de los operadores de U convierte a los tipos abarcados por T, entonces TX es el tipo que más abarca del conjunto combinado de tipos de origen de estos operadores. Si no se puede encontrar exactamente el tipo más abarcado, la conversión es ambigua y se genera un error en tiempo de compilación.
* De lo contrario, TX es el tipo más abarcado del conjunto combinado de tipos de destino de los operadores de U. Si no se encuentra un tipo más abarcado, la conversión será ambigua y se producirá un error en tiempo de compilación.
* Se buscar el operador de conversión más específico:
* Si U contiene exactamente un operador de conversión definida por el usuario que convierte de SX a TX, este es el operador de conversión más específico.
* De lo contrario, si U contiene exactamente un operador de conversión de elevación que realiza la conversión de SX a TX, este es el operador de conversión más específico.
* De lo contrario, la conversión es ambigua y se genera un error en tiempo de compilación.
* Por último, se aplica la conversión:
* Si E no tiene el tipo SX, se realiza una conversión explícita estándar de E a SX.
* Se llama al operador de conversión más específico definido por el usuario para convertir de SX a TX.
* Si TX no es T, se realiza una conversión explícita estándar de TX a T.

Si ya existe una conversión explícita definida por el usuario de una variable de tipoS a T, existe una conversión explícita definida por el usuario de un tipo S a un tipo T.

## Conversiones de función anónima

Una expresión de método anónimo (anonymous-method-expression) o expresión lambda (lambda-expression) se clasifica como una función anónima (§7.15). La expresión no tiene un tipo, pero se puede convertir de manera implícita en un tipo de árbol de expresiones o un tipo delegado compatible. En concreto, una función anónima F es compatible con un tipo delegado D siempre que:

* Si F contiene una signatura de función anónima (anonymous-function-signature), D y F tienen el mismo número de parámetros.
* Si F no contiene una signatura de función anónima (anonymous-function-signature), D puede no tener ningún parámetro o tener parámetros de cualquier tipo siempre y cuando ninguno de ellos tenga el modificador de parámetros out.
* Si F tiene una lista de parámetros escritos explícitamente, cada parámetro de D tiene el mismo tipo y los mismos modificadores que el parámetro correspondiente en F.
* Si F tiene una lista de parámetros de tipo implícito, D no tiene ningún parámetro ref u out.
* Si el cuerpo de F es una expresión, y bien D tiene un tipo devuelto void o F es asincrónico y D tiene el tipo devuelto Task, cuando se da a cada parámetro de F el tipo del parámetro correspondiente en D, el cuerpo de F es una expresión válida (wrt §7) que se permitiría como expresión de instrucción (statement-expression) (§8.6).
* Si el cuerpo de F es un bloque de instrucciones, y bien D tiene un tipo devuelto void o F es asincrónico y D tiene el tipo devuelto Task, cuando se da a cada parámetro de F el tipo del parámetro correspondiente en D, el cuerpo de F es un bloque de instrucción válido (wrt §7) en el que ninguna instrucción return especifica una expresión.
* Si el cuerpo de F es una expresión y *bien* F no es asincrónico y D tiene un tipo devuelto T que no sea void *o* F es asincrónico y D tiene un tipo devuelto Task<T>, cuando se da a cada parámetro de F el tipo del parámetro correspondiente en D, el cuerpo de F es una expresión válida (wrt §7) que es implícitamente convertible en T.
* Si el cuerpo de F es un bloque de instrucciones, y *bien* F no es asincrónico y D tiene un tipo devuelto T que no es void, *o* F es asincrónico y D tiene un tipo devuelto Task<T>, cuando se da a cada parámetro de F el tipo del parámetro correspondiente en D, el cuerpo de F es un bloque de instrucción válido (wrt §8.2) con un punto final que no es alcanzable en el que cada instrucción return especifica una expresión que es implícitamente convertible a T.

Por brevedad, en esta sección se usa la forma corta de los tipos de tarea Task y Task<T> (§10.14).

Una expresión lambda F es compatible con un tipo de árbol de expresión Expression<D> si F es compatible con el tipo delegado D. Tenga en cuenta que esto no se aplica a los métodos anónimos, solamente a las expresiones lambda.

Algunas expresiones lambda no se pueden convertir a tipos de árbol de expresión: aunque la conversión existe, da error en tiempo de compilación. Este es el caso si la expresión lambda:

* Tiene cuerpo de bloque
* Contiene operadores de asignación simples o compuestos
* Contiene una expresión enlazada dinámicamente
* Es asincrónico

En el ejemplo siguiente se utiliza un tipo delegado genérico Func<A,R> que representa una función que toma un argumento del tipo A y devuelve un valor de tipo R:

delegate R Func<A,R>(A arg);

En las asignaciones

Func<int,int> f1 = x => x + 1; // Ok

Func<int,double> f2 = x => x + 1; // Ok

Func<double,int> f3 = x => x + 1; // Error

Func<int, Task<int>> f4 = async x => x + 1; // Ok

el parámetro y los tipos de valor devuelto de cada función anónima se determinan a partir del tipo de la variable a la que la función anónima se asigna.

La primera asignación convierte correctamente la función anónima en el tipo delegado Func<int,int> porque, cuando se da a x el tipo int, x + 1 es una expresión válida que es implícitamente convertible al tipo int.

Del mismo modo, la segunda asignación convierte correctamente la función anónima en el tipo delegado Func<int,double> porque el resultado de x + 1 (de tipo int) es implícitamente convertible al tipo double.

Sin embargo, la tercera asignación es un error en tiempo de compilación porque, cuando se da a x el tipo double, el resultado de x + 1 (de tipo double) no es implícitamente convertible al tipo int.

La cuarta asignación convierte correctamente la función asincrónica en el tipo delegado Func<int, Task<int>> porque el resultado de x + 1 (de tipo int) es implícitamente convertible al tipo de resultado int del tipo de tarea Task<int>.

Las funciones anónimas pueden influenciar la resolución de sobrecargas y participar en la inferencia de tipos. Consulte §7.5para obtener más detalles.

### Evaluación de conversiones de función anónima a tipos delegados

La conversión de una función anónima a un tipo delegado genera una instancia de delegado que hace referencia a la función anónima y al conjunto (posiblemente vacío) de variables externas capturadas y activas en el momento de la evaluación. Si se invoca el delegado, se ejecuta el cuerpo de la función anónima. El código del cuerpo se ejecuta con el conjunto de variables externas capturadas al que el delegado hace referencia.

La lista de invocación de un delegado producida desde una función anónima contiene una sola entrada. No se especifican el objeto y el método de destino exactos del delegado. En concreto, no se especifica si el objeto de destino del delegado es null, el valor this del miembro de función envolvente o algún otro objeto.

Las conversiones de funciones anónimas de semánticas idénticas con el mismo conjunto (posiblemente vacío) de instancias de variables externas capturadas a los mismos tipos delegados pueden devolver (aunque no es obligatorio) la misma instancia de delegado. En este contexto, semántica idéntica hace referencia a que la ejecución de funciones anónimas generará en todos los casos los mismos efectos si se dan los mismos argumentos. Esta regla permite optimizar código como el siguiente.

delegate double Function(double x);

class Test  
{  
 static double[] Apply(double[] a, Function f) {  
 double[] result = new double[a.Length];  
 for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);  
 return result;  
 }

static void F(double[] a, double[] b) {  
 a = Apply(a, (double x) => Math.Sin(x));  
 b = Apply(b, (double y) => Math.Sin(y));  
 ...  
 }  
}

Puesto que los dos delegados de función anónima tienen el mismo conjunto (vacío) de variables externas capturadas y puesto que las funciones anónimas son semánticamente idénticas, el compilador puede tener delegados que hagan referencia al mismo método de destino. De hecho, el compilador puede devolver la misma instancia de delegado de ambas expresiones de función anónima.

### Evaluación de conversiones de función anónima a tipos delegados de árbol de expresiones

La conversión de una función anónima a un tipo de árbol de expresiones produce un árbol de expresiones (§4.6). Más concretamente, la evaluación de la conversión de función anónima conduce a la construcción de una estructura de objetos que representa la estructura de la función anónima en sí. La estructura precisa del árbol de expresión, así como el proceso exacto para su creación, se definen en la implementación.

### Ejemplo de implementación

En esta sección se describe una posible implementación de conversiones de funciones anónimas en términos de otras construcciones del lenguaje C#. La implementación aquí descrita se basa en los mismos principios utilizados por el compilador de Microsoft C#, pero no es una implementación obligatoria ni tampoco la única posible. Sólo se mencionan brevemente los árboles de expresiones, ya que su semántica exacta está fuera del ámbito de este documento.

En el resto de esta sección se proporcionan varios ejemplos de código que incluye funciones anónimas con diferentes características. En cada uno de los ejemplos se proporciona la traducción correspondiente al código que sólo utiliza otras construcciones estándar del lenguaje C#. En los ejemplos, se supone que el identificador D representa el siguiente tipo de delegado:

public delegate void D();

La estructura más sencilla de una función anónima es la que no captura variables externas:

class Test  
{  
 static void F() {  
 D d = () => { Console.WriteLine("test"); };  
 }  
}

Esto se puede traducir en una creación de instancias de delegado que hace referencia a un método estático generado por compilador en el que se sitúa la función anónima:

class Test  
{  
 static void F() {  
 D d = new D(\_\_Method1);  
 }

static void \_\_Method1() {  
 Console.WriteLine("test");  
 }  
}

En el siguiente ejemplo, la función anónima hace referencia a miembros de instancia de this:

class Test  
{  
 int x;

void F() {  
 D d = () => { Console.WriteLine(x); };  
 }  
}

Se puede traducir en un método de instancia generado por compilador que contiene el código de la función anónima:

class Test  
{  
 int x;

void F() {  
 D d = new D(\_\_Method1);  
 }

void \_\_Method1() {  
 Console.WriteLine(x);  
 }  
}

En este ejemplo, la función anónima captura una variable local:

class Test  
{  
 void F() {  
 int y = 123;  
 D d = () => { Console.WriteLine(y); };  
 }  
}

Ahora, el período de duración de la variable local se debe extender al menos al período de duración del delegado de la función anónima. Esto se puede lograr “activando” la variable local en un campo de una clase generada por compilador. Entonces, la creación de instancias de la variable local (§7.15.5.2) corresponde a la creación de una instancia de la clase generada por compilador y el acceso a la variable local corresponde al acceso a un campo en la instancia de la clase generada por compilador. La función anónima se convierte en un método de instancias de la clase generada por compilador:

class Test  
{  
 void F() {  
 \_\_Locals1 \_\_locals1 = new \_\_Locals1();  
 \_\_locals1.y = 123;  
 D d = new D(\_\_locals1.\_\_Method1);  
 }

class \_\_Locals1  
 {  
 public int y;

public void \_\_Method1() {  
 Console.WriteLine(y);  
 }  
 }  
}

Finalmente, la siguiente función anónima captura this así como dos variables locales con diferentes períodos de duración:

class Test  
{  
 int x;

void F() {  
 int y = 123;  
 for (int i = 0; i < 10; i++) {  
 int z = i \* 2;  
 D d = () => { Console.WriteLine(x + y + z); };  
 }  
 }  
}

Aquí se crea una clase generada por compilador para cada bloque de instrucciones en los que se capturan valores locales para que los valores locales de bloques diferentes tengan períodos de duración independientes. Una instancia de \_\_Locals2, la clase generada por compilador para el bloque interior de instrucciones, contiene la variable local z y un campo que hace referencia a una instancia de \_\_Locals1. Una instancia de \_\_Locals1, la clase generada por compilador para el bloque exterior de instrucciones, contiene la variable local y y un campo que hace referencia a this del miembro de función envolvente. Con estas estructuras de datos es posible alcanzar todas las variables externas capturadas mediante una instancia de \_\_Local2 y el código de la función anónima se puede implementar como un método de una instancia de dicha clase.

class Test  
{  
 void F() {  
 \_\_Locals1 \_\_locals1 = new \_\_Locals1();  
 \_\_locals1.\_\_this = this;  
 \_\_locals1.y = 123;  
 for (int i = 0; i < 10; i++) {  
 \_\_Locals2 \_\_locals2 = new \_\_Locals2();  
 \_\_locals2.\_\_locals1 = \_\_locals1;  
 \_\_locals2.z = i \* 2;  
 D d = new D(\_\_locals2.\_\_Method1);  
 }  
 }

class \_\_Locals1  
 {  
 public Test \_\_this;  
 public int y;  
 }

class \_\_Locals2  
 {  
 public \_\_Locals1 \_\_locals1;  
 public int z;

public void \_\_Method1() {  
 Console.WriteLine(\_\_locals1.\_\_this.x + \_\_locals1.y + z);  
 }  
 }  
}

La misma técnica aplicada aquí para capturar variables locales también se puede utilizar a la hora de convertir funciones anónimas a árboles de expresiones: las referencias a los objetos generados por el compilador se pueden almacenar en el árbol de expresiones, y el acceso a las variables locales se puede representar como acceso a campos en esos objetos. La ventaja de este método es que permite que las variables locales de “elevación” sean compartidas entre árboles de expresiones y delegados.

## Conversiones de grupo de métodos

Existe una conversión implícita (§6.1) desde un grupo de métodos (§7.1) a un tipo delegado compatible. Dado un tipo delegado D y una expresión E clasificada como un grupo de métodos, existe una conversión implícita de E a D si E contiene al menos un método aplicable en su estructura normal (§7.5.3.1) a una lista de argumentos construidos por el uso de tipos de parámetro y modificadores de D, tal como se describe a continuación.

La aplicación en tiempo de compilación de una conversión desde un grupo de métodos E a un tipo delegado D se describe a continuación. Tenga en cuenta que la existencia de una conversión implícita de E a D no garantiza que la aplicación en tiempo de compilación de la conversión se realizará sin errores.

* Se selecciona un único método M, correspondiente a una invocación de método (§7.6.5.1) con la estructura E(A), con las siguientes modificaciones:
* La lista de argumentos A es una lista de expresiones, cada una clasificada como una variable y con el tipo y el modificador (ref o bien out) del parámetro correspondiente en la lista de parámetros formales (formal-parameter-list) de D, con la excepción de los parámetros de tipo dynamic, donde la expresión correspondiente tiene el tipo object en lugar de dynamic.
* Los métodos de candidatos considerados solo son los que se aplican en su estructura normal y no omiten ningún parámetro opcional (§7.5.3.1). Así, este tipo de métodos se omiten si se aplican solo en su estructura expandida o si uno o varios de sus parámetros opcionales no tiene un parámetro correspondiente en D.
* Se considera que existe una conversión si el algoritmo de §7.6.5.1 genera un método individual mejor M con el mismo número de parámetros que D.
* Aunque la conversión exista, se genera un error en tiempo de compilación si el método M seleccionado no es compatible (§15.2) con el tipo delegado D.
* Si el método seleccionado M es un método de instancia, la expresión de instancia asociada a E determina el objeto de destino del delegado.
* Si el método seleccionado M es un método de extensión que se denota mediante un acceso a miembros en una expresión de instancia, esa expresión de instancia determina el objeto de destino del delegado.
* El resultado de la conversión es un valor del tipo D, un delegado recién creado que hace referencia al método seleccionado y al objeto de destino.

Tenga en cuenta que este proceso puede conducir a la creación de un delegado para un método de extensión, si el algoritmo de §7.6.5.1 no logra encontrar un método de instancia pero sí logra culminar con éxito el procesamiento de la invocación de E(A) como una invocación del método de extensión (§7.6.5.2). Así, el delegado ha creado capturas del método de extensión así como su primer argumento.

En el siguiente ejemplo se demuestran las conversiones de grupo de métodos:

delegate string D1(object o);

delegate object D2(string s);

delegate object D3();

delegate string D4(object o, params object[] a);

delegate string D5(int i);

class Test  
{  
 static string F(object o) {...}

static void G() {  
 D1 d1 = F; // Ok  
 D2 d2 = F; // Ok  
 D3 d3 = F; // Error – not applicable  
 D4 d4 = F; // Error – not applicable in normal form  
 D5 d5 = F; // Error – applicable but not compatible

}  
}

La asignación a d1 convierte de manera implícita el grupo de métodos F en un valor del tipo D1.

La asignación a d2 muestra cómo es posible crear un delegado para un método que tiene menos parámetros derivados (contravariantes) y más tipos de valor devuelto (covariantes) derivados.

La asignación a d3 muestra cómo no existe ninguna conversión si el método no es aplicable.

La asignación a d4 muestra cómo el método debe ser aplicable en su estructura formal.

La asignación a d5 muestra la manera en que parámetros y tipos de valor devuelto del delegado y del método pueden diferenciarse solo por los tipos de referencia.

Al igual que ocurre con otras conversiones implícitas y explícitas, el operador de conversión se puede utilizar para realizar de manera explícita una conversión de grupo de métodos. Por lo tanto, el siguiente ejemplo:

object obj = new EventHandler(myDialog.OkClick);

se puede escribir de la siguiente manera:

object obj = (EventHandler)myDialog.OkClick;

Los grupos de método pueden influenciar la resolución de sobrecargas y participar en la inferencia de tipos. Consulte §7.5para obtener más detalles.

La evaluación en tiempo de ejecución de una conversión de grupo de métodos se realiza de la siguiente forma:

* Si el método seleccionado en tiempo de compilación es un método de instancia, o es un método de extensión al que se tiene acceso como método de instancia, el objeto de destino del delegado se determina a partir de la expresión de instancia asociada con E:
* Se evalúa la expresión de instancia. Si esta evaluación da lugar a una excepción, no se ejecutan nuevos pasos.
* Si la expresión de instancia es de un tipo de referencia (reference-type), el valor calculado por la expresión de instancia es el objeto de destino. Si el método seleccionado es un método de instancia y el objeto de destino es null, se produce una System.NullReferenceException y no se ejecutan más pasos.
* Si la expresión de instancia es de un tipo de valor (value-type), se realiza una operación boxing (§4.3.1) para convertir el valor en un objeto, que pasa a ser el objeto de destino.
* De lo contrario, el método seleccionado es parte de una llamada a método estático, y el objeto de destino del delegado es null.
* Se asigna una nueva instancia del tipo delegado D. Si no hay memoria disponible suficiente para asignar la nueva instancia, se produce una excepción System.OutOfMemoryException y no se ejecutan más pasos.
* Se inicializa la nueva instancia de delegado con una referencia al método que se determinó en tiempo de compilación y una referencia al objeto de destino antes calculado.

# Expresiones

Una expresión es una secuencia de operadores y operandos. En este capítulo se definen la sintaxis, el orden de evaluación de los operandos y los operadores, así como el significado de las expresiones.

## Clasificaciones de expresión

Una expresión se puede clasificar como:

* Un valor. Todos los valores tienen asociado un tipo.
* Una variable. Todas variables tienen un tipo asociado, esto es, el tipo declarado de la variable.
* Un espacio de nombres. Una expresión con su clasificación sólo puede aparecer como el miembro izquierdo de un acceso a miembro (member-access) (§7.6.4). En cualquier otro contexto, una expresión que se clasifica como un espacio de nombres produce un error en tiempo de compilación.
* Un tipo. Una expresión con esta clasificación sólo puede aparecer como el lado izquierdo de un acceso a miembro (member-access) (§7.6.4), o como un operando para el operador as (§7.10.11), el operador is (§7.10.10) o el operador typeof (§7.6.11). En cualquier otro contexto, una expresión que se clasifica como un tipo produce un error en tiempo de compilación.
* Un grupo de métodos, que es un conjunto de métodos sobrecargados producidos por una búsqueda de miembros (§7.4). Un grupo de métodos puede tener asociada una expresión de instancia y una lista de argumentos de tipo asociados. Cuando se invoca un método de instancia, el resultado de la evaluación de la expresión de instancia se convierte en la instancia representada por this (§7.6.7). Se permite un grupo de métodos en una expresión de invocación (invocation-expression) (§7.6.5), una expresión de creación de delegados (delegate-creation-expression) (§7.6.10.5) y como la parte izquierda de un operador is, y se puede convertir implícitamente en un tipo delegado compatible (§6.6). En cualquier otro contexto, una expresión que se clasifica como un grupo de métodos produce un error en tiempo de compilación.
* Un literal null. Una expresión con esta clasificación se puede convertir de manera implícita en un tipo de referencia o un tipo que acepta valores NULL.
* Una función anónima. Una expresión con esta clasificación se puede convertir de manera implícita en un tipo delegado compatible o un tipo de árbol de expresiones.
* Un acceso de propiedad. Todos los accesos de propiedad tienen un tipo asociado, esto es, el tipo declarado de la propiedad. Además, un acceso a propiedad puede tener asociada una expresión de instancia. Si se llama a un descriptor de acceso (el bloque get o set) de un acceso a propiedad de una instancia, el resultado de la evaluación de la expresión de instancia se convierte en la instancia representada por this (§7.6.7).
* Un acceso de evento. Todos los accesos de evento tienen un tipo asociado, el tipo declarado del evento. Además, un acceso de evento puede tener asociada una expresión de instancia. Un acceso a evento puede aparecer como el operando izquierdo de los operadores += y -= (§7.17.3). En cualquier otro contexto, una expresión que se clasifica como un acceso de evento produce un error en tiempo de compilación.
* Un acceso a indizador. Todos los accesos a indizadores tienen un tipo asociado, el tipo declarado del indizador. Además, un acceso a indizador tiene asociadas una expresión de instancia y una lista de argumentos. Si se llama a un descriptor de acceso (el bloque get o set) de un indizador, el resultado de la evaluación de la expresión de instancia pasa a ser la instancia representada por this (§7.6.7), y el resultado de la evaluación de la lista de argumentos se convierte en la lista de parámetros de la invocación.
* Nada. Esto ocurre cuando la expresión es una invocación de un método con el tipo de valor devuelto void. Una expresión clasificada como nada sólo es válida en el contexto de una expresión de instrucción (statement-expression) (§8.6).

El resultado final de una expresión nunca es un espacio de nombres, un tipo, un grupo de métodos o un acceso de evento. En lugar de ello, como se ha mencionado antes, estas categorías de expresiones son construcciones intermedias que sólo están permitidas en determinados contextos.

Un acceso a propiedad o un acceso a indizador siempre se reclasifican como un valor mediante una llamada del descriptor de acceso get (get-accessor) o del descriptor de acceso set (set-accessor). El descriptor de acceso concreto viene determinado por el contexto del acceso a propiedad o del acceso a indizador: si el acceso es el destino de una asignación, se llama al descriptor de acceso set (set-accessor) para asignar un nuevo valor (§7.17.1). En otro caso, se invoca el descriptor de acceso get (get-accessor) para obtener el valor actual (§7.1.1).

### Valores de expresiones

La mayoría de las construcciones que involucran una expresión requieren en definitiva que la expresión denote un valor. En estos casos, si la expresión real denota un espacio de nombres, un tipo, un grupo de métodos o nada, se produce un error en tiempo de compilación. No obstante, si la expresión denota un acceso a propiedad, un acceso a indizador o una variable, el valor de la propiedad, el indizador o la variable se sustituyen de forma implícita:

* El valor de una variable es sencillamente el valor almacenado en la ubicación de almacenamiento que ella misma identifica. Una variable debe considerarse asignada definitivamente (§5.3) para poder obtener su valor o, de lo contrario, se produce un error de tiempo de compilación.
* El valor de una expresión de acceso a propiedad se obtiene mediante una llamada al descriptor de acceso get (get-accessor) de la propiedad. Si la propiedad no tiene un descriptor de acceso get (get-accessor), se produce un error durante la compilación. En caso contrario, se realiza una llamada a un miembro de función (§7.5.4) y el resultado de la llamada pasa a ser el valor de la expresión de acceso a propiedad.
* El valor de una expresión de acceso a indizador se obtiene mediante una llamada al descriptor de acceso get (get-accessor) del indizador. Si el indizador no tiene un descriptor de acceso get (get-accessor), se produce un error durante la compilación. En caso contrario, se realiza una llamada a un miembro de función (§7.5.4) con la lista de argumentos asociada a la expresión de acceso al indizador, y el resultado de la llamada se convierte en el valor de la expresión de acceso al indizador.

## Enlace estático y dinámico

El proceso de determinar el significado de una operación basada en el tipo o valor de las expresiones constitutivas (argumentos, operandos, receptores) se suele conocer como enlace. Por ejemplo, el significado de una llamada a un método se determina según el tipo del receptor y los argumentos. El significado de un operador se determina según el tipo de sus operandos.

En C# el significado de una operación se determina por lo general en tiempo de compilación, según el tipo en tiempo de compilación de sus expresiones constitutivas. De igual modo, si una expresión contiene un error, el compilador detecta e informa sobre el error. Este enfoque se denomina enlace estático.

Sin embargo, si una expresión es una expresión dinámica (es decir, tiene el tipo dynamic) indica que todo enlace en el que participe debería basarse en su tipo en tiempo de ejecución (es decir, el tipo actual del objeto que denota en tiempo de ejecución) en lugar del tipo que tiene en tiempo de compilación. El enlace de una operación tal se retrasa hasta el momento en que se va a ejecutar el programa durante el funcionamiento del programa. Esto se denomina enlace dinámico.

Cuando una operación se enlaza dinámicamente, el compilador no hace apenas ninguna comprobación. En cambio, si el enlace en tiempo de ejecución da error, los errores se emiten como excepciones en tiempo de ejecución.

Las operaciones siguientes están sujetas a enlace en C#:

* Acceso a miembros: e.M
* Método de invocación: e.M(e1,…,en)
* Invocación de delegados: e(e1,…,en)
* Acceso a elementos: e[e1,…,en]
* Crear objetos: new C(e1,…,en)
* Operadores unarios sobrecargados: +, -, !, ~, ++, --, true, false
* Operadores binarios sobrecargados: +, -, \*, /, %, &, &&, |, ||, ??, ^, <<, >>, ==,!=, >, <, >=, <=
* Operadores de asignación: =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=
* Conversiones implícitas y explícitas

Cuando no hay expresiones dinámicas implicadas, la opción predeterminada en C# es el enlace estático, lo que significa que los tipos en tiempo de compilación de expresiones constitutivas se usan en el proceso de selección. Sin embargo, cuando una de las expresiones constitutivas de las operaciones arriba mencionadas es una expresión dinámica, la operación se enlaza dinámicamente.

### Tiempo de enlace

El enlace estático tiene lugar en tiempo de compilación, mientras que el dinámico ocurre en tiempo de ejecución. En las secciones siguientes, con tiempo de enlace se hace referencia al tiempo de compilación o de ejecución, según en cual tenga lugar el enlace.

En el ejemplo siguiente se muestran los conceptos de enlace dinámico y estático, y de tiempo de enlace:

object o = 5;  
dynamic d = 5;

Console.WriteLine(5); // static binding to Console.WriteLine(int)  
Console.WriteLine(o); // static binding to Console.WriteLine(object)  
Console.WriteLine(d); // dynamic binding to Console.WriteLine(int)

Las dos primeras llamadas se enlazan estáticamente: la sobrecarga de Console.WriteLine se elige según el tipo en tiempo de compilación de su argumento. De esta forma, el enlace se hace en tiempo de compilación.

La tercera llamada se enlaza dinámicamente: la sobrecarga de Console.WriteLine se elige según el tipo en tiempo de ejecución de su argumento. Esto sucede porque el argumento es una expresión dinámica, su tipo en tiempo de compilación es dynamic. De esta forma, el tiempo de enlace de esta tercera llamada es el tiempo de ejecución.

### Enlace dinámico

El objetivo del enlace dinámico es permitir que los programas de C# interactuar con objetos dinámicos, es decir, los objetos que no siguen las reglas normales del sistema de tipos de C#. Los objetos dinámicos pueden ser objetos de otros lenguajes de programación con diferentes sistemas de tipos o pueden ser objetos que se configuran mediante programación para implementar su propia semántica de enlace para operaciones diferentes.

El mecanismo por el que un objeto dinámico implementa sus propias semánticas se define según la implementación. Una interfaz dada, también definida según la implementación, se implementa mediante objetos dinámicos para indicar al tiempo en ejecución de C# que tienen semántica especial. De este modo, cuando las operaciones de un objeto dinámico están enlazadas dinámicamente, su propia semántica de enlace, en lugar de la de C# que se especifica en este documento, asume el control.

Aunque el objetivo del enlace dinámico es permitir que los programas de C# interoperen con objetos dinámicos, C# permite el enlace dinámico en todos los objetos, sean o no dinámicos. Esto permite una integración más suave de los objetos dinámicos, ya que los resultados de operaciones en ellos pueden no ser objetos dinámicos, pero aún son de un tipo desconocido para el programador en tiempo de compilación. También el enlace dinámico puede ayudar a eliminar el código basado en reflexión proclive a error incluso cuando ningún objeto implicado sea objeto dinámico.

En las secciones siguientes se describe cada construcción en el lenguaje exactamente cuando se aplica el enlace dinámico, qué comprobación se hace en tiempo de compilación (en caso de que se haga alguna) y cuál es la clasificación de la expresión y el resultado en tiempo de compilación.

### Tipos de datos de expresiones constitutivas

Cuando una operación se enlaza estáticamente, el tipo de una expresión constitutiva (por ejemplo un receptor, un argumento, un índice o un operando) siempre se considera que es el tipo en tiempo de compilación de esa expresión.

Cuando una operación se enlaza dinámicamente, el tipo de una expresión constitutiva se determina de diferentes formas dependiendo del tipo en tiempo de compilación de la expresión constitutiva.

* Una expresión constitutiva de tipo en tiempo de compilación dynamic se considera que tiene el tipo del valor real en el que la expresión evalúa en tiempo de ejecución.
* Una expresión constitutiva cuyo tipo en tiempo de compilación es un parámetro de tipo se considera que tiene el tipo al que dicho parámetro se enlaza en tiempo de ejecución.
* En cualquier otro caso, la expresión constitutiva se considera que tiene su tipo en tiempo de compilación.

## Operadores

Las expresiones se construyen a partir de operandos y operadores. Los operadores de una expresión indican qué operaciones se aplican a los operandos. Entre los ejemplos de operadores se incluyen +, -, \*, / y new. Son ejemplos de operandos los literales, campos, variables locales y expresiones.

Existen tres tipos de operadores:

* Operadores unarios. Los operadores unarios tienen un operando y utilizan la notación de prefijo (como –x) o de postfijo (como x++).
* Operadores binarios. Los operadores binarios tienen dos operandos y utilizan una notación infija (por ejemplo, x + y).
* Operador ternario. Solo existe un operador ternario, ?:, tiene tres operandos y utiliza notación infija (c? x: y).

El orden de evaluación de los operadores de una expresión está determinado por la prioridad y asociatividad de los operadores (§7.3.1).

Los operandos de una expresión se evalúan de izquierda a derecha. Por ejemplo, en F(i) + G(i++) \* H(i), se llama al método F con el valor antiguo i, después se llama al método G con el valor antiguo de i y, por último, se llama al método H con el valor nuevo de i. Esto es aparte y no está relacionado con la prioridad de los operadores.

Algunos operadores no se pueden sobrecargar. La sobrecarga de operadores permite utilizar implementaciones de operadores definidas por el usuario en operaciones en las que al menos uno de los operandos es de un tipo struct o clase definido por el usuario (§7.3.2).

### Prioridad y asociatividad de los operadores

Cuando una expresión contiene varios operadores, la prioridad de los operadores controla el orden de evaluación de los operadores individuales. Por ejemplo, la expresión x + y \* z se evalúa como x + (y \* z) porque el operador \* tiene prioridad sobre el operador binario +. La prioridad de un operador está determinada por la definición de su producción gramatical asociada. Por ejemplo, una expresión aditiva (additive-expression) consta de una secuencia de expresiones multiplicativas (multiplicative-expressions) separadas por los operadores + o -, lo que da a estos operadores una menor prioridad que a \*, / y %.

En la tabla siguiente se resumen todos los operadores, en orden de prioridad de mayor a menor:

|  |  |  |
| --- | --- | --- |
| **Sección** | **Categoría** | **Operadores** |
| 7.6 | Principal | x.y f(x) a[x] x++ x-- new  typeof default checked unchecked delegate |
| 7.7 | Unario | + - ! ~ ++x --x (T)x await x |
| 7.8 | Multiplicativo | \* / % |
| 7.8 | Sumatorio | + - |
| 7.9 | Desplazamiento | << >> |
| 7.10 | Comprobación de tipos y relacionales | < > <= >= is as |
| 7.10 | Igualdad | == != |
| 7.11 | AND lógico | & |
| 7.11 | XOR lógico | ^ |
| 7.11 | OR lógico | | |
| 7.12 | AND condicional | && |
| 7.12 | OR condicional | || |
| 7.13 | Uso combinado de Null | ?? |
| 7.14 | Condicional | ?: |
| 7.17, 7.15 | Asignación y expresión lambda | = \*= /= %= += -= <<= >>= &= ^= |=  => |

Cuando un operando se encuentra entre dos operadores con igual grado de prioridad, la asociatividad de los operadores controla el orden en que se ejecutan las operaciones.

* A excepción de los operadores de asignación y el operador de incorporación de NULL, todos los operadores binarios son asociativos por la izquierda, lo que significa que las operaciones se realizan de izquierda a derecha. Por ejemplo, x + y + z se evalúa como (x + y) + z.
* Los operadores de asignación, el operador de incorporación de NULL y el operador condicional (?:) son asociativos por la derecha, lo que significa que las operaciones se ejecutan de derecha a izquierda. Por ejemplo, x = y = z se evalúa como x = (y = z).

La precedencia y asociatividad pueden controlarse mediante el uso de paréntesis. Por ejemplo, x + y \* z primero multiplica y por z y después suma el resultado a x, pero (x + y) \* z primero suma x e y, y después multiplica el resultado por z.

### Sobrecarga de operadores

Todos los operadores unarios y binarios tienen implementaciones predefinidas que están disponibles automáticamente en cualquier expresión. Además de las implementaciones predefinidas, pueden introducirse implementaciones definidas por el usuario si se incluyen declaraciones operator en las clases y structs (§10.10). Las implementaciones de operador definidas por el usuario siempre tienen precedencia sobre las implementaciones de operador predefinidas: sólo se consideran las implementaciones de operador predefinidas cuando no existen implementaciones de operador definidas por el usuario que puedan aplicarse, como se explica en §7.3.3 y §7.3.4.

Los operadores unarios sobrecargables son:

+ - ! ~ ++ -- true false

Aunque true y false no se utilizan explícitamente en las expresiones (por lo que no se incluyen en la tabla de prioridades de §7.3.1), se consideran operadores porque se los llama en varios contextos de expresión: expresiones booleanas (§7.20) y expresiones que implican el condicional (§7.14) y los operadores lógicos condicionales (§7.12).

Los operadores binarios sobrecargables son:

+ - \* / % & | ^ << >> == != > < >= <=

Sólo los operadores mencionados pueden sobrecargarse. En concreto, no es posible sobrecargar accesos a miembros, llamadas a métodos o los operadores =, &&, ||, ??, ?:, =>, checked, unchecked, new, typeof, default, as e is.

Cuando se sobrecarga un operador binario, el operador correspondiente de asignación, si lo hay, también se sobrecarga de modo implícito. Por ejemplo, una sobrecarga del operador \* también es una sobrecarga del operador \*=. Esta categoría se explica con más detalle en la sección §7.17.2. Debe tenerse en cuenta que el propio operador de asignación (=) no se puede sobrecargar. Una asignación siempre realiza una simple copia bit a bit de un valor en una variable.

Las operaciones de conversión de tipo, como (T)x, se sobrecargan proporcionando conversiones definidas por el usuario (§6.4).

El acceso a elementos, del tipo a[x], no se considera un operador sobrecargable. En lugar de ello, se acepta la indización definida por el usuario mediante indizadores (§10.9).

En las expresiones, las referencias a los operadores se realizan mediante la notación de operadores y, en las declaraciones, las referencias a los operadores se realizan mediante la notación funcional. En la tabla siguiente se muestra la relación entre las notaciones de operador y funcional para los operadores unarios y binarios. En la primera entrada, op denota cualquier operador de prefijo unario sobrecargable. En la segunda entrada, op denota los operadores de sufijo unarios postfijos ++ y --. En la primera entrada, op denota cualquier operador binario sobrecargable.

|  |  |
| --- | --- |
| **Notación de operador** | **Notación funcional** |
| op x | operator op(x) |
| x op | operator op(x) |
| x op y | operator op(x, y) |

Las declaraciones de operador definidas por el usuario siempre requieren que por lo menos uno de los parámetros sea del tipo de la clase o struct que contiene la declaración del operador. Por lo tanto, no es posible que un operador definido por el usuario tenga la misma firma que un operador predefinido.

Las declaraciones de operador definidas por el usuario no pueden modificar la sintaxis, precedencia o asociatividad de un operador. Por ejemplo, el operador / siempre es un operador binario, siempre tiene el nivel de precedencia especificado en la §7.3.1 y siempre es asociativo por la izquierda.

Aunque es posible que un operador definido por el usuario realice cualquier cálculo que le interese, no se recomiendan las implementaciones que generan resultados distintos de los que intuitivamente pueden esperarse. Por ejemplo, una implementación de operator == debe comparar la igualdad de los dos operandos y devolver un resultado bool apropiado.

Las descripciones de los operadores individuales desde la §7.6 a la §7.12 especifican las implementaciones predefinidas de los operadores y cualquier regla adicional aplicable a cada operador. En las descripciones se utilizan los términos resolución de sobrecarga de operador unario, resolución de sobrecarga de operador binario y promoción numérica, cuyas definiciones se encuentran en las siguientes secciones.

### Resolución de sobrecarga de operador unario

Una operación con la estructura op x o x op, donde op es un operador unario sobrecargable, y x es una expresión de tipo X, se procesa como sigue:

* El conjunto de operadores candidatos definidos por el usuario suministrados por X para la operación operator op(x) se determina aplicando las reglas de la §7.3.5.
* Si el conjunto de operadores candidatos definidos por el usuario no está vacío, se convierte en el conjunto de operadores candidatos para la operación. De lo contrario, las implementaciones del operator op unario predefinidas, incluidos los formatos de elevación, se convierten en el conjunto de operadores candidatos para la operación. Las implementaciones predefinidas de un operador dado se especifican en la descripción del operador (§7.6 y §7.7).
* Las reglas de resolución de las sobrecargas de §7.5.3 se aplican al conjunto de operadores candidatos para seleccionar el mejor operador con respecto a la lista de argumentos (x), y este operador es el resultado del proceso de resolución de las sobrecargas. Si la resolución de las sobrecargas no puede seleccionar un operador único idóneo, se producirá un error en tiempo de enlace.

### Resolución de sobrecarga de operador binario

Una operación con el formato x op y , donde op es un operador binario sobrecargable, x es una expresión de tipo X e y es una expresión de tipo Y, se procesa como sigue:

* Se determina el conjunto de operadores candidatos definidos por el usuario suministrado por X y Y para la operación operator op(x, y). El conjunto consta de la unión de los operadores candidatos suministrados por X y los operadores candidatos suministrados por Y, cada uno de los cuales se determina mediante las reglas de §7.3.5. Para el conjunto combinado, los candidatos se combinan de la siguiente forma:
* Si X e Y son del mismo tipo, o si X e Y se derivan de un tipo base común, los operadores candidatos compartidos sólo se producen una vez en el conjunto combinado.
* Si hay una conversión de identidad entre X e Y, un operador opY proporcionado por Y tiene el mismo valor devuelto que un operador opX proporcionado por X y los tipos de operandos de opY tienen una conversión de identidad en los tipos de operandos correspondientes de opX, solo se produce opX en el conjunto.
* Si el conjunto de operadores candidatos definidos por el usuario no está vacío, se convierte en el conjunto de operadores candidatos para la operación. De lo contrario, las implementaciones del operator op binario predefinidas, incluidos los formatos de elevación, se convierten en el conjunto de operadores candidatos para la operación. Las implementaciones predefinidas de un operador dado se especifican en la descripción del operador (§7.8 a §7.12). Para los operadores de enumeración y delegado predefinidos, los únicos que se consideran son los definidos por un tipo de enumeración o delegado que sea el tipo de tiempo de enlace de uno de los operandos.
* Las reglas de resolución de las sobrecargas de §7.5.3 se aplican al conjunto de operadores candidatos para seleccionar el mejor operador con respecto a la lista de argumentos (x, y), y este operador es el resultado del proceso de resolución de las sobrecargas. Si la resolución de las sobrecargas no puede seleccionar un operador único idóneo, se producirá un error en tiempo de enlace.

### Operadores candidatos definidos por el usuario

Dados un tipo T y una operación operator op(A), donde op es un operador sobrecargable y A es una lista de argumentos, el conjunto de operadores candidatos definidos por el usuario suministrado por T para el operator op(A) se determina como sigue:

* Determinar el tipo T0. Si T es un tipo que acepta valores NULL, T0 es su tipo subyacente, de lo contrario T0 es igual a T.
* Para todas las declaraciones de operator op en T0 y todas los formatos de elevación de dichos operadores, si por lo menos un operador es aplicable (§7.5.3.1) con respecto a la lista de argumentos A, entonces el conjunto de operadores candidatos consta de todos los operadores aplicables en T0.
* De lo contrario, si T0 es object, el conjunto de operadores candidatos está vacío.
* En otros casos, el conjunto de operadores candidatos suministrado por T0 es el conjunto de operadores candidatos suministrado por la clase base directa de T0, o la clase base efectiva de T0 si T0 es un tipo de parámetro.

### Promociones numéricas

Una promoción numérica consiste en realizar de forma automática determinadas conversiones implícitas de los operandos de los operadores numéricos unarios y binarios predefinidos. La promoción numérica no es un mecanismo exclusivo, sino más bien un efecto de la aplicación de la resolución de las sobrecargas a los operadores predefinidos. La promoción numérica en concreto no afecta a la evaluación de los operadores definidos por el usuario, aunque dichos operadores puedan implementarse de manera que presenten efectos similares.

Como ejemplo de promoción numérica, consideremos las implementaciones predefinidas del operador binario \*:

int operator \*(int x, int y);  
uint operator \*(uint x, uint y);  
long operator \*(long x, long y);  
ulong operator \*(ulong x, ulong y);  
float operator \*(float x, float y);  
double operator \*(double x, double y);  
decimal operator \*(decimal x, decimal y);

Cuando se aplican las reglas de resolución de las sobrecargas (§7.5.3) a este conjunto de operadores, el efecto que se produce es la selección del primero de los operadores para el cual existen conversiones implícitas de los tipos de los operandos. Por ejemplo, para la operación b \* s, donde b es un byte y s es short, la resolución de las sobrecargas selecciona operator \*(int, int) como el mejor operador. De esta forma, el efecto producido es que b y s se convierten a int, y el tipo del resultado es int. Del mismo modo, para la operación i \* d, donde i es un int y d es double, la resolución de las sobrecargas selecciona operator \*(double, double) como el mejor operador.

#### Promociones numéricas unarias

Una promoción numérica unaria se produce para los operandos de los operadores unarios predefinidos +, – y ~. Una promoción numérica unaria consiste sencillamente en la conversión de operandos de tipo sbyte, byte, short, ushort o char al tipo int. Asimismo, para el operador unario –, la promoción numérica unaria convierte los operandos del tipo uint al tipo long.

#### Promociones numéricas binarias

La promoción numérica binaria se produce para los operandos de los operadores binarios predefinidos +, –, \*, /, %, &, |, ^, ==, !=, >, <, >= y <=. La promoción numérica binaria convierte implícitamente los dos operandos a un tipo común que, en caso de los operadores no relacionales, también se convierte en el tipo del resultado de la operación. Una promoción numérica binaria consiste en aplicar las reglas siguientes, en el orden en que se exponen aquí:

* Si uno de los operandos es de tipo decimal, el otro se convierte al tipo decimal o se produce un error en tiempo de enlace si el otro operando es de tipo float o double.
* O bien, si uno de los operandos es de tipo double, el otro se convierte al tipo double.
* O bien, si uno de los operandos es de tipo float, el otro se convierte al tipo float.
* De lo contrario, si uno de los operandos es de tipo ulong, el otro se convierte al tipo ulong o se produce un error en tiempo de enlace si el otro operando es de tipo sbyte, short, int o long.
* O bien, si uno de los operandos es de tipo long, el otro se convierte al tipo long.
* O bien, si uno de los operandos es de tipo uint y el otro es de tipo sbyte, short o int, los dos operandos se convierten al tipo long.
* O bien, si uno de los operandos es de tipo uint, el otro se convierte al tipo uint.
* O bien, los dos operandos se convierten al tipo int.

Téngase en cuenta que la primera regla no permite las operaciones que mezclan el tipo decimal con los tipos double y float. La regla se basa en que no hay conversiones implícitas entre el tipo decimal y los tipos double y float.

También debe tenerse en cuenta que un operando no puede ser de tipo ulong si el otro es de un tipo entero con signo. El motivo es que no existe un tipo entero que pueda representar la gama completa de ulong así como los tipos enteros con signo.

En los dos casos anteriores, puede utilizarse una expresión de conversión de tipos para convertir de forma explícita uno de los operandos a un tipo que sea compatible con el otro.

En el ejemplo

decimal AddPercent(decimal x, double percent) {  
 return x \* (1.0 + percent / 100.0);  
}

se produce un error de tiempo de enlace porque un decimal no puede multiplicarse por un double. El error se resuelve mediante la conversión explícita del segundo operando a decimal, de la manera siguiente:

decimal AddPercent(decimal x, double percent) {  
 return x \* (decimal)(1.0 + percent / 100.0);  
}

### Operadores de elevación

Los operadores de elevación permiten utilizar operadores predefinidos y definidos por el usuario que funcionan en tipos de valor que no aceptan valores NULL con estructuras que aceptan valores NULL de dichos tipos. Los operadores de elevación se construyen a partir de operadores predefinidos y definidos por el usuario que cumplen ciertos requisitos como se describe a continuación:

* Para los operadores unarios

+ ++ - -- ! ~

existe un formato de elevación de un operador si el operando y los tipos de resultado son tipos de valor que no aceptan valores NULL. El formato de elevación se construye agregando un modificador ? único al operando y a los tipos de resultado. El operador de elevación genera un valor NULL si el operando es NULL. De lo contrario, el operador de elevación desajusta el operando, aplica el operador subyacente y ajusta el resultado.

* Para los operadores binarios

+ - \* / % & | ^ << >>

existe un formato de elevación si el operando y los tipos de resultado son todos tipos de valor que no aceptan valores NULL. El formato de elevación se construye agregando un modificador ? único a cada operando y al tipo de resultado. El operador de elevación genera un valor NULL si uno o varios operandos son NULL (una de las excepciones son los operadores & y | del tipo bool? como se describe en la sección §7.11.3). De lo contrario, el operador de elevación desajusta los operandos, aplica el operador subyacente y ajusta el resultado.

* Para los operadores de igualdad

== !=

existe un formato de elevación si los tipos de operando son tipos de valor que no aceptan valores NULL y si el tipo de resultado es bool. El formato de elevación se construye agregando un modificador ? único a cada tipo de operando. El operador de elevación considera que dos valores NULL son iguales y que un valor NULL no es igual a un valor que no es NULL. Si ninguno de los operandos es NULL, el operador de elevación desajusta los operandos y aplica el operador subyacente para generar el resultado bool.

* Para los operadores relacionales

< > <= >=

existe un formato de elevación si los tipos de operando son tipos de valor que no aceptan valores NULL y si el tipo de resultado es bool. El formato de elevación se construye agregando un modificador ? único a cada tipo de operando. El operador de elevación genera el valor false si uno o ambos operandos es NULL. De lo contrario, el operador de elevación desajusta los operandos y aplica el operador subyacente para generar el resultado bool.

## Búsqueda de miembros

Una búsqueda de miembros es el proceso por el cual se determina el significado de un nombre en el contexto de un tipo. Una búsqueda de miembros puede ocurrir como parte de la evaluación de un nombre simple (simple-name) (§7.6.2) o un acceso a miembro (member-access) (§7.6.4) en una expresión. Si se produce un acceso a miembros (member-access) o un nombre simple (simple-name) como la expresión primaria (primary-expression) de una expresión de invocación (invocation-expression) (§7.6.5.1), se dice que el miembro se ha invocado.

Si un miembro es un método o evento, o si es una constante, campo o propiedad de un tipo delegado (§15) o de tipo dynamic (§4.7), se dice que el miembro es invocable.

La búsqueda de miembros no sólo tiene en cuenta el nombre de un miembro, sino que también considera el número de parámetros de tipo que el miembro tiene independientemente de su accesibilidad. Para la búsqueda de miembros, los métodos genéricos y los tipos genéricos anidados tienen el número de parámetros de tipo indicado en sus respectivas declaraciones y el resto de los miembros no tienen ningún parámetro tipo.

Una búsqueda de miembros de un nombre N con parámetros de tipo K en un tipo T se procesa como sigue:

* En primer lugar, se determina un conjunto de miembros accesibles denominado N:
* Si T es un parámetro de tipo, el conjunto es la unión de conjuntos de miembros accesibles denominados N en cada uno de los tipos especificados como una restricción principal o secundaria (§10.1.5) para T, junto con el conjunto de miembros accesibles denominado N en object.
* De lo contrario, el conjunto está formado por todos los miembros accesibles (§3.5) denominados N en T, incluidos los miembros heredados y los miembros accesibles denominados N en object. Si T es un tipo construido, el conjunto de miembros se obtiene sustituyendo los argumentos de tipo como se describe en §10.3.2. Se excluyen del conjunto los miembros que incluyen un modificador override.
* A continuación, si K es cero, se quitan todos los tipos anidados cuyas declaraciones incluyan parámetros de tipo. Si K no es cero, se quitan todos los miembros con un número diferente de parámetros de tipo. Tenga en cuenta que cuando K es cero, no se quitan los métodos que tienen parámetros de tipo puesto que el proceso de inferencia de tipos (§7.5.2) puede inferir los argumentos de tipo.
* A continuación, si el miembro es invocado, se quitan del conjunto todos los miembros no invocables.
* A continuación, se quitan del conjunto los miembros que están ocultos por otros miembros. Por cada miembro S.M del conjunto, donde S es el tipo en el que se declara el miembro M, se aplican las siguientes reglas:
* Si M es un miembro de constante, campo, propiedad, evento, tipo o enumeración, entonces se quitan del conjunto todos los miembros declarados en un tipo base de S.
* Si M es una declaración de tipo, entonces todos los miembros que no sean declaraciones en un tipo base de S se quitan del conjunto, se eliminarán todas las declaraciones de tipo con el mismo número de parámetros de tipo que M declarados en un tipo base de S.
* Si M es un método, se quitarán del conjunto todos los miembros que no son métodos declarados en un tipo base de S.
* A continuación, se quitan del conjunto los miembros de interfaz ocultos por miembros de clase. Este paso sólo tiene efecto si T es un parámetro de tipo y T tiene una clase base efectiva diferente de object y un conjunto de interfaces efectivas que no estén vacías (§10.1.5). Por cada miembro S.M del conjunto, donde S es el tipo en el que se declara el miembro M, se aplican las siguientes reglas si S es una declaración de clase diferente de object:
* Si M es un miembro de constante, campo, propiedad, evento, miembro de enumeración o declaración de tipo, se quitan del conjunto todos los miembros declarados en una declaración de interfaz.
* Si M es un método, entonces todos los miembros que no son métodos declarados en una declaración de interfaz se quitan del conjunto, así como todos los métodos con la misma signatura que M declarados en una declaración de interfaz.
* Por último, una vez quitados los miembros ocultos, se determina el resultado de la búsqueda:
* Si el conjunto está formado por un solo miembro que no es un método, entonces este miembro es el resultado de la búsqueda.
* O bien, si el conjunto sólo contiene métodos, entonces este grupo de métodos es el resultado de la búsqueda.
* De lo contrario, la búsqueda es ambigua y se genera un error en tiempo de enlace.

Para búsquedas de miembros en tipos que no sean interfaces ni parámetros de tipo, y búsquedas de miembros en interfaces que sean estrictamente de herencia simple (cada interfaz en la cadena de la herencia tienen exactamente cero o una interfaz base directa), el efecto de las reglas de búsqueda es sencillamente que los miembros derivados ocultan a los miembros base del mismo nombre o la misma firma. Este tipo de búsquedas de herencia simple nunca son ambiguas. Las ambigüedades que pueden surgir de las búsquedas de miembros en interfaces de herencia múltiple se describen en la §13.2.5.

### Tipos base

Para los fines de búsqueda de miembros, se considera que un tipo T tiene los siguientes tipos base:

* Si T es object o dynamic, entonces T no tiene tipo base.
* Si T en un tipo enum (enum-type), los tipos base de T son los tipos de clase System.Enum, System.ValueType y object.
* Si T en un tipo struct (struct-type), los tipos base de T son los tipos de clase System.ValueType y object.
* Si T es un tipo de clase (class-type), los tipos base de T son las clases base de T, incluido el tipo de clase object.
* Si T es un tipo de interfaz (interface-type), los tipos base de T son las interfaces base de T y el tipo de clase object.
* Si T en un tipo de matriz (array-type), los tipos base de T son los tipos de clase System.Array y object.
* Si T en un tipo delegado (delegate-type), los tipos base de T son los tipos de clase System.Delegate y object.

## Miembros de función

Los miembros de función son miembros que contienen instrucciones ejecutables. Siempre son miembros de tipos y no pueden ser miembros de espacios de nombres. C# define las siguientes categorías de miembros de función:

* Métodos
* Propiedades
* Eventos
* Indizadores
* Operadores definidos por el usuario
* Constructores de instancia
* Constructores static
* Destructores

Excepto para los destructores y los constructores estáticos (que no se pueden invocar de manera explícita), las instrucciones contenidas en miembros de función se ejecutan mediante invocaciones de miembros de función. La sintaxis concreta de la programación de invocaciones de miembros de función depende de la categoría del miembro concreto.

La lista de argumentos (§7.5.1) de la invocación de un miembro de función proporciona valores reales o referencias de variable a los parámetros del miembro de función.

Las llamadas a métodos genéricos pueden usar la inferencia de tipos para determinar el conjunto de argumentos de tipo que se pasan al método. Este proceso se describe en §7.5.2.

Las llamadas de métodos, indizadores, operadores y constructores de instancia utilizan la resolución de sobrecargas para determinar a qué miembro de un conjunto candidato de miembros de función se debe llamar. Este proceso se describe en §7.5.3.

Una vez identificado un miembro de función concreto en tiempo de enlace, posiblemente mediante resolución de sobrecargas, el proceso exacto de invocar el miembro de función en tiempo de ejecución se explica en §7.5.4.

En la tabla siguiente se resume el procesamiento que tiene lugar en las construcciones que implican las seis categorías de miembros de función, que se puede invocar explícitamente. En la tabla, e, x, y y value indican expresiones clasificadas como variables o valores, T indica una expresión clasificada como un tipo, F es el nombre simple de un método y P es el nombre simple de una propiedad.

| **Construcción** | **Ejemplo** | **Descripción** |
| --- | --- | --- |
| Invocación de método | F(x, y) | Se aplica la resolución de sobrecargas para seleccionar el mejor método F de la clase o struct contenedora. Se llama al método con la lista de argumentos (x, y). Si el método no es static, la expresión de instancia es this. |
| T.F(x, y) | Se aplica la resolución de sobrecargas para seleccionar el mejor método F de la clase o struct T. Se produce un error en tiempo de enlace si el método no es static. Se llama al método con la lista de argumentos (x, y). |
| e.F(x, y) | Se aplica la resolución de sobrecargas para seleccionar el mejor método F de la clase, struct o interfaz dada por el tipo de e. Se produce un error en tiempo de enlace si el método es static. Se llama al método con la expresión de instancia e y la lista de argumentos (x, y). |
| Acceso de propiedad | P | Se invoca al descriptor de acceso get de la propiedad P en la clase o struct contenedoras. Se produce un error en tiempo de compilación si P es de solo escritura. Si P no es static, la expresión de instancia es this. |
| P = value | Se invoca al descriptor de acceso set de la propiedad P en la clase o struct contenedora con la lista de argumentos (value). Se produce un error en tiempo de compilación si P es de solo lectura. Si P no es static, la expresión de instancia es this. |
| T.P | Se invoca al descriptor de acceso get de la propiedad P en la clase o struct T. Se produce un error en tiempo de compilación si P no es static o si P es de solo escritura. |
| T.P = value | Se invoca al descriptor de acceso set de la propiedad P en la clase o struct T con la lista de argumentos (value). Se produce un error en tiempo de compilación si P no es static o si P es de solo lectura. |
| e.P | Se invoca al descriptor de acceso get de la propiedad P en la clase, struct o interfaz dada por el tipo de e con la expresión de instancia e. Se produce un error en tiempo de enlace si P es static o si P es de solo escritura. |
| e.P = value | Se invoca el descriptor de acceso set de la propiedad P en la clase, struct o interfaz dada por el tipo de e con la expresión de instancia e y la lista de argumentos (value). Se produce un error en tiempo de enlace si P es static o si P es de solo lectura. |
| Acceso a evento | E += value | Se invoca al descriptor de acceso add del evento E en la clase o struct contenedoras. Si E no es static, la expresión de instancia es this. |
| E -= value | Se invoca al descriptor de acceso remove del evento E en la clase o struct contenedoras. Si E no es static, la expresión de instancia es this. |
| T.E += value | Se invoca al descriptor de acceso add del evento E en la clase o struct T. Si E no es estático, se produce un error en tiempo de enlace. |
| T.E -= value | Se invoca al descriptor de acceso remove del evento E en la clase o struct T. Si E no es estático, se produce un error en tiempo de enlace. |
| e.E += value | Se invoca al descriptor de acceso add del evento E en la clase, struct o interfaz dada por el tipo de e con la expresión de instancia e. Si E es static, se produce un error en tiempo de enlace. |
| e.E -= value | Se invoca al descriptor de acceso remove del evento E en la clase, struct o interfaz dada por el tipo de e con la expresión de instancia e. Si E es static, se produce un error en tiempo de enlace. |
| Acceso al indizador | e[x, y] | La resolución de sobrecarga se aplica para seleccionar el mejor indizador de la clase, struct o interfaz dada por el tipo de e. El descriptor de acceso get del indizador se invoca con la expresión de instancia e y la lista de argumentos (x, y). Se produce un error en tiempo de enlace si el indizador es de sólo escritura. |
| e[x, y] = value | Se aplica la resolución de sobrecargas para seleccionar el mejor indizador de la clase, struct o interfaz dada por el tipo de e. Se invoca el descriptor de acceso set del indizador con la expresión de instancia e y la lista de argumentos (x, y, value). Se produce un error en tiempo de enlace si el indizador es de sólo lectura. |
| Invocación de operador | -x | Se aplica la resolución de sobrecargas para seleccionar el mejor operador unario de la clase, struct o interfaz dada por el tipo de x. Se invoca al operador seleccionado con la lista de argumentos (x). |
| x + y | Se aplica la resolución de sobrecargas, para seleccionar el mejor operador binario de las clases o structs dados por los tipos de x e y. El operador seleccionado se invoca con la lista de argumentos (x, y). |
| Invocación de constructores de instancias | new T(x, y) | La resolución de sobrecarga se aplica para seleccionar el mejor constructor de la clase o struct T. Se invoca al constructor de instancias con la lista de argumentos (x, y). |

### Listas de argumentos

Toda invocación de delegado y miembro de función incluye una lista de argumentos que proporciona los valores reales o las referencias de variable de los parámetros del miembro de función. La sintaxis para especificar la lista de argumentos de una invocación de miembro de función depende de la categoría del miembro de función:

* Para los constructores de instancia, métodos, indizadores y delegados, los argumentos se especifican como una lista de argumentos (argument-list), como se explica a continuación. Para los indicadores, cuando se invoca el descriptor de acceso set, la lista de argumentos incluye además la expresión especificada como el operando derecho del operador de asignación.
* Para las propiedades, la lista de argumentos está vacía cuando se llama al descriptor de acceso get, y está formada por la expresión especificada como el operando derecho del operador de asignación cuando se invoca el descriptor de acceso set.
* Para los eventos, la lista de argumentos está formada por la expresión especificada como el operando derecho del operador += o -=.
* Para los operadores definidos por el usuario, la lista de argumentos está formada por el operando único del operador unario o los dos operandos del operador binario.

Los argumentos de propiedades (§10.7), eventos (§10.8) y operadores definidos por el usuario (§10.10) siempre se pasan como parámetros de valor (§10.6.1.1). Los argumentos de indizadores (§10.9) siempre se pasan como parámetros de valor (§10.6.1.1) o matrices de parámetros (§10.6.1.4). Los parámetros de referencia y de salida no se aceptan para estas categorías de miembros de función.

Los argumentos de llamadas a constructores de instancia, métodos, indicadores o delegados se especifican como una lista de argumentos (argument-list):

argument-list:  
argument  
argument-list , argument

argument:  
argument-nameopt argument-value

argument-name:  
identifier :

argument-value:  
expression  
ref variable-reference  
out variable-reference

Una lista de argumentos (argument-list) consta de uno o más argumentos (arguments), separados por comas. Cada argumento consiste en un nombre de argumento (argument-name) opcional seguido de un valor de argumento (argument-value). Un argumento con un nombre de argumento (argument-name) se conoce como argumento con nombre, mientras que un argumento sin nombre de argumento es un argumento posicional. Es un error que un argumento posicional aparezca después de un argumento con nombre en una lista de argumentos.

El valor de argumento puede tomar una de las siguientes formas:

* Una expresión (expression), para indicar que el argumento se pasa como un parámetro de valor (§10.6.1.1).
* La palabra clave ref seguida de una referencia a variable (variable-reference) (§5.4), para indicar que el argumento se pasa como un parámetro de referencia (§10.6.1.2). Una variable debe estar asignada de manera definitiva (§5.3) para poder pasarla como parámetro de referencia.
* La palabra clave out seguida por una referencia a variable (variable-reference) (§5.4), para indicar que el argumento se pasa como un parámetro de salida (§10.6.1.3). Una variable se considera asignada de manera definitiva (§5.3) después de una llamada a miembro de función en la que la variable se pasa como un parámetro de salida.

El formato determina el **modo de pasar los parámetros** del argumento: valor, referencia o resultado, respectivamente.

#### Parámetros correspondientes

Para cada argumento de la lista de argumentos tiene que haber un parámetro correspondiente en el delegado o miembro de función que se está invocando.

La lista de parámetros usada a continuación se determina del modo siguiente:

* Para los métodos virtuales y los indizadores definidos en clases, la lista de parámetros se toma de la declaración más específica o invalidación del miembro de función, comenzando por el tipo estático del receptor y buscando en todas sus clases base.
* Para los métodos de interfaz y los indizadores, la lista de parámetros se toma de la definición más específica del miembro, comenzando por el tipo de interfaz y buscando en las interfaces base. Si no se encuentra una lista única de parámetros, se construye una lista de parámetros con nombres inaccesibles y sin parámetros opcionales, de forma que las invocaciones no puedan usar parámetros con nombre ni omitir argumentos opcionales.
* Para los métodos parciales, se usa la lista de parámetros de la declaración de definición del método parcial.
* Para el resto de los delegados y miembros de función, solo hay una única lista de parámetros, que es la que se utiliza.

La posición de un argumento o parámetro se define como el número de argumentos o parámetros que le preceden en la lista de argumentos o en la lista de parámetros.

Los parámetros correspondientes para los argumentos de los miembros de función se establecen como sigue:

* Los argumentos de la lista de argumentos de delegados, indicadores, métodos y constructores de instancias:
* Un argumento posicional en el que un parámetro fijado aparece en la misma posición en la lista correspondiente a ese parámetro.
* Un argumento posicional de un miem pasado por la matriz de parámetros debe ser una expresión única de un tipo que sea implícitamente convertible (§) al tipo de la matriz de parámetros.
* Un argumento posicional de un miembro de función con una matriz de parámetros invocada en su forma expandida, donde ningún parámetro fijado aparece en la misma posición en la lista de parámetros, corresponde a un elemento de la matriz de parámetros.
* Un argumento con nombre se corresponde con el parámetro del mismo nombre de la lista de parámetros.
* Para los indizadores, cuando se invoca el descriptor de acceso set, la expresión especificada como el operando derecho del operador de asignación corresponde al parámetro implícito value de la declaración del descriptor de acceso set.
* Para las propiedades, cuando se invoca el descriptor de acceso get no hay argumentos. Cuando se invoca el descriptor de acceso set, la expresión especificada como el operando derecho del operador de asignación corresponde al parámetro implícito value de la declaración del descriptor de acceso set.
* Para los operadores unarios definidos por el usuario (conversiones incluidas), el operando único corresponde al parámetro único de la declaración del operador.
* Para los operadores binarios definidos por el usuario, el operando izquierdo corresponde al primer parámetro y el derecho al segundo parámetro de la declaración del operador.

#### Evaluación en tiempo de ejecución de listas de argumentos

Durante el procesamiento de una invocación de miembro de función (§7.5.4) en tiempo de ejecución, las referencias de expresiones o de variables de una lista de argumentos se evalúan por orden, de izquierda a derecha, como sigue:

* Para un parámetro de valor, se evalúa la expresión del argumento y se realiza una conversión implícita (§6.1) al tipo de parámetro correspondiente. El valor resultante pasa a ser el valor inicial del parámetro de valor en la invocación del miembro de función.
* Para un parámetro de referencia o de salida, se evalúa la referencia de variable, mientras que la ubicación de almacenamiento resultante pasa a ser la ubicación de almacenamiento, representada por el parámetro de la invocación del miembro de función. Si la referencia de variable dada como parámetro de referencia o de salida es un elemento de matriz de tipo de referencia (reference-type), se lleva a cabo una comprobación en tiempo de ejecución para garantizar que el tipo de elemento de la matriz es idéntico al tipo del parámetro. Si esta comprobación no es correcta, se inicia una excepción System.ArrayTypeMismatchException.

Los métodos, indizadores y constructores de instancia pueden declarar su parámetro situado más a la derecha como una matriz de parámetros (§10.6.1.4). Estos miembros de función se invocan en su forma normal o en su forma expandida, dependiendo de lo que sea aplicable (§7.5.3.1):

* Si se invoca un miembro de función con una matriz de parámetros en su forma normal, el argumento pasado por la matriz de parámetros debe ser una expresión única que sea implícitamente convertible (§6.1) al tipo de la matriz de parámetros. En este caso, la matriz de parámetros actúa exactamente como un parámetro de valor.
* Si se invoca un miembro de función con una matriz de parámetros en su forma expandida, la invocación debe especificar cero o más argumentos posicionales para la matriz de parámetros, donde cada argumento es una expresión implícitamente convertible (§6.1) al tipo de los elementos de la matriz de parámetros. En este caso, la invocación crea una instancia del tipo de la matriz de parámetros con una longitud correspondiente al número de argumentos, inicializa los elementos de la instancia de matriz con los valores de los argumentos especificados y utiliza la instancia de matriz recién creada como argumento real.

Las expresiones de una lista de argumentos siempre se evalúan en el orden en que están escritas. Por lo tanto, el siguiente ejemplo:

class Test  
{  
 static void F(int x, int y = -1, int z = -2) {  
 System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);  
 }

static void Main() {  
 int i = 0;  
 F(i++, i++, i++);  
 F(z: i++, x: i++);  
 }  
}

produce el resultado

x = 0, y = 1, z = 2  
x = 4, y = -1, z = 3

Las reglas de covarianza matricial (§12.5) permiten que un valor de un tipo de matriz A[] se trate como una referencia a una instancia de un tipo matricial B[], siempre que exista una conversión implícita de referencias de B a A. Debido a estas reglas, si se pasa un elemento de matriz de un tipo de referencia (reference-type) como un parámetro de referencia o de salida, se lleva a cabo una comprobación en tiempo de ejecución para garantizar que el tipo de elemento de la matriz es idéntico al tipo del parámetro. En el ejemplo

class Test  
{  
 static void F(ref object x) {...}

static void Main() {  
 object[] a = new object[10];  
 object[] b = new string[10];  
 F(ref a[0]); // Ok  
 F(ref b[1]); // ArrayTypeMismatchException  
 }  
}

la segunda invocación de F produce una excepción System.ArrayTypeMismatchException, puesto que el tipo de elemento real de b es string y no object.

Cuando se llama a un miembro de función con una matriz de parámetros en su forma expandida, la invocación se trata exactamente como si se hubiera insertado una expresión de creación de matriz con un inicializador de matriz (§7.6.10.4) alrededor de los parámetros expandidos. Por ejemplo, dada la declaración

void F(int x, int y, params object[] args);

las invocaciones siguientes de la forma expandida del método

F(10, 20);  
F(10, 20, 30, 40);  
F(10, 20, 1, "hello", 3.0);

se corresponden exactamente con

F(10, 20, new object[] {});  
F(10, 20, new object[] {30, 40});  
F(10, 20, new object[] {1, "hello", 3.0});

En particular, debe tenerse en cuenta que si se pasan cero argumentos como matriz de parámetros, se crea una matriz vacía.

Cuando se omiten los argumentos de un miembro de función con parámetros opcionales correspondientes, los argumentos predeterminados de la declaración de miembros de función se pasan de forma implícita. Como estos son siempre constantes, su evaluación no tiene efecto en el orden de evaluación de los argumentos restantes.

### Inferencia de tipos

Cuando se llama a un método genérico sin especificar argumentos de tipo, una inferencia de tipos procesa los intentos para deducir los argumentos de tipo para la llamada. La presencia de la inferencia de tipo permite utilizar una sintaxis más adecuada para llamar a un método genérico y permite que el programador no especifique información de tipo redundante. Por ejemplo, dada la declaración de método:

class Chooser  
{  
 static Random rand = new Random();

public static T Choose<T>(T first, T second) {  
 return (rand.Next(2) == 0)? first: second;  
 }  
}

es posible invocar el método Choose sin especificar explícitamente un argumento de tipo:

int i = Chooser.Choose(5, 213); // Calls Choose<int>

string s = Chooser.Choose("foo", "bar"); // Calls Choose<string>

Mediante la inferencia de tipo, los argumentos de tipo int y string se determinan desde los argumentos al método.

La inferencia de tipo tiene lugar como parte del proceso en tiempo de enlace de una invocación de método (§7.6.5.1) y tiene lugar antes del paso de resolución de sobrecarga de la invocación. Cuando se especifica un grupo de métodos concreto en una invocación del método y no se especifican argumentos como parte de esta invocación, se aplica la inferencia de tipo para cada método genérico del grupo de métodos. Si la inferencia de tipo es correcta, los argumentos de tipo se utilizan para determinar los tipos de argumentos para la resolución de sobrecarga subsiguiente. Si la resolución de sobrecarga elige un método genérico como el que se va a invocar, los argumentos de tipo que se deducen se utilizan como argumentos de tipo reales para la invocación. Si no se puede realizar la inferencia de tipo para un método concreto, dicho método no participa en la resolución de sobrecarga. Aunque se genere un error en la inferencia de tipo, esto no provocará un error en tiempo de enlace. Sin embargo, en ocasiones esto lleva a un error en tiempo de enlace cuando la resolución de sobrecarga no puede encontrar ningún método aplicable.

Si el número de argumentos suministrados es diferente del número de parámetros del método, la deducción genera inmediatamente un error. En caso contrario, se asume que el método genérico tiene la siguiente firma:

Tr M<X1…Xn>(T1 x1 … Tm xm)

Con una llamada a método que tiene el formato M(E1 …Em) la tarea de la inferencia de tipos es buscar argumentos de tipo único S1…Sn para cada uno de los parámetros de tipo X1…Xn con el fin de que la llamada M<S1…Sn>(E1…Em)sea válida.

Durante el proceso de inferencia, cada parámetro de tipo Xi se fija (*fixed*) a un tipo determinado Si o queda sin fijar (*unfixed*) con un conjunto asociado de límites (*bounds).* Cada uno de los límites es algún tipo T . Inicialmente cada variable de tipo Xi es sin fijar con un conjunto de límites vacío.

La inferencia de tipos tiene lugar en fases. En cada fase se intentará inferir argumentos de tipo para más variables de tipo basados en los hallazgos de la fase anterior. En la primera fase se realizan algunas inferencias iniciales de límites, mientras que en la segunda fase se fijan variables de tipo a tipos específicos y se infieren más límites. Es posible que la segunda fase se tenga que repetir varias veces.

*Nota:* la inferencia de tipos tiene lugar no solo cuando se llama a un método genérico. La inferencia de tipos para la conversión de grupos de métodos se describe en la sección §7.5.2.13 y cómo buscar el mejor tipo común de un conjunto de expresiones se describe en la sección §7.5.2.14.

#### La primera fase

Para cada uno de los argumentos de método Ei:

* Si Ei es una función anónima, se realiza una inferencia explícita de tipo de parámetro (§7.5.2.7) desde Ei a Ti
* En caso contrario, si Ei tiene un tipo U y xi es un parámetro de valor, se realiza una *inferencia de límite inferior* de U a Ti.
* En caso contrario, si Ei tiene un tipo U y xi es un parámetro ref o out, entonces se realiza una inferencia exacta de U a Ti.
* En caso contrario, no se hace inferencia para este argumento.

#### La segunda fase

La segunda fase se realiza de la siguiente forma:

* Todas las variables de tipo sin fijar Xi que no dependen de (§7.5.2.5) ninguna Xj se fijan (§7.5.2.10).
* Si no existen variables de este tipo, todas las variables de tipo sin fijar Xi se fijan, y para ellas se cumplen las siguientes condiciones:
  + Hay al menos una variable de tipo Xj que depende de Xi
  + Xi tiene un conjunto vacío de límites
* Si no existieran variables de este tipo y hubiera variables de tipo sin fijar, no se produciría correctamente la inferencia de tipos.
* En caso contrario, si no hubiera ninguna otra variable de tipo sin fijar, la inferencia de tipos se produciría correctamente.
* En caso contrario, para todos los argumentos Ei con el tipo de parámetro correspondiente Ti donde los tipos de resultado (§7.5.2.4) contienen variables de tipo sin fijar Xj pero los tipos de entrada (§7.5.2.3) no, se produce una inferencia de tipos de resultado (§7.5.2.6) desde Ei a Ti. A continuación, se repite la segunda fase.

#### Tipos de entrada

Si E es un grupo de métodos o una función anónima con asignación de tipo implícita y T es un tipo delegado o tipo de árbol de expresiones, entonces todos los tipos de parámetro de T son *tipos de entrada* *de* E *con el tipo* T.

#### Tipos de resultado

Si E es un grupo de métodos o una función anónima y T es un tipo delegado o tipo de árbol de expresiones, entonces el tipo de resultado de T es un *tipo de entrada* *de* E *con el tipo* T.

#### Dependencia

Una variable de tipo sin fijar Xi *depende directamente de* una variable de tipo sin fijar Xj si para algún argumento Ek con el tipo Tk Xj aparece en un tipo de entrada de Ek con el tipo Tk y Xi aparece en un tipo de salida de Ek con el tipo Tk.

Xj *depende de* Xi si Xj depende directamente de Xi o si Xi depende directamente de Xk y Xk depende de Xj. Así, “depende de” es el cierre transitivo, que no reflexivo de “depende directamente de”.

#### Inferencias de tipo de resultado

Una inferencia de tipos de resultado se realiza desde una expresión E a un tipo T de la siguiente forma:

* Si E es una función anónima con el tipo devuelto inferido U (§7.5.2.12) y T es un tipo de delegado o tipo de árbol de expresiones con un tipo devuelto Tb, se realiza una inferencia de límite inferior (§7.5.2.9) de U a Tb.
* En caso contrario, si E es un grupo de métodos y T es un tipo delegado o tipo de árbol de expresiones con tipos de parámetro T1…Tk y el tipo devuelto Tb, y la resolución de sobrecarga de E con los tipos T1…Tk genera un único método con el tipo devuelto U, entonces se realiza una inferencia de límite inferior desde U a Tb.
* En caso contrario, si E es una expresión con el tipo U, entonces se realiza una inferencia de límite inferior desde U a T.
* En caso contrario, no se realiza ninguna inferencia.

#### Inferencias explícitas de tipo de parámetro

Una inferencia explícita de tipo de parámetro se realiza a partir de una expresión E a un tipo T de la siguiente forma:

* Si E es una función anónima con asignación de tipo explícita que tiene tipos de parámetro U1…Uk y T es el tipo delegado o tipo de árbol de expresiones con tipos de parámetro V1…Vk, entonces para cada Ui se realiza una inferencia exacta (§7.5.2.8) desde Ui al correspondiente Vi.

#### Inferencias exactas

Una inferencia exacta desde un tipo U a un tipo V se realiza de la siguiente manera:

* Si V es una de las variables de tipo sin fijar Xi, entonces U se agrega al conjunto de límites exactos para Xi.
* En caso contrario, los conjuntos V1…Vk y U1…Uk se determinan comprobando si se aplica alguno de los casos siguientes:
* V es un tipo de matriz V1[…] y U es un tipo de matriz U1[…] del mismo rango
* V es el tipo V1? y U es el tipo U1?
* V es un tipo construido C<V1…Vk> and U es un tipo construido C<U1…Uk>

Si ninguno de estos casos es aplicable, se realiza una inferencia exacta de cada Ui en el correspondiente Vi.

* En caso contrario, no se realiza ninguna inferencia.

#### Inferencias de límite inferior

Una inferencia de límite inferior desde un tipo U a un tipo V se realiza de la siguiente manera:

* Si V es una de las variables de tipo sin fijar Xi, entonces U se agrega al conjunto de límites inferiores para Xi.
* En caso contrario, si V es el tipo V1? y U es el tipo U1? entonces se realiza una interferencia de límite inferior de U1 a V1.
* En caso contrario, los conjuntos U1…Uk y V1…Vk se determinan comprobando si se aplica alguno de los casos siguientes:
* V es un tipo de matriz V1[…]y U es un tipo de matriz U1[…]del mismo rango
* V es uno de IEnumerable<V1>, ICollection<V1> o IList<V1> y U es un tipo de matriz unidimensional U1[]
* V es una clase, struct, interfaz o tipo de delegado construido C<V1…Vk> y hay un tipo único C<U1…Uk> tal que U (o, si U es un parámetro de tipo, su clase base efectiva o cualquier miembro de su conjunto de interfaz efectiva) es idéntico a C<U1…Uk> o lo implementa o lo hereda de forma directa o indirecta.

(La restricción de “unicidad” significa que en el caso interface C<T>{} class U: C<X>, C<Y>{}, no se hace inferencia cuando se infiere de U a C<T> porque U1 podría ser X o Y).

Si ninguno de estos casos es aplicable, se realiza una inferencia de cada Ui en el correspondiente Vi, del modo siguiente:

* Si no se sabe si Ui es un tipo de referencia, se hace una *inferencia exacta*
* En caso contrario, si U es un tipo de matriz, se realiza una *inferencia de límite inferior*.
* En caso contrario, si V es C<V1…Vk>, la inferencia depende del parámetro de tipo i-enésimo de C:
* Si es una covariante, entonces se realiza una *inferencia de límite inferior*.
* Si es una contravariante, se realiza una *inferencia de límite superior*.
* Si es una invariante, se realiza una *inferencia exacta*.
* En caso contrario, no se realiza ninguna inferencia.

#### Inferencias de límite superior

Una inferencia de límite superior de un tipo U a un tipo V se realiza de la siguiente manera:

* Si V es una de las variables de tipo sin fijar Xi, entonces U se agrega al conjunto de límites superiores para Xi.
* En caso contrario, los conjuntos V1…Vk y U1…Uk se determinan comprobando si se aplica alguno de los casos siguientes:
* U es un tipo de matriz U1[…]y V es un tipo de matriz V1[…]del mismo rango
* U es uno de IEnumerable<Ue>, ICollection<Ue> o IList<Ue> y V es un tipo de matriz unidimensional Ve[]
* U es el tipo U1? y V es el tipo V1?
* U es una clase, struct, interfaz o tipo de delegado construido C<U1…Uk> y V es una clase, struct, interfaz o tipo de delegado que es idéntico, hereda (directa o indirectamente) o implementa (directa o indirectamente) un tipo único C<V1…Vk>

(La restricción de “unicidad” significa que en el caso interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{}, no se hace inferencia cuando se infiere de C<U1> a V<Q>. Las inferencias no se hacen de U1 en X<Q> o Y<Q>).

Si ninguno de estos casos es aplicable, se realiza una inferencia de cada Ui en el correspondiente Vi, del modo siguiente:

* Si no se sabe si Ui es un tipo de referencia, se hace una *inferencia exacta*
* En caso contrario, si V es un tipo de matriz, se realiza una *inferencia de límite superior*.
* En caso contrario, si U es C<U1…Uk>, la inferencia depende del parámetro de tipo i-enésimo de C:
* Si es una covariante, se realiza una *inferencia de límite superior*.
* Si es una contravariante, se realiza una *inferencia de límite inferior*.
* Si es una invariante, se realiza una *inferencia exacta*.
* En caso contrario, no se realiza ninguna inferencia.

#### Fijar tipos

Una variable de tipo sin fijar Xi con un conjunto de límites se fija (fixed) de la siguiente forma:

* El conjunto de *tipos de candidatos* Uj se inicia como conjunto de todos los tipos en el conjunto de límites para Xi.
* A continuación, se examina cada límite para Xi: para cada límite exacto U de Xi se quitan todos los tipos Uj que no son idénticos a U del conjunto de candidatos. Para cada límite inferior U de Xi se quitan todos los tipos Uj para los que *no* hay conversión implícita desde U del conjunto de candidatos. Para cada límite superior U de Xi se quitan todos los tipos Uj desde los que *no* hay conversión implícita a U del conjunto de candidatos.
* Si entre los tipos de candidatos restantes Uj hay un tipo único V para el que hay una conversión implícita desde todos los demás tipos de candidatos, entonces Xi se fija en V.
* En caso contrario, la inferencia de tipos no se realiza correctamente.

#### Tipo de resultado inferido

El tipo de resultado inferido de una función anónima F se utiliza durante la inferencia de tipos y la resolución de sobrecargas. El tipo de resultado inferido sólo se puede determinar para una función anónima cuando se conocen todos los tipos de resultados, ya sea porque se dan explícitamente, a través de una conversión de función anónima, o inferidos durante una inferencia de tipos en una invocación de método genérico envolvente.

El tipo de resultado inferido se determina como sigue:

* Si el cuerpo de F es una expresión (expression) que tiene el tipo, el tipo de resultado inferido de F es el tipo de esa expresión.
* Si el cuerpo de F es un bloque (block) y el conjunto de expresiones en las instrucciones return del bloque tiene un tipo común mejor T (§7.5.2.14), el tipo de resultado inferido de F es T.
* En caso contrario, no se puede inferir un tipo de resultado para F.

El tipo devuelto inferido se determina como sigue:

* Si F es asincrónico y el cuerpo de F es una expresión clasificada como nada (§7.1) o un bloque de instrucciones donde ninguna instrucción devuelta tiene expresiones, el tipo devuelto inferido es System.Threading.Tasks.Task
* Si F es asincrónico y tiene un tipo de resultado inferido T, el tipo devuelto inferido es System.Threading.Tasks.Task<T>.
* Si F no es asincrónico y tiene un tipo de resultado inferido T, el tipo devuelto inferido es T.
* En caso contrario, no se puede inferir un tipo devuelto para F.

Como ejemplo de inferencia de tipos con funciones anónimas, considere el método de extensión Select declarado en la clase System.Linq.Enumerable:

namespace System.Linq  
{  
 public static class Enumerable  
 {  
 public static IEnumerable<TResult> Select<TSource,TResult>(  
 this IEnumerable<TSource> source,  
 Func<TSource,TResult> selector)  
 {  
 foreach (TSource element in source) yield return selector(element);  
 }  
 }  
}

Asumiendo que el espacio de nombres System.Linq se importó mediante una cláusula using, y dada una clase Customer con una propiedad Name de tipo string, el método Select se puede usar para seleccionar los nombres de una lista de clientes (customers):

List<Customer> customers = GetCustomerList();  
IEnumerable<string> names = customers.Select(c => c.Name);

La invocación del método de extensión (§7.6.5.2) de Select se procesa reescribiendo la invocación para una invocación de método estático:

IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);

Dado que los argumentos de tipo no se especificaron explícitamente, la inferencia de tipos se utiliza para inferir los argumentos de tipo. En primer lugar, el argumento customers se relaciona con el parámetro source, infiriendo T para que sea Customer. A continuación, mediante el proceso de inferencia de tipos de la función anónima anterior, se da a c el tipo Customer, y la expresión c.Name se relaciona con el tipo devuelto del parámetro selector, que infiere que S es string. Así, la invocación es equivalente a

Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)

y el resultado es de tipo IEnumerable<string>.

El siguiente ejemplo demuestra cómo una inferencia de tipo de la función anónima permite que la información de tipo “fluya” entre los argumentos de una invocación de método genérico. Dado el método:

static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {  
 return f2(f1(value));  
}

Escriba la inferencia de tipos para la invocación:

double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);

se procesa como sigue: en primer lugar, el argumento "1:15:30" se relaciona con el parámetro value, infiriendo X para que sea string. A continuación, al parámetro de la primera función anónima, s, se le da el tipo inferido string, y la expresión TimeSpan.Parse(s) se relaciona con el tipo devuelto f1, que infiere que Y es System.TimeSpan. Por último, al parámetro de la segunda función anónima, t, se le da el tipo inferido System.TimeSpan, y la expresión t.TotalSeconds se relaciona con el tipo devuelto f2, que infiere que Z es double. Así, el resultado de la invocación es de tipo double.

#### Inferencia de tipos para la conversión de grupos de métodos

De manera similar a las llamadas de métodos genéricos, la inferencia de tipos se debe aplicar también cuando un grupo de métodos M que contiene un método genérico se convierte a un tipo delegado D (§6.6) determinado. Dado un método

Tr M<X1…Xn>(T1 x1 … Tm xm)

y el grupo de métodos M que se está asignando al tipo delegado D, la tarea de la inferencia de tipos consiste en buscar los argumentos de tipo S1…Sn para que la expresión:

M<S1…Sn>

pase a ser compatible (§15.1) con D.

A diferencia del algoritmo de inferencia de tipos para llamadas de métodos genéricos, en este caso solo hay tipos de argumentos, no expresiones de argumentos. En particular, no hay funciones anónimas y por lo tanto no hay necesidad de varias fases de inferencia.

En lugar de eso, todas las Xi se consideran sin fijar y se realiza una inferencia de límite inferior desde cada tipo de argumento Uj de D al tipo de parámetro correspondiente Tj de M. Si no se encuentran límites para algún Xi, la inferencia de tipos da error. En caso contrario, todas las Xi se fijan en las Si correspondientes, que son el resultado de la inferencia de tipos.

#### Buscar el mejor tipo común de un conjunto de expresiones

En algunos casos, un tipo común necesita inferirse para un conjunto de expresiones. En particular, los tipos de elementos de las matrices con asignación de tipo implícita y los tipos de resultado de funciones anónimas con cuerpos de bloque (block) se encuentra de esta manera.

De forma intuitiva, dado un conjunto de expresiones E1…Em esta inferencia debería ser equivalente a la llamada de un método

Tr M<X>(X x1 … X xm)

con Ei como argumentos.

Más precisamente, la inferencia comienza con una variable sin fijar de tipo X. Las inferencias de tipos de resultado se hacen de cada Ei a X. Por último, X es fijo y, si no da error, el tipo resultante S es el mejor tipo común de resultado para las expresiones. Si el tipo S existe, las excepciones no tienen el mejor tipo común.

### Resolución de sobrecargas

La resolución de sobrecargas es un mecanismo del tiempo de enlace que selecciona el mejor miembro de función que puede invocarse dados una lista de argumentos y un conjunto de miembros de función candidatos. La resolución de sobrecargas selecciona el miembro de función al que se invoca en los contextos siguientes únicos en C#:

* Invocación de un método con nombre en una expresión de invocación (invocation-expression) (§7.6.5.1).
* Invocación de un constructor de instancia con nombre en una expresión de creación de objeto (object-creation-expression) (§7.6.10.1).
* Invocación de un descriptor de acceso a indizador mediante un acceso a elementos (element-access) (§7.6.6).
* Llamada de un operador predefinido o definido por el usuario al que se hace referencia en una expresión (§7.3.3 y §7.3.4).

Cada uno de estos contextos define el conjunto de miembros de función candidatos y la lista de argumentos a su propia manera, según se describe detalladamente en las secciones anteriormente citadas. Por ejemplo, el conjunto de candidatos para una invocación a un método no incluye métodos marcados con override (§7.4) y los métodos de una clase base no son candidatos si ningún método de una clase derivada es aplicable (§7.6.5.1).

Una vez identificados los miembros de función candidatos y la lista de argumentos, la selección del mejor miembro de función es la misma en todos los casos:

* Dado el conjunto de miembros de función candidatos aplicables, se localiza el mejor miembro de función del conjunto. Si el conjunto sólo contiene un miembro de función, ése es el mejor miembro. O bien, el mejor miembro de función es aquél que es mejor que todos los demás miembros de función con respecto a la lista de argumentos dada, a condición de que cada miembro de función se compare con todos los demás miembros aplicando las reglas de la §7.5.3.2. Si no hay exactamente un miembro de función mejor que todos los demás, la llamada de un miembro de función es ambigua y se produce un error durante el tiempo de enlace.

En las secciones siguientes se define el significado exacto de los términos miembro de función aplicable y mejor miembro de función.

#### Miembro de función aplicable

Se dice que un miembro de función es un miembro de función aplicable con respecto a una lista de argumentos A si todas las condiciones siguientes son verdaderas:

* Cada argumento de A corresponde a un parámetro en la declaración de miembro de función como se describe en §7.5.1.1, a cada parámetro le corresponde como máximo un argumento y cualquier parámetro al que no le corresponda ningún argumento es un parámetro opcional.
* Para cada argumento de A, el modo de pasar los parámetros del argumento (§7.5.1) es idéntico al del parámetro correspondiente (§1.6.6.1); además
* para un parámetro de valor o una matriz de parámetros, existe una conversión implícita (§6.1) de la expresión de argumento al tipo del parámetro correspondiente, o
* para un parámetro out o ref, el tipo de la expresión de argumento es idéntico al tipo del parámetro correspondiente. Después de todo, un parámetro ref u out es un alias para el argumento que se pasa.

Para un miembro de función que incluye una matriz de parámetros, si el miembro es aplicable por las reglas anteriores, se dice que es aplicable en su forma normal. Si un miembro de función que incluye una matriz de parámetros no es aplicable en su forma normal, el miembro puede ser aplicable en su forma expandida:

* La forma expandida se construye mediante el reemplazo de la matriz de parámetros en la declaración del miembro de función con cero o más parámetros de valor del tipo de los elementos de la matriz de parámetros, de forma que el número de argumentos de la lista de argumentos A coincida con el número total de parámetros. Si A tiene menos argumentos que el número de parámetros de tipo fijado de la declaración del miembro de función, la forma expandida del miembro de función no puede construirse y, por lo tanto, no es aplicable.
* O bien, la forma expandida es aplicable si, para cada argumento de A, el modo de pasar los parámetros del argumento es idéntico al modo de pasar los parámetros del parámetro correspondiente, y
* para un parámetro de valores fijados o parámetro de valores creados por la expansión, existe una conversión implícita (§6.1) del tipo del argumento al tipo del parámetro correspondiente, o
* para un parámetro ref u out, el tipo del argumento es idéntico al tipo del parámetro correspondiente

#### Mejor miembro de función

Con objeto de determinar el mejor miembro de función, se construye una lista de argumentos A con las expresiones de argumentos propiamente en el orden en que figuran en la lista de argumentos original.

Las listas de parámetros para cada miembro de función candidato se construyen del modo siguiente:

* Se usa la forma expandida si el miembro de función solo era aplicable en la forma expandida.
* Los parámetros opcionales sin argumentos correspondientes se quitan de la lista de parámetros
* Los parámetros se reordenan de forma que aparecen en la misma posición que el argumento correspondiente en la lista de argumentos.

Dada una lista de argumentos A con un conjunto de expresiones de argumento { E1, E2, ..., EN } y dos miembros de función aplicables MP y MQ con tipos de parámetro { P1, P2, ..., PN } y { Q1, Q2, ..., QN }, MP se define para que sea un miembro de función mejor que MQ si

* por cada argumento, la conversión implícita de EX en QX no es mejor que la conversión implícita de EX en PX, y
* al menos para un argumento, la conversión de EX en PX es mejor que la conversión de EX en QX.

Cuando se efectúa esta evaluación, si MP o MQ es aplicable en su forma expandida, PX o QX hace referencia a un parámetro en la forma expandida de la lista de parámetros.

En caso de que las secuencias de tipo de parámetro {P1, P2, …, PN} y {Q1, Q2, …, QN} sean equivalentes (es decir, cada Pi tiene una conversión de identidad en Qi correspondiente), se aplican las siguientes reglas de ruptura de lazos, en orden, para determinar el mejor miembro de función.

* Si MP es un método no genérico y MQ es un método genérico, MP es mejor que MQ.
* De lo contrario, si MP es aplicable en su forma normal y MQ tiene una matriz params y es aplicable solo en su forma expandida, entonces MP es mejor que MQ.
* De lo contrario, si MP tiene más parámetros declarados que MQ, MP es mejor que MQ. Esto puede ocurrir si ambos métodos tienen matrices params y solo son aplicables en sus formas expandidas.
* De lo contrario, si todos los parámetros de MP tienen un argumento correspondiente mientras que los argumentos predeterminados tienen que sustituirse para un parámetro opcional por lo menos en MQ entonces MP es mejor que MQ.
* De lo contrario, si MP tiene más tipos de parámetros específicos que MQ, MP es mejor que MQ. Supongamos que {R1, R2, …, RN} y {S1, S2, …, SN} representan los tipos de parámetros sin expandir y sin crear instancias de MP y MQ. Los tipos de parámetro de MP son más específicos que los de MQ si, para cada parámetro, RX no es menos específico que SX y, para al menos un parámetro, RX es más específico que SX:
* Un parámetro de tipo es menos específico que un parámetro que no es de tipo.
* De manera recursiva, un tipo construido es más específico que otro tipo construido (con el mismo número de argumentos de tipo) si al menos un argumento de tipo es más específico y no hay argumentos de tipo menos específicos que el argumento de tipo correspondiente.
* Un tipo de matriz es más específico que otro tipo de matriz (con el mismo número de dimensiones) si el tipo de elemento del primero es más específico que el tipo de elemento del segundo.
* En caso contrario, si un miembro es un operador de no elevación y el otro es un operador de elevación, el de no elevación es mejor.
* En caso contrario, ningún miembro de función es mejor.

#### Mejor conversión de expresiones

Dada una conversión implícita C1 que convierte de una expresión E a un tipo T1 y una conversión implícita C2 que convierte de una expresión E a un tipo T2, C1 es mejor conversión que C2 si es cierto algo de lo siguiente:

* E tiene un tipo S y existe una conversión de identidad de S en T1 pero no de S en T2
* E no es una función anónima y T1 es mejor destino de conversión que T2 (§7.5.3.5)
* E es una función anónima, T1 es un tipo delegado D1 o un tipo de árbol de expresiones Expression<D1>, T2 es un tipo delegado D2 o un tipo de árbol de expresiones Expression<D2> y se cumple una de las condiciones siguientes:
* D1 es mejor destino de conversión que D2
* D1 y D2 tienen listas de parámetros idénticas y se cumple algo de lo siguiente:
* D1 tiene un tipo devuelto Y1, y D2 tiene un tipo devuelto Y2, existe un tipo devuelto inferido X para E en el contexto de esa lista de parámetros (§7.5.2.12), y la conversión de X a Y1 es mejor que la conversión de X a Y2
* E es asincrónico, D1 tiene un tipo devuelto Task<Y1>, y D2 tiene un tipo devuelto Task<Y2>, existe un tipo devuelto inferido Task<X> para E en el contexto de esa lista de parámetros (§7.5.2.12), y la conversión de X a Y1 es mejor que la conversión de X a Y2
* D1 tiene un tipo devuelto Y, y D2 devuelve void

#### Mejor conversión de tipos

Dada una conversión C1 que convierte de un tipo S a un tipo T1 y una conversión C2 que convierte de un tipo S a un tipo T2, C1 es mejor conversión que C2 si se cumple algo de lo siguiente:

* Existe una conversión de identidad de S en T1 pero no de S en T2
* T1 es mejor destino de conversión que T2 (§7.5.3.5)

#### Mejor destino de conversión

Dados dos tipos diferentes T1 y T2, T1 es mejor destino de conversión que T2 si se cumple algo de lo siguiente:

* Existe una conversión implícita de T1 a T2 y no existe conversión implícita de T2 a T1
* T1 es un tipo entero con signo y T2 es un tipo entero sin signo. Concretamente:
* T1 es sbyte y T2 es byte, ushort, uint o ulong
* T1 es short y T2 es ushort, uint o ulong
* T1 es int y T2 es uint o ulong
* T1 es long y T2 es ulong

#### Sobrecargas en clases genéricas

Las firmas declaradas deben ser únicas, pero es posible que al sustituir argumentos de tipo se generen firmas idénticas. En estos casos, y gracias a las reglas de desempate de la resolución de sobrecarga antes mencionada, se escoge el miembro más especifico.

En los siguientes ejemplos se muestran sobrecargas válidas y no válidas según dicha regla:

interface I1<T> {...}

interface I2<T> {...}

class G1<U>  
{  
 int F1(U u); // Overload resulotion for G<int>.F1  
 int F1(int i); // will pick non-generic

void F2(I1<U> a); // Valid overload  
 void F2(I2<U> a);  
}

class G2<U,V>  
{  
 void F3(U u, V v); // Valid, but overload resolution for  
 void F3(V v, U u); // G2<int,int>.F3 will fail

void F4(U u, I1<V> v); // Valid, but overload resolution for   
 void F4(I1<V> v, U u); // G2<I1<int>,int>.F4 will fail

void F5(U u1, I1<V> v2); // Valid overload  
 void F5(V v1, U u2);

void F6(ref U u); // valid overload  
 void F6(out V v);  
}

### Comprobación en tiempo de compilación de resolución de sobrecarga dinámica

Aunque la resolución de sobrecarga de una operación enlazada dinámicamente tiene lugar en tiempo de ejecución, a veces es posible conocer en tiempo de ejecución la lista de miembros de función entre los que se elegirá una sobrecarga:

* Para una invocación de método (§7.6.5.1) de un tipo, o de un valor cuyo tipo estático no sea dynamic, el conjunto de métodos accesibles del grupo de métodos se conoce en tiempo de compilación.
* Para una expresión de creación de objetos (§7.6.10.1), el conjunto de constructores accesibles del tipo se conoce en tiempo de compilación.
* Para un acceso a indizador (§7.6.6.2), el conjunto de indizadores accesibles en el receptor se conoce en tiempo de compilación.

En estos casos se lleva a cabo una comprobación limitada en tiempo de compilación de cada miembro del conjunto de miembros de función conocido, para ver si es posible asegurarse de que no se invocan en tiempo de ejecución. Para cada miembro de función F se construye una lista de argumentos y parámetros modificados:

* En primer lugar, si F es un método genérico y se proporcionaron argumentos de tipo, estos se sustituyen por los parámetros de tipo de la lista de parámetros. Sin embargo, si no se proporcionaron argumentos de tipo, no se produce dicha sustitución.
* A continuación, se elide cualquier parámetro con un tipo abierto (es decir, contiene un parámetro de tipo; consulte §4.4.2), junto con sus parámetros correspondientes.

Para que F supere la comprobación, deben cumplirse las condiciones siguientes:

* La lista de parámetros modificados para F es aplicable a la lista de argumentos modificados de acuerdo con la sección §7.5.3.1.
* Todos los tipos construidos de la lista de parámetros modificados cumplen sus restricciones (§4.4.4).
* Si los parámetros de tipo de F se sustituyeron en el paso anterior, se cumplen sus restricciones.
* Si F es un método estático, el grupo de métodos no debe haberse obtenido de un acceso a miembro (member-access) de cuyo receptor se conozca en tiempo de compilación que es una variable o valor.
* Si F es un método de instancia, el grupo de métodos no debe haberse obtenido de un acceso a miembro (member-access) de cuyo receptor se conozca en tiempo de compilación que es un tipo.

Si ningún método candidato pasa esta prueba, se produce un error en tiempo de compilación.

### Invocación de miembros de función

En esta sección se explica el proceso que tiene lugar en tiempo de ejecución para invocar un miembro de función concreto. Se supone que un proceso de tiempo de enlace ya ha determinado el miembro concreto que se invoca, posiblemente por la aplicación de la resolución de sobrecargas a un conjunto de miembros de función candidatos.

Para la descripción del proceso de invocación, los miembros de función se dividen en dos categorías:

* Miembros de función estáticos. Son constructores de instancia, métodos estáticos, descriptores de acceso a propiedad estáticos y operadores definidos por el usuario. Los miembros de función estáticos siempre son no virtuales.
* Miembros de función de instancia. Son métodos de instancia, descriptores de acceso a propiedad de instancia y descriptores de acceso a indizador. Los miembros de función de instancia son virtuales o no virtuales, y siempre se les llama en una instancia concreta. La instancia se calcula mediante una expresión de instancia, y queda accesible dentro del miembro de función como this (§7.6.7).

El procesamiento en tiempo de ejecución de una invocación de miembro de función requiere los pasos siguientes, donde M es el miembro de función y, si M es un miembro de instancia, E es la expresión de instancia:

* Si M es un miembro de función estático:
* La lista de argumentos se evalúa según lo descrito en §7.5.1.
* Se invoca M.
* Si M es un miembro de función de instancia declarado en un tipo de valor (value-type):
* Se evalúa E. Si esta evaluación da lugar a una excepción, no se ejecutan nuevos pasos.
* Si E no está clasificado como variable, se crea una variable local temporal del tipo de E y se asigna el valor de E a la variable. E se reclasifica entonces como referencia a esa variable local temporal. La variable temporal está accesible como this dentro de M, pero no de otra forma. Por lo tanto, solamente si E es una variable verdadera es posible que el llamador observe los cambios que M efectúa en this.
* La lista de argumentos se evalúa según lo descrito en §7.5.1.
* Se invoca M. La variable a que hace referencia E pasa a ser la variable a que hace referencia this.
* Si M es un miembro de función de instancia declarado en un tipo de referencia (reference-type):
* Se evalúa E. Si esta evaluación da lugar a una excepción, no se ejecutan nuevos pasos.
* La lista de argumentos se evalúa según lo descrito en §7.5.1.
* Si el tipo de E es un tipo de valor (value-type), se realiza una conversión boxing (§4.3.1) para convertir E al tipo object, y E se considera de tipo object en los pasos siguientes. En este caso, M solo podría ser un miembro de System.Object.
* Se comprueba la validez del valor de E. Si el valor de E es null, se inicia una excepción System.NullReferenceException y no se ejecutan más pasos.
* La implementación del miembro de función que se va a invocar se determina de la siguiente forma:
* Si el tipo de tiempo de enlace de E es una interfaz, el miembro de función que hay que invocar es la implementación de M que proporciona el tipo de tiempo de ejecución de la instancia a la que hace referencia E. Este miembro de función se determina aplicando las reglas de asignación de interfaces (§13.4.4) para determinar la implementación de M proporcionada por el tipo en tiempo de ejecución de la instancia a la que hace referencia E.
* De lo contrario, si M es un miembro de función virtual, el miembro de función que hay que invocar es la implementación de M que proporciona el tipo de tiempo de ejecución de la instancia a la que hace referencia E. Este miembro de función se determina aplicando las reglas para determinar la implementación más derivada (§10.6.3) de M relativa al tipo en tiempo de ejecución de la instancia a la que hace referencia E.
* De lo contrario, M será un miembro de función no virtual y el miembro de función que habrá que invocar será el propio M.
* Se invoca la implementación del miembro de función determinada en el paso anterior. El objeto al que hace referencia E se convierte en el objeto al que hace referencia this.

#### Invocaciones en instancias de conversión boxing

Un miembro de función implementado en un tipo de valor (value-type) puede invocarse mediante una instancia de conversión boxing del tipo de valor en las situaciones siguientes:

* Si el miembro de función es un override de un método heredado del tipo object y se invoca mediante una expresión de instancia de tipo object.
* Si el miembro de función es una implementación de un miembro de función de interfaz y se invoca mediante una expresión de instancia de un tipo de interfaz (interface-type).
* Si se invoca el miembro de función a través de un delegado.

En estas situaciones, se considera que la instancia convertida mediante boxing contiene una variable de tipo de valor (value-type), y esta variable se convierte en la variable a la que se hace referencia con this dentro de la invocación del miembro de función. Esto, concretamente, significa que cuando se invoca un miembro de función en una instancia convertida mediante boxing, el miembro de función puede modificar el valor contenido en dicha instancia.

## Expresiones primarias

Las expresiones primarias incluyen las formas más simples de las expresiones.

primary-expression:   
primary-no-array-creation-expression  
array-creation-expression

primary-no-array-creation-expression:  
literal  
simple-name  
parenthesized-expression  
member-access  
invocation-expression  
element-access  
this-access  
base-access  
post-increment-expression  
post-decrement-expression  
object-creation-expression  
delegate-creation-expression  
anonymous-object-creation-expression  
typeof-expression  
 checked-expression  
unchecked-expression   
default-value-expression  
anonymous-method-expression

Las expresiones primarias se dividen entre expresiones de creación de matrices (array-creation-expressions) y expresiones primarias que no crean matrices (primary-no-array-creation-expression). Si las expresiones de creación de matrices se tratan de esta manera, en lugar de enumerarlas junto con todas las formas de expresiones simples, se permite que la gramática deshabilite potencialmente códigos confusos como

object o = new int[3][1];

que, de otra manera, podría interpretarse como

object o = (new int[3])[1];

### Literales

Una expresión primaria (primary-expression) compuesta por un literal (§2.4.4) se clasifica como un valor.

### Nombres sencillos

Un nombre simple (simple-name) está formado por un identificador; opcionalmente le sigue una lista de argumentos de tipo:

simple-name:  
identifier type-argument-listopt

Un nombre simple (simple-name) tiene la forma I o la forma I<A1, ..., AK>, donde I es un identificador único y <A1, ..., AK> es una lista de argumentos de tipo opcional (type-argument-list). Si no se especifica ninguna lista de argumentos de tipo (type-argument-list), se considera que K es cero. El nombre simple (simple-name) se evalúa y clasifica como sigue:

* Si K es cero y el nombre simple (simple-name) aparece dentro de un bloque (block) y si el espacio de declaración de variables locales (§3.3) del bloque (block) (o de un bloque contenedor) contiene una variable local, un parámetro o una constante con el nombre I, el nombre simple (simple-name) hace referencia a dicho parámetro, constante o variable local y se clasifica como una variable o un valor.
* Si K es cero y el nombre simple (simple-name) aparece dentro del cuerpo de una declaración de método genérico y si dicha declaración incluye un parámetro de tipo con el nombre I, el nombre simple (simple-name) hace referencia a dicho parámetro de tipo.
* De lo contrario, para cada tipo de instancia T (§10.3.1), empezando por el tipo de instancia de la declaración de tipo envolvente inmediata y continuando con el tipo de instancia de cada clase envolvente o declaración de struct (si las hubiera):
* Si K es cero y la declaración de T incluye un parámetro de tipo con el nombre I, el nombre simple (simple-name) hace referencia a dicho parámetro de tipo.
* De lo contrario, si la búsqueda de un miembro (§7.4) de I en T con argumentos de tipo K da una coincidencia:
* Si T es el tipo de instancia de la clase envolvente o del tipo struct inmediato y la búsqueda identifica uno o más métodos, el resultado es un grupo de métodos con una expresión de instancia asociada de this. Si se especificó una lista de argumentos de tipo, se utiliza para llamar a un método genérico (§7.6.5.1).
* De lo contrario, si T es el tipo de instancia de la clase envolvente o del tipo struct inmediato, si la búsqueda identifica un miembro de instancia, y si la referencia ocurre dentro del bloque (block) de un constructor de instancia, un método de instancia o un descriptor de acceso a instancia, el resultado es el mismo que un acceso a miembros (§7.6.4) de la forma this.I. Esto sólo puede ocurrir cuando K es cero.
* De lo contrario, el resultado es el mismo que un acceso a miembros (§7.6.4) con el formato T.I o T.I<A1, ..., AK>. En este caso, se produce un error durante el enlace si el nombre simple (simple-name) hace referencia a un miembro de instancia.
* De lo contrario, para cada espacio de nombres N, empezando por el espacio de nombres en el que se produce el nombre simple (simple-name), continuando por cada uno de los espacios de nombres envolventes (si los hubiera) y terminando por el espacio de nombres global, se irán evaluando los siguientes pasos hasta que se localice una entidad:
* Si K es cero e I es el nombre de un espacio de nombres en N:
* Si la ubicación donde tiene lugar el nombre simple (simple-name) tiene una declaración de espacio de nombres para N y la declaración de espacio de nombres contiene una directiva de alias extern (extern-alias-directive) o una directiva de alias using (using-alias-directive) que asocia el nombre I con un nombre de espacio de nombres o de tipo, el nombre simple (simple-name) es ambiguo y se genera un error en tiempo de compilación.
* De lo contrario, el nombre simple (simple-name) hace referencia al espacio de nombres denominado I en N.
* De lo contrario, si N contiene un tipo accesible con un nombre I y parámetros de tipo K:
* Si K es cero y la ubicación donde tiene lugar el nombre simple (simple-name) tiene una declaración de espacio de nombres para N y la declaración de espacio de nombres contiene una directiva de alias extern (extern-alias-directive) o una directiva de alias using (using-alias-directive) que asocia el nombre I con un nombre de espacio de nombres o de tipo, el nombre simple (simple-name) es ambiguo y se genera un error en tiempo de compilación.
* De lo contrario, el nombre de espacio de nombres o de tipo (namespace-or-type-name) hace referencia al tipo construido con los argumentos de tipo dados.
* De lo contrario, si la ubicación donde tiene lugar el nombre simple (simple-name) tiene lugar en una declaración de espacio de nombres para N:
* Si K es cero y la declaración de espacio de nombres contiene una directiva de alias extern (extern-alias-directive) o una directiva de alias using (using-alias-directive) que asocia el nombre I a un espacio de nombres o tipo importado, entonces el nombre simple (simple-name) hace referencia a dicho espacio de nombres o tipo.
* De lo contrario, si los espacios de nombres importados por las directivas using de espacio de nombres (using-namespace-directives) de la declaración del espacio de nombres contienen exactamente un único tipo con el nombre I y parámetros de tipo K, entonces el nombre simple (simple-name) hace referencia al tipo construido con los argumentos de tipo dados.
* De lo contrario, si los espacios de nombres importados por las directivas using de espacio de nombres (using-namespace-directives) de la declaración del espacio de nombres contienen más de un tipo con el nombre I y parámetros de tipo K, entonces el nombre simple (simple-name) será ambiguo y se producirá un error.

Tenga en cuenta que este paso es paralelo al paso correspondiente en el procesamiento de un nombre de espacio de nombres o de tipo (namespace-or-type-name) (§3.8).

* De lo contrario, el nombre simple (simple-name) no está definido y se produce un error en tiempo de compilación.

#### Significado invariable en bloques

Por cada repetición de un identificador dado como un nombre simple completo (simple-name) (sin lista de argumentos de tipo) en una expresión o declarador, dentro del espacio de declaración de la variable local (§3.3) que envuelve inmediatamente a dicha repetición, todas las demás repeticiones del mismo identificador como nombre simple completo (simple-name) en una expresión o declarador deben hacer referencia a la misma entidad. Esta regla asegura que el significado de un nombre siempre es el mismo dentro de un bloque, bloque switch, instrucción for-, foreach- o using-, o función anónima determinada.

El ejemplo

class Test  
{  
 double x;

void F(bool b) {  
 x = 1.0;  
 if (b) {  
 int x;  
 x = 1;  
 }  
 }  
}

produce un error durante la compilación porque x hace referencia a varias entidades dentro del bloque externo (cuya extensión incluye el bloque anidado de la instrucción if). Por el contrario, el ejemplo:

class Test  
{  
 double x;

void F(bool b) {  
 if (b) {  
 x = 1.0;  
 }  
 else {  
 int x;  
 x = 1;  
 }  
 }  
}

está permitido porque el nombre x nunca se utiliza en el bloque externo.

Debe tenerse en cuenta que la regla del significado invariable sólo se aplica a los nombres simples. Es perfectamente válido que el mismo identificador tenga un significado como nombre simple y otro distinto como operando derecho de un acceso a miembros (§7.6.4). Por ejemplo:

struct Point  
{  
 int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

El ejemplo anterior ilustra un modelo común de uso de los nombres de campos como nombres de parámetros en un constructor de instancia. En el ejemplo, los nombres simples x e y hacen referencia a los parámetros, pero ello no impide que expresiones de acceso a miembros como this.x y this.y obtengan acceso a los campos.

### Expresiones entre paréntesis

Una expresión entre paréntesis (parenthesized-expression) consiste en una expresión (expression) encerrada entre paréntesis.

parenthesized-expression:  
( expression )

Una expresión entre paréntesis (parenthesized-expression) se evalúa mediante la evaluación de la expresión (expression) contenida entre los paréntesis. Si la expresión (expression) entre paréntesis denota un espacio de nombres o un tipo, se produce un error durante la compilación. De lo contrario, el resultado de la expresión entre paréntesis (parenthesized-expression) es el resultado de la evaluación de la expresión (expression) contenida.

### Acceso a miembros

Un acceso a miembro (member-access) consiste en una expresión primaria (primary-expression), un tipo predefinido (predefined-type) o un miembro de alias calificado (qualified-alias-member) seguido por un símbolo (token) “.” y, después, por un identificador (identifier); opcionalmente, a este último puede seguirlo una lista de argumentos de tipo (type-argument-list).

member-access:  
primary-expression . identifier type-argument-listopt  
predefined-type . identifier type-argument-listopt  
qualified-alias-member . identifier type-argument-listopt

predefined-type: one of  
bool byte char decimal double float int long  
object sbyte short string uint ulong ushort

La producción qualified-alias-member se define en §9.7.

Un acceso a miembro (member-access) adopta la forma E.I o la forma E.I<A1, ..., AK>, donde E es una expresión primaria, I es un identificador único y <A1, ..., AK> es una lista de argumentos de tipo (type-argument-list) opcional. Si no se especifica ninguna lista de argumentos de tipo (type-argument-list), se considera que K es cero.

Un acceso a miembros (member-access) con la expresión primaria (primary-expression) de tipo dynamic se enlaza dinámicamente (§7.2.2). En este caso, el compilador clasifica el acceso a miembros como acceso de propiedad de tipo dynamic. Las reglas siguientes para determinar el significado del acceso a miembros (member-access) se aplican en tiempo de ejecución, usando el tipo en tiempo de ejecución en lugar del tipo en tiempo de compilación de la expresión primaria (primary-expression). Si esta clasificación en tiempo de ejecución lleva a un grupo de métodos, el acceso a miembros debe ser la expresión primaria (primary-expression) de una expresión de invocación (invocation-expression).

El acceso a miembro (member-access) se evalúa y clasifica como sigue:

* Si K es cero y E es un espacio de nombres y E contiene un espacio de nombres anidado con el nombre I, el resultado es dicho espacio de nombres.
* De lo contrario, si E es un espacio de nombres y E contiene un tipo accesible con el nombre I y parámetros de tipo K, el resultado es el tipo construido con los argumentos de tipo dados.
* Si E es un tipo predefinido (predefined-type) o una expresión primaria (primary-expression) clasificada como un tipo, si E no es un parámetro de tipo y si una búsqueda de miembros (§7.4) de I en E con parámetros de tipo K produce una coincidencia, entonces E.I se evalúa y clasifica como sigue:
* Si I identifica un tipo, el resultado es el tipo construido con los argumentos de tipo dados.
* Si I identifica uno o varios métodos, entonces el resultado es un grupo de métodos sin una expresión de instancia asociada. Si se especificó una lista de argumentos de tipo, se utiliza para llamar a un método genérico (§7.6.5.1).
* Si I identifica una propiedad static, entonces el resultado es un acceso a propiedad sin una expresión de instancia asociada.
* Si I identifica un campo static:
* Si el campo es readonly y la referencia ocurre fuera del constructor estático de la clase o struct donde está declarado el campo, entonces el resultado es un valor, concretamente, el valor del campo estático I en E.
* En caso contrario, el resultado es una variable, en concreto, el campo estático I en E.
* Si I identifica un evento static:
* Si la referencia ocurre dentro de la clase o struct donde está declarado el evento, y éste se ha declarado sin declaraciones de descriptor de acceso a evento (event-accessor-declarations) (§10.8), entonces E.I se procesa exactamente como si I fuera un campo estático.
* O bien, el resultado es un acceso a evento sin una expresión de instancia asociada.
* Si I identifica una constante, entonces el resultado es un valor, en concreto el valor de la constante.
* Si I identifica un miembro de enumeración, entonces el resultado es un valor, a saber, el valor de dicho miembro de enumeración.
* De lo contrario, E.I será una referencia de miembro no válida y se producirá un error en tiempo de compilación.
* Si E es un acceso a propiedad, un acceso a indizador, una variable o un valor, cuyo tipo es T, y una búsqueda de miembros (§7.4) de I en T con argumentos de tipo K produce una coincidencia, entonces E.I se evalúa y clasifica como sigue:
* En primer lugar, si E es un acceso a propiedad o a indizador, entonces se obtiene el valor del acceso a propiedad o a indizador (§7.1.1) y E se reclasifica como un valor.
* Si I identifica uno o varios métodos, entonces el resultado es un grupo de métodos con una expresión de instancia asociada de E. Si se especificó una lista de argumentos de tipo, se utiliza para llamar a un método genérico (§7.6.5.1).
* Si I identifica una propiedad de instancia, entonces el resultado es un acceso a propiedad con una expresión de instancia asociada de E.
* Si T es un tipo de clase (class-type) e I identifica un campo de instancia de dicho tipo de clase:
* Si el valor de E es null, se inicia una excepción System.NullReferenceException.
* De lo contrario, si el campo es readonly y la referencia ocurre fuera de un constructor de instancia de la clase donde está declarado el campo, entonces el resultado es un valor, concretamente, el valor del campo I en el objeto al que hace referencia E.
* De lo contrario, el resultado es una variable, en concreto, el campo I en el objeto al que hace referencia E.
* Si T es un tipo struct (struct-type) e I identifica un campo de instancia de dicho tipo:
* Si E es un valor o si el campo es readonly y la referencia ocurre fuera de un constructor de instancia del struct donde está declarado el campo, entonces el resultado es un valor, concretamente, el valor del campo I en la instancia de struct dada por E.
* O bien, el resultado es una variable, en concreto, el campo I en la instancia de struct dada por E.
* Si I identifica un evento de instancia:
* Si la referencia ocurre dentro de la clase o struct donde está declarado el evento, y éste se ha declarado sin declaraciones de descriptor de acceso a evento (event-accessor-declarations) (§10.8), y la referencia no aparece como lado izquierdo de un operador += o -=, entonces E.I se procesa exactamente como si I fuera un campo de instancia.
* De lo contrario, el resultado es un acceso a evento con una expresión de instancia asociada de E.
* En caso contrario, se realiza un intento de procesar E.I como una invocación del método de extensión (§7.6.5.2). Si esto da error, E.I será una referencia de miembro no válida y se producirá un error en tiempo de enlace.

#### Nombres simples y nombres de tipos idénticos

En un acceso a miembros con la estructura E.I, si E es un solo identificador y el significado de E como nombre simple (simple-name) (§7.6.2) es una constante, un campo, una propiedad, una variable local o un parámetro del mismo tipo que el significado de E como nombre de tipo (type-name) (§3.8), entonces se permiten los dos posibles significados de E. Los dos posibles significados de E.I nunca son ambiguos, puesto que I necesariamente tiene que ser un miembro del tipo E en los dos casos. Es decir, la regla sencillamente permite el acceso a los miembros estáticos y tipos anidados de E donde, de otra forma, se habría producido un error en tiempo de compilación. Por ejemplo:

struct Color  
{  
 public static readonly Color White = new Color(...);  
 public static readonly Color Black = new Color(...);

public Color Complement() {...}  
}

class A  
{  
 public Color Color; // Field Color of type Color

void F() {  
 Color = Color.Black; // References Color.Black static member  
 Color = Color.Complement(); // Invokes Complement() on Color field  
 }

static void G() {  
 Color c = Color.White; // References Color.White static member  
 }  
}

Dentro de la clase A, estas repeticiones del identificador Color que hacen referencia al tipo Color están subrayadas, y las que hacen referencia al campo Color no.

#### Ambigüedades gramaticales

Las producciones para nombre simple (simple-name) (§7.6.2) y acceso a miembro (member-access) (§7.6.4) pueden dar lugar a ambigüedades en la gramática de las expresiones. Por ejemplo, la instrucción:

F(G<A,B>(7));

se puede interpretar como una llamada a F con dos argumentos, G < A y B > (7). También se puede interpretar como una llamada a F con un argumento, que es una llamada a un método genérico G con dos argumentos tipo y un argumento normal.

Si se puede analizar (en contexto) una secuencia de símbolos como un nombre simple (simple-name) (§7.6.2), un acceso a miembro (member-access) (§7.6.4) o un acceso a miembros de puntero (pointer-member-access) (§18.5.2) que termine con una lista de argumentos de tipo (type-argument-list) (§4.4.1), se examinará el token que aparece inmediatamente después del token de cierre >. Si se trata de

( ) ] } : ; , . ? == != | ^

la lista de argumentos de tipo (type-argument-list) se retiene como parte del nombre simple (simple-name), del acceso a miembro (member-access) o del acceso a miembro de puntero (pointer-member-access) y se descarta cualquier otro análisis posible de la secuencia de token. De lo contrario, lista de argumentos de tipo (type-argument-list) no se considera parte del nombre simple (simple-name), del acceso a miembro (member-access) o del acceso a miembro de puntero (pointer-member-access) incluso si no hay otro análisis posible de la secuencia de tokens. Tenga en cuenta que estas reglas no se aplican al analizar una lista de argumentos de tipo (type-argument-list) en un espacio de nombres o de tipo (namespace-or-type-name) (§3.8). La instrucción

F(G<A,B>(7));

se interpretará, de acuerdo con esta regla, como una llamada a F con un argumento, que es una llamada a un método genérico G con dos argumentos de tipo y un argumento normal. Cada una de las instrucciones

F(G < A, B > 7);  
F(G < A, B >> 7);

se interpretarán como una llamada a F con dos argumentos. La instrucción

x = F < A > +y;

se interpretará como un operador menor que, mayor que y unario de signo más, como si la instrucción se hubiese escrito x = (F < A) > (+y), en lugar de como un nombre simple (simple-name) con una lista de argumentos de tipo (type-argument-list) seguida de un operador binario de signo más. En la instrucción

x = y is C<T> + z;

los tokens C<T> se interpretan como un nombre de espacio de nombres o de tipo (namespace-or-type-name) con una lista de argumentos de tipo (type-argument-list).

### Expresiones de invocación

Una expresión de invocación o llamada (invocation-expression) se utiliza para llamar a un método.

invocation-expression:  
primary-expression ( argument-listopt )

Una expresión de invocación (invocation-expression) se enlaza dinámicamente (§7.2.2) si es cierto algo de lo siguiente:

* La expresión primaria (primary-expression) tiene un tipo en tiempo de compilación dynamic.
* Al menos un argumento de la lista de argumentos (argument-list) opcional tiene un tipo en tiempo de compilación dynamic y la expresión primaria (primary-expression) no tiene un tipo de delegado.

En este caso, el compilador clasifica la expresión de invocación (invocation-expression) como un valor de tipo dynamic. Las reglas siguientes para determinar el significado de la expresión de invocación (invocation-expression) se aplican en tiempo de ejecución, usando el tipo en tiempo de ejecución en lugar del tipo en tiempo de compilación de la expresión primaria (primary-expression) y los argumentos que tienen tipo dynamic en tiempo de compilación. Si la expresión primaria no tiene el tipo en tiempo de compilación dynamic, el método de invocación experimenta una comprobación limitada en tiempo de compilación como se describe en §7.5.4.

La expresión primaria (primary-expression) de una expresión de invocación (invocation-expression) debe ser un grupo de métodos o un valor de un tipo delegado (delegate-type). Si la expresión primaria (primary-expression) es un grupo de métodos, la expresión de invocación (invocation-expression) es una invocación de método (§7.6.5.1). Si la expresión primaria (primary-expression) es un valor de un tipo delegado (delegate-type), la expresión de invocación (invocation-expression) es una invocación de delegado (§7.6.5.3). Si la expresión primaria (primary-expression) no es un grupo de métodos ni un valor de un tipo de delegado (delegate-type), se produce un error en tiempo de enlace.

La lista de argumentos (argument-list) opcional (§7.5.1) proporciona valores o referencias de variables para los parámetros del método.

El resultado de evaluar una expresión de invocación (invocation-expression) se clasifica como sigue:

* Si la expresión de invocación (invocation-expression) invoca un método o un delegado que devuelve void, el resultado es nada. Una expresión que se clasifica como nada sólo está permitida en el contexto de una expresión de instrucción (statement-expression) (§8.6) o como el cuerpo de una expresión lambda (lambda-expression) (§7.15). En caso contrario, se producirá un error en tiempo de enlace.
* En caso contrario, el resultado es un valor del tipo que devuelve el método o el delegado.

#### Invocaciones de método

Para una invocación de método, la expresión primaria (primary-expression) de la expresión de invocación (invocation-expression) debe ser un grupo de métodos. El grupo de métodos identifica el método que se invoca o el conjunto de métodos sobrecargados entre los cuales se elige un método concreto para invocar. En el último caso, la determinación del método concreto que se invoca se basa en el contexto suministrado por los tipos de los argumentos de lista de argumentos (argument-list).

El procesamiento en tiempo de enlace de una invocación de método con la estructura M(A), donde M es un grupo de métodos (que posiblemente incluya una lista de argumentos de tipo [type-argument-list]) y A es una lista de argumentos (argument-list) opcional, se compone de los siguientes pasos:

* Se construye el conjunto de métodos candidatos para la invocación del método. Para cada método F asociado con el grupo de métodos M:
* Si F es un método no genérico, F es un candidato cuando:
* M no tiene una lista de argumentos de tipo y
* F es aplicable en lo que concierne a A (§7.5.3.1).
* Si F es un método genérico y M no tiene una lista de argumentos de tipo, F es un candidato cuando:
* La inferencia de tipo (§7.5.2) se realiza correctamente, deduciendo una lista de argumentos de tipo para la llamada y
* Una vez inferidos, los argumentos de tipo se sustituyen por los parámetros de tipo del método correspondientes, todos los tipos construidos en la lista de parámetros de F cumplen sus restricciones (§4.4.4) y la lista de parámetros de F es aplicable en lo que concierne a A (§7.5.3.1).
* Si F es un método genérico y M incluye una lista de argumentos de tipo, F es un candidato cuando:
* F tiene el mismo número de parámetros de tipo del método que el suministrado en la lista de argumentos de tipo y
* Los argumentos de tipo se sustituyen por los parámetros de tipo del método correspondientes, todos los tipos construidos en la lista de parámetros de F cumplen sus restricciones (§4.4.4) y la lista de parámetros de F es aplicable en lo que concierne a A (§7.5.3.1).
* El conjunto de métodos candidatos se limita a métodos procedentes de los tipos más derivados: para cada método C.F del conjunto, donde C es el tipo en el que se declara el método F, se quitan del conjunto todos los métodos declarados en un tipo base de C. Si C es un tipo de clase diferente de object, se quitan del conjunto todos los métodos declarados en un tipo de interfaz. (Esta última regla solo tiene efecto cuando el grupo de métodos es el resultado de una búsqueda de miembros en un parámetro de tipo con una clase base efectiva diferente de object y un conjunto de interfaces efectivas que no estén vacías).
* Si el conjunto resultante de métodos candidatos está vacío, no se realizan los pasos siguientes, y en lugar de eso se realiza un intento de procesar la invocación como una invocación del método de extensión (§7.6.5.2). Si esto da error, significa que no existen métodos aplicables, y se produce un error en tiempo de enlace.
* El mejor método del conjunto de métodos candidatos se identifica mediante las reglas de resolución de sobrecargas de §7.5.3. Si no puede identificarse un método individual mejor, la invocación del método es ambigua, y se produce un error en tiempo de enlace. Al realizar la resolución de sobrecarga, se tienen en cuenta los parámetros de un método genérico después de sustituir los argumentos de tipo (suministrados o inferidos) para los parámetros de tipo del método correspondientes.
* Se realiza la **validación final** del método mejor elegido:
* El método se valida en el contexto del grupo de métodos: si el mejor método es un método estático, el grupo de métodos debe haberse obtenido de un nombre simple (simple-name) o de un acceso a miembro (member-access) mediante un tipo. Si el método mejor es un método de instancia, el grupo de métodos debe haberse obtenido de un nombre simple (simple-name), de un acceso a miembro (member-access) a través de una variable o un valor, o de un acceso a base (base-access). Si no se cumple ninguna de estas dos condiciones, se producirá un error en tiempo de enlace.
* Si el método mejor es un método genérico, los argumentos de tipo (suministrados o inferidos) se cotejan con las restricciones (§4.4.4) declaradas en el método genérico. Si un argumento de tipo no satisface las restricciones correspondientes del parámetro de tipo, se generará un error en tiempo de enlace.

Una vez seleccionado un método y validado en tiempo de enlace por los pasos anteriores, se procesa la invocación real en tiempo de ejecución conforme a las reglas de invocación de un miembro de función explicadas en §7.5.4.

El efecto intuitivo de las reglas de resolución explicadas es como sigue: para localizar el método concreto invocado por una invocación de método, se empieza con el tipo indicado por la invocación del método y se continúa en la cadena de la herencia hasta encontrar por lo menos una declaración de método aplicable, accesible y no de invalidación. Después, se lleva a cabo la inferencia de tipos y la resolución de sobrecargas en el conjunto de métodos aplicables, accesibles y de no invalidación declarados en dicho tipo y se invoca el método seleccionado por este procedimiento. Si no se encuentra ningún método, pruebe en su lugar a procesar la invocación como una invocación del método de extensión.

#### Invocaciones del método de extensión

En una invocación de método (§7.5.5.1) una de las estructuras:

expr . identifier ( )

expr . identifier ( args )

expr . identifier < typeargs > ( )

expr . identifier < typeargs > ( args )

si el procesamiento normal de la invocación no encuentra métodos aplicables, se realiza un intento de procesar la estructura como una invocación del método de extensión. Si expr o alguno de los args tiene un tipo dynamic en tiempo de compilación, los métodos de extensión no se aplicarán.

El objetivo es buscar el mejor nombre de tipo (type-name) C, para que se produzca la invocación del método estático correspondiente:

C . identifier ( expr )

C . identifier ( expr , args )

C . identifier < typeargs > ( expr )

C . identifier < typeargs > ( expr , args )

Un método de extensión Ci.Mj es candidato si:

* Ci es una clase no genérica y no anidada
* El nombre de Mj es identifier
* Mj es accesible y aplicable cuando se aplica a los argumentos como un método estático tal y como se muestra anteriormente
* Existe una conversión de identidad, de referencia o boxing implícita desde expr al tipo del primer parámetro de Mj.

La búsqueda de C se realiza de la siguiente forma:

* Empezando por la declaración del espacio de nombres envolvente más cercana, siguiendo por cada declaración de espacio de nombres envolvente, y terminando por la unidad de compilación contenedora, se realizan sucesivos intentos de encontrar un conjunto de candidatos para los métodos de extensión:
* Si el espacio de nombres o la unidad de compilación dada contiene directamente declaraciones de tipo no genérico Ci con métodos de extensión candidatos Mj, el conjunto de dichos métodos de extensión es el conjunto candidato.
* Si los espacios de nombres importado usando directivas de espacio de nombres en el espacio de nombres o la unidad de compilación dada contienen directamente declaraciones de tipo no genérico Ci con métodos de extensión candidatos Mj, el conjunto de dichos métodos de extensión es el conjunto candidato.
* Si no se encuentra ningún conjunto de candidatos en la unidad de compilación o la declaración de espacios de nombres envolventes, se produce un error durante la compilación.
* En caso contrario, se aplica la resolución de sobrecargas en el conjunto de candidatos tal como se describe en la sección (§7.5.3). Si no se encuentra un método individual mejor, se produce un error en tiempo de compilación.
* C es el tipo dentro del que se encuentra declarado el mejor método como un método de extensión.

Cuando se utiliza C como destino, la llamada a método se procesa como una invocación de método estático (§7.5.4).

Las reglas anteriores significan que los métodos de instancia tienen prioridad sobre los métodos de extensión, que los métodos de extensión disponibles en declaraciones de espacios de nombres internos tienen prioridad sobre los métodos de extensión disponibles en las declaraciones de espacios de nombres externos, y que los métodos de extensión declarados directamente en un espacio de nombres tienen prioridad sobre los importados dentro del mismo espacio de nombres con una directiva using de espacio de nombres. Por ejemplo:

public static class E  
{  
 public static void F(this object obj, int i) { }

public static void F(this object obj, string s) { }  
}

class A { }

class B  
{  
 public void F(int i) { }  
}

class C  
{  
 public void F(object obj) { }  
}

class X  
{  
 static void Test(A a, B b, C c) {  
 a.F(1); // E.F(object, int)  
 a.F("hello"); // E.F(object, string)

b.F(1); // B.F(int)  
 b.F("hello"); // E.F(object, string)

c.F(1); // C.F(object)  
 c.F("hello"); // C.F(object)  
 }  
}

En el ejemplo, el método de B tiene prioridad sobre el primer método de extensión, y el método de C tiene prioridad sobre ambos métodos de extensión.

public static class C  
{  
 public static void F(this int i) { Console.WriteLine("C.F({0})", i); }  
 public static void G(this int i) { Console.WriteLine("C.G({0})", i); }  
 public static void H(this int i) { Console.WriteLine("C.H({0})", i); }  
}

namespace N1  
{  
 public static class D  
 {  
 public static void F(this int i) { Console.WriteLine("D.F({0})", i); }  
 public static void G(this int i) { Console.WriteLine("D.G({0})", i); }  
 }  
}

namespace N2  
{  
 using N1;

public static class E  
 {  
 public static void F(this int i) { Console.WriteLine("E.F({0})", i); }  
 }

class Test  
 {  
 static void Main(string[] args)  
 {  
 1.F();  
 2.G();  
 3.H();  
 }  
 }  
}

El resultado de este ejemplo es:

E.F(1)  
D.G(2)  
C.H(3)

D.G tiene prioridad sobre C.G, y E.F tiene prioridad sobre ambos, D.F y C.F.

#### Invocaciones de delegados

Para una invocación de delegado, la expresión primaria (primary-expression) de la expresión de invocación (invocation-expression) debe ser un valor de un tipo delegado (delegate-type). Además, considerando que el tipo delegado (delegate-type) debe ser un miembro de función con la misma lista de parámetros que el tipo delegado (delegate-type), este debe ser aplicable (§7.5.3.1) con respecto a la lista de argumentos (argument-list) de la expresión de invocación (invocation-expression).

El procesamiento en tiempo de ejecución de una invocación de delegado con la estructura D(A), donde D es una expresión primaria (primary-expression) de un tipo delegado (delegate-type) y A es una lista de argumentos (argument-list) opcional, se compone de los siguientes pasos:

* Se evalúa D. Si esta evaluación da lugar a una excepción, no se ejecutan nuevos pasos.
* Se comprueba la validez del valor de D. Si el valor de D es null, se inicia una excepción System.NullReferenceException y no se ejecutan más pasos.
* En otro caso, D es una referencia a una instancia de delegado. Las llamadas a miembros de función (§7.5.4) se realizan en cada una de las entidades invocables de la lista de llamadas del delegado. En el caso de las entidades invocables y que constan de una instancia y de un método de instancia, la instancia para la invocación es la contenida en la entidad a la que se puede llamar.

### Acceso a elementos

Un acceso a elementos (element-access) consta de una expresión primaria sin creación de matrices (primary-no-array-creation-expression), seguida de un token “[”, una lista de argumentos (argument-list) y un token “]”. La lista de argumentos (argument-list) consta de uno o más argumentos (argument), separados por comas.

element-access:  
primary-no-array-creation-expression [ argument-list ]

La lista de argumentos (argument-list) de un acceso a elementos (element-access) no puede contener argumentos ref ni out.

Un acceso a elementos (element-access) se enlaza dinámicamente (§7.2.2) si es cierto algo de lo siguiente:

* La expresión primaria sin creación de matriz (primary-no-array-creation-expression) tiene un tipo dynamic en tiempo de compilación.
* Al menos una expresión de la lista de argumentos (argument-list) tiene un tipo dynamic en tiempo de compilación y la expresión primaria sin creación de matriz no tiene un tipo de matriz.

En este caso, el compilador clasifica el acceso a elementos (element-access) como un valor de tipo dynamic. Las reglas siguientes para determinar el significado del acceso a elementos (element-access) se aplican en tiempo de ejecución, usando el tipo en tiempo de ejecución en lugar del tipo en tiempo de compilación de la expresión primaria sin creación de matriz (primary-no-array-creation-expression) y las expresiones de la lista de argumentos (argument-list) que tienen un tipo dynamic en tiempo de compilación. Si la expresión primaria sin creación de matrices no tiene un tipo dynamic en tiempo de compilación, el acceso a elementos experimenta una comprobación limitada en tiempo de compilación como se describe en §7.5.4.

Si la expresión primaria sin creación de matrices (primary-no-array-creation-expression) de un acceso a elementos (element-access) es un valor de tipo matricial (array-type), el acceso a elementos (element-access) es un acceso a matriz (§7.6.6.1). De lo contrario, la expresión primaria sin creación de matrices (primary-no-array-creation-expression) debe ser una variable o un valor de un tipo de clase, struct o interfaz que tenga uno o más miembros indizadores, en cuyo caso el acceso a elementos (element-access) es un acceso a indizador (§7.6.6.2).

#### Acceso a matrices

Para un acceso a matrices, la expresión primaria sin creación de matrices (primary-no-array-creation-expression) del acceso a elementos (element-access) debe ser un valor de tipo matricial (array-type). Además, la lista de argumentos de un acceso a matrices no puede contener argumentos con nombre. El número de expresiones de la lista de argumentos debe ser la misma que el rango del tipo matricial, y cada expression debe ser de tipo int, uint, long, ulong, o debe ser implícitamente convertible en uno o varios de esos tipos.

El resultado de la evaluación de un acceso a matriz es una variable del tipo de elementos de la matriz, es decir, el elemento de matriz seleccionado por los valores de las expresiones en la lista de argumentos (argument-list).

El procesamiento en tiempo de ejecución de un acceso a matriz con la estructura P[A] (donde P es una expresión primaria sin creación de matrices (primary-no-array-creation-expression) de un tipo matricial (array-type) y A es una lista de argumentos (argument-list) se compone de los siguientes pasos:

* Se evalúa P. Si esta evaluación da lugar a una excepción, no se ejecutan nuevos pasos.
* Las expresiones de índice de la lista de argumentos (argument-list) se evalúan por orden, de izquierda a derecha. Después de la evaluación de cada expresión de índice, se realiza una conversión implícita (§6.1) a uno de los tipos siguientes: int, uint, long, ulong. Se elige el primer tipo de esta lista para el cual existe una conversión implícita. Por ejemplo, si la expresión de índice es de tipo short, se lleva a cabo una conversión implícita a int, puesto que son posibles conversiones implícitas de short a int y de short a long. Si la evaluación de una expresión de índice o de la conversión implícita posterior causa una excepción, no se evalúan otras expresiones de índice y no se ejecutan nuevos pasos.
* Se comprueba la validez del valor de P. Si el valor de P es null, se inicia una excepción System.NullReferenceException y no se ejecutan más pasos.
* Se compara otra vez el valor de cada expresión de lista de argumentos con los límites reales de cada dimensión de la instancia matricial a que hace referencia P. Si uno o más valores no están comprendidos en el intervalo, se inicia una excepción System.IndexOutOfRangeException y no se ejecutan nuevos pasos.
* Se calcula la ubicación del elemento de matriz dada por la expresión o expresiones de índice, que pasa a ser el resultado del acceso a matriz.

#### Acceso al indizador

Para un acceso a indizador, la expresión primaria sin creación de matrices (primary-no-array-creation-expression) del acceso a elementos (element-access) debe ser una variable o un valor de un tipo de clase, struct o interfaz, y este tipo debe implementar uno o más indizadores que sean aplicables con respecto a la lista de argumentos (argument-list) del acceso a elementos (element-access).

El procesamiento en tiempo de enlace de un acceso a indizador de la forma P[A], donde P es una expresión primaria sin creación de matrices (primary-no-array-creation-expression) de un tipo T de clase, struct o interfaz, y A es una lista de argumentos (argument-list), se compone de los siguientes pasos:

* Se construye el conjunto de indizadores suministrados por T. El conjunto está formado por todos los indizadores declarados en T o un tipo base de T que no son declaraciones override y están accesibles en el contexto actual (§3.5).
* El conjunto se reduce a aquellos indizadores que sean aplicables y no estén ocultos por otros indizadores. Las reglas siguientes se aplican a todos los indizadores S.I del conjunto, donde S es el tipo en el que se declara el indizador I:
* Si I no es aplicable con respecto a A (§7.5.3.1), entonces I se quita del conjunto.
* Si I es aplicable con respecto a A (§7.5.3.1), entonces todos los indizadores declarados en un tipo base de S se quitan del conjunto.
* Si I es aplicable con respecto a A (§7.5.3.1) y S es un tipo de clase distinto de object, todos los indizadores declarados en una interfaz se quitan del conjunto.
* Si el conjunto resultante de indizadores candidatos está vacío, entonces no existen indizadores aplicables, y se produce un error en tiempo de enlace.
* El mejor indizador del conjunto de indizadores candidatos se identifica mediante las reglas de resolución de sobrecargas de §7.5.3. Si no puede identificarse un indizador individual mejor, el acceso a indizador es ambiguo y se produce un error en tiempo de enlace.
* Las expresiones de índice de la lista de argumentos (argument-list) se evalúan por orden, de izquierda a derecha. El resultado de procesar el acceso a indizador es una expresión clasificada como un acceso a indizador. La expresión de acceso a indizador hace referencia al indizador determinado en el paso anterior, y tiene una expresión de instancia asociada de P y una lista de argumentos asociada de A.

Dependiendo del contexto en que se utilice, un acceso a indizador provoca la invocación del descriptor de acceso get (get-accessor) o del descriptor de acceso set (set-accessor) del indizador. Si el acceso a indizador es el destino de una asignación, se invoca el descriptor de acceso set (set-accessor) para asignar un nuevo valor (§7.17.1). En todos los demás casos, se invoca el descriptor de acceso get (get-accessor) para obtener el valor actual (§7.1.1).

### Acceso a this

Un acceso a this (this-access) consta de la palabra reservada this.

this-access:  
this

El acceso a this (this-access) solo se permite en el bloque (block) de un constructor de instancia, un método de instancia o un descriptor de acceso a una instancia. Tiene uno de los siguientes significados:

* Cuando se utiliza this en una expresión primaria (primary-expression) dentro de un constructor de instancia de una clase, ésta se clasifica como un valor. El tipo del valor es el tipo de instancia (§10.3.1) de la clase dentro de la cual ocurre la utilización y el valor es una referencia al objeto que se construye.
* Cuando se utiliza this en una expresión primaria (primary-expression) dentro de un método de instancia o de un descriptor de acceso a instancia de una clase, esta se clasifica como un valor. El tipo del valor es el tipo de instancia (§10.3.1) de la clase dentro de la cual ocurre la utilización, y el valor es una referencia al objeto para el cual se invoca el método o el descriptor de acceso.
* Cuando se utiliza this en una expresión primaria (primary-expression) dentro de un constructor de instancia de un struct, esta se clasifica como una variable. El tipo de la variable es el tipo de instancia (§10.3.1) del struct dentro de la cual ocurre la utilización y la variable representa el struct que se construye. La variable this de un constructor de instancia de un struct se comporta exactamente igual que un parámetro out del tipo struct; esto significa concretamente que la variable debe estar asignada definitivamente en todas las rutas de ejecución del constructor de instancia.
* Cuando se utiliza this en una expresión primaria (primary-expression) dentro de un método de instancia o un descriptor de acceso a instancia de un struct, ésta se clasifica como una variable. El tipo de la variable es el tipo de instancia (§10.3.1) del struct donde se da el uso.
* Si el método o el descriptor de acceso no es un iterador (§10.14), la variable this representa el struct para el que se invoca el método o el descriptor de acceso, y se comporta exactamente igual que el parámetro ref del tipo struct.
* Si el método o el descriptor de acceso no es un iterador, la variable this representa una copia del struct para el que se invoca el método o el descriptor de acceso, y se comporta exactamente igual que el parámetro de valor del tipo struct.

El uso de this en una expresión primaria (primary-expression) dentro de un contexto distinto de los mencionados anteriormente, produce un error en tiempo de compilación. En particular, no es posible hacer referencia a this en un método estático, en un descriptor de acceso a una propiedad estática o en un inicializador de variable (variable-initializer) de una declaración de campo.

### Acceso a bases

Un acceso a base (base-access) está formado por la palabra reservada base seguida de un token “.” y un identificador, o de una lista de argumentos (argument-list) encerrada entre corchetes:

base-access:  
base . identifier  
base [ argument-list ]

Un acceso a base (base-access) permite obtener acceso a miembros de clase base que están ocultos por miembros de nombres similares en la clase o struct actuales. El acceso a base (base-access) solo se permite en el bloque (block) de un constructor de instancia, un método de instancia o un descriptor de acceso a una instancia. Si base.I ocurre en una clase o un struct, I debe denotar un miembro de la clase base de dicha clase o struct. De igual manera, si base[E] ocurre en una clase, debe existir un indizador aplicable en la clase base.

En tiempo de enlace, las expresiones de acceso base (base-access) con el formato base.I y base[E] se evalúan exactamente igual que si se escribieran ((B)this).I y ((B)this)[E], donde B es la clase base de la clase o struct en que tiene lugar la construcción. Por tanto, base.I y base[E] corresponden a this.I y this[E], excepto en que this se ve como una instancia de la clase base.

Cuando un acceso a base (base-access) hace referencia a un miembro de función virtual (un método, una propiedad o un indizador), cambia la función que se debe invocar en tiempo de ejecución (§7.5.4). El miembro de función que se invoca se determina buscando la implementación más derivada (§10.6.3) del miembro de función con respecto a B (en lugar de la relativa al tipo de tiempo de ejecución de this, como sería normal en un acceso distinto del acceso a base). Por lo tanto, dentro de un override de un miembro de función virtual, un acceso a base (base-access) puede utilizarse para llamar a la implementación heredada del miembro de función. Si el miembro de función al que hace referencia un acceso a base (base-access) es abstracto, se producirá un error en tiempo de enlace.

### Operadores postfijos de incremento y decremento

post-increment-expression:  
primary-expression ++

post-decrement-expression:  
primary-expression --

El operando de una operación de sufijo de incremento o decremento debe ser una expresión clasificada como una variable, un acceso a propiedad o un acceso a indizador. El resultado de la operación es un valor del mismo tipo que el operando.

Si la expresión primaria tiene el tipo dynamic en tiempo de compilación, el operador se enlaza dinámicamente (§7.2.2), la expresión de sufijo de incremento o decremento (post-increment-expression o post-decrement-expression) tiene el tipo dynamic en tiempo de compilación y las siguientes reglas se aplican en tiempo de ejecución usando el tipo en tiempo de ejecución de la expresión primaria.

Si el operando de una operación de postfijo de incremento o decremento es una propiedad o un acceso a indizador, la propiedad o el indizador debe tener tanto un descriptor de acceso get como set. Si no es éste el caso, se produce un error en tiempo de enlace.

La resolución de sobrecargas de operadores unarios (§7.3.3) se aplica para seleccionar una implementación de operador concreta. Los operadores ++ y -- predefinidos existen para los tipos siguientes: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal y cualquier tipo enum. Los operadores predefinidos ++ devuelven el valor generado al sumar 1 al operando, y los operadores -- predefinidos devuelven el valor generado al restarle 1. En un contexto checked, si el resultado de esta suma o resta se encuentra fuera del intervalo del tipo del resultado y el tipo del resultado es un tipo entero o enum, se inicia una excepción System.OverflowException.

El procesamiento en tiempo de ejecución de una operación de postfijo de incremento o decremento de la forma x++ o x-- consta de los pasos siguientes:

* Si x se clasifica como una variable:
* x se evalúa para producir la variable.
* Se guarda el valor de x.
* Se invoca el operador seleccionado con el valor guardado de x como argumento.
* El valor devuelto por el operador se almacena en la ubicación dada por la evaluación de x.
* El valor guardado de x es el resultado de la operación.
* Si x se clasifica como una propiedad o un acceso a indizador:
* Se evalúan la expresión de instancia (si x no es static) y la lista de argumentos (si x es un acceso a indizador) asociadas con x, y el resultado se utiliza en las posteriores invocaciones de descriptor de acceso get y set.
* Se invoca el descriptor de acceso get de x y se guarda el valor devuelto.
* Se invoca el operador seleccionado con el valor guardado de x como argumento.
* Se invoca el descriptor de acceso set de x con el valor devuelto por el operador como argumento value.
* El valor guardado de x es el resultado de la operación.

Los operadores ++ y -- admiten la notación de prefijos (§7.7.5). Normalmente, el resultado de x++ o x-- es el valor de x antes de la operación, mientras que el resultado de ++x o --x es el valor de x después de la operación. En uno u otro caso, x tiene el mismo valor después de la operación.

Una implementación de operator ++ u operator -- puede invocarse mediante la notación de postfijo o prefijo. No es posible contar con implementaciones de operador independientes para las dos notaciones.

### El operador new

El operador new se utiliza para crear nuevas instancias de tipos.

Existen tres formas de expresiones new:

* Las expresiones de creación de objetos se utilizan para crear nuevas instancias de tipos de clase y tipos de valor.
* Expresiones de creación de matrices, que se utilizan para crear nuevas instancias de tipos de matriz.
* Expresiones de creación de delegados, que se utilizan para crear nuevas instancias de tipos delegados.

El operador new implica la creación de una instancia de un tipo, pero no necesariamente la asignación dinámica de memoria. En concreto, las instancias de tipos de valor no requieren más memoria que las variables en las que residen, por lo que no se llevan a cabo asignaciones dinámicas cuando se usa new para crear instancias de tipos de valor.

#### Expresiones de creación de objetos

Una expresión de creación de objetos (object-creation-expression) se utiliza para crear una nueva instancia de un tipo de clase (class-type) o un tipo de valor (value-type).

object-creation-expression:  
new type ( argument-listopt ) object-or-collection-initializeropt   
new type object-or-collection-initializer

object-or-collection-initializer:  
object-initializer  
collection-initializer

El tipo (type) de una expresión de creación de objetos (object-creation-expression) debe ser un tipo de clase (class-type), un tipo de valor (value-type) o un tipo de parámetro (type-parameter). El tipo (type) no puede ser un tipo de clase (class-type) abstract.

La lista de argumentos (argument-list) opcional (§7.5.1) solo está permitida si el tipo (type) es un tipo de clase (class-type) o un tipo struct (struct-type).

Una expresión de creación de objetos puede omitir la lista de argumentos de constructores y los paréntesis de cierre siempre que incluya un inicializador de objeto o de colección. Omitir la lista de argumentos de constructores y los paréntesis de cierre es equivalente a especificar una lista de argumentos vacía.

El procesamiento de una expresión de creación de objetos que incluye un inicializador de objeto o de colección consiste en procesar primero el constructor de instancias y, a continuación, el miembro de las inicializaciones de elemento especificadas por el inicializador de objeto (§7.6.10.2) o el inicializador de colección (§7.6.10.3).

Si alguno de los argumentos de la *lista de argumentos* opcional tiene el tipo dynamic en tiempo de compilación, la expresión de creación de objetos se enlaza dinámicamente (§7.2.2) y las siguientes reglas se aplican en tiempo de ejecución usando el tipo en tiempo de ejecución de los argumentos de la lista de argumentos que tengan el tipo dynamic en tiempo de compilación. Sin embargo, la creación de objetos recibe una comprobación en tiempo de compilación limitada como se describe en §7.5.4.

El procesamiento en tiempo de enlace de una expresión de creación de objetos (object-creation-expression) de la forma new T(A), donde T es un tipo de clase (class-type) o un tipo de valor (value-type) y A es una lista de argumentos (argument-list) opcional, se compone de los siguientes pasos:

* Si T es un tipo de valor (value-type) y A no está presente:
* La expresión de creación de objetos (object-creation-expression) es una invocación de constructor predeterminado. El resultado de la expresión de creación de objetos (object-creation-expression) es un valor de tipo T, en concreto, el valor predeterminado de T según se define en §4.1.1.
* En caso contrario, si T es un parámetro de tipo (type-parameter) y A no está presente:
* Si no se ha especificado ninguna restricción de tipo de valor ni de constructor (§10.1.5) para T, se produce un error en tiempo de enlace.
* El resultado de la expresión de creación de objetos (object-creation-expression) es un valor del tipo en tiempo de ejecución al que está enlazado el parámetro de tipo, en concreto el resultado de invocar el constructor predeterminado de dicho tipo. El tipo en tiempo de ejecución puede ser un tipo de referencia o un tipo de valor.
* En caso contrario, si T es un tipo de clase (class-type) o un tipo struct (struct-type):
* Si T es un tipo de clase (class-type) abstract, se produce un error en tiempo de compilación.
* Los constructores de instancia que se invoca se determina con las reglas de resolución de sobrecargas de §7.5.3. El conjunto de constructores de instancia candidatos está formado por todos los constructores de instancia accesibles declarados en T, que son aplicables con respecto a A (§7.5.3.1). Si el conjunto de constructores de instancia candidatos está vacío o no es posible identificar un solo constructor de instancia idóneo, se produce un error en tiempo de enlace.
* El resultado de la expresión de creación de objetos (object-creation-expression) es un valor de tipo T, en concreto, el valor generado al invocar el constructor de instancia determinado en el paso anterior.
* De lo contrario, la expresión de creación de objetos (object-creation-expression) no es válida y se produce un error en tiempo de enlace.

Incluso si la expresión de creación de objetos se enlaza dinámicamente, el tipo en tiempo de compilación sigue siendo T.

El procesamiento en tiempo de compilación de una expresión de creación de objetos (object-creation-expression) con la forma new T(A), donde T es un tipo de clase (class-type) o un tipo struct (struct-type) y A es una lista de argumentos (argument-list) opcional, se compone de los siguientes pasos:

* Si T es un tipo de clase (class-type):
* Se asigna una nueva instancia de la clase T. Si no hay memoria disponible suficiente para asignar la nueva instancia, se produce una excepción System.OutOfMemoryException y no se ejecutan más pasos.
* Todos los campos de la nueva instancia se inicializan con sus valores predeterminados (§5.2).
* Los constructores de instancia se invoca conforme a las reglas de invocación de un miembro de función (§7.5.4). Se pasa automáticamente al constructor de instancia una referencia a la instancia recién asignada y se posibilita el acceso a dicha instancia como this desde el constructor.
* Si T es un tipo struct (struct-type):
* Se crea una instancia de tipo T mediante la asignación de una variable local temporal. Dado que se requiere un constructor de instancia de un tipo struct (struct-type) para asignar definitivamente un valor a cada campo de la instancia que se crea, no es necesaria una inicialización de la variable temporal.
* Los constructores de instancia se invoca conforme a las reglas de invocación de un miembro de función (§7.5.4). Se pasa automáticamente al constructor de instancia una referencia a la instancia recién asignada y se posibilita el acceso a dicha instancia como this desde el constructor.

#### Inicializadores de objeto

Un inicializador de objeto especifica valores para cero o más campos o propiedades de un objeto.

object-initializer:  
{ member-initializer-listopt }  
{ member-initializer-list , }

member-initializer-list:  
member-initializer  
member-initializer-list , member-initializer

member-initializer:  
identifier = initializer-value

initializer-value:  
expression  
object-or-collection-initializer

Un inicializador de objeto se compone de una secuencia de inicializadores de miembro, que figuran entre los tokens { y } separados por comas. Cada inicializador de miembro debe nombrar una propiedad o campo accesible del objeto que se inicializa, seguido por un signo igual y una expresión o un inicializador de objeto o colección. Es un error que un inicializador de objeto incluya más de un inicializador de miembro para la misma propiedad o campo. No es posible que un inicializador de objeto haga referencia a un objeto recién creado que se está inicializando.

Un inicializador de miembro que especifica una expresión después de un signo igual se procesa igual que una asignación (§7.17.1) al campo o propiedad.

Un inicializador de miembro que especifica un inicializador de objeto después de un signo igual es un inicializador de objeto anidado, es decir una inicialización de un objeto incrustado. En lugar de asignar un nuevo valor al campo o propiedad, las asignaciones de un inicializador de objeto anidado se tratan como asignaciones a miembros del campo o propiedad. Los inicializadores de objetos anidados no se pueden aplicar a propiedades con un tipo de valor, o a campos de sólo lectura con un tipo de valor.

Un inicializador de miembro que especifica un inicializador de colección después de un signo igual es una inicialización de una colección incrustada. En lugar de asignar una nueva colección al campo o propiedad, los elementos dados en un inicializador se agregan a la colección referenciada por el campo o propiedad. El campo o la propiedad deben ser un tipo de colección que cumpla los requisitos especificados en la sección §7.6.10.3.

La siguiente clase representa un punto con dos coordenadas:

public class Point  
{  
 int x, y;

public int X { get { return x; } set { x = value; } }  
 public int Y { get { return y; } set { y = value; } }  
}

Una instancia de Point se puede crear e inicializar de la siguiente forma:

Point a = new Point { X = 0, Y = 1 };

que tiene el mismo efecto que

Point \_\_a = new Point();  
\_\_a.X = 0;  
\_\_a.Y = 1;   
Point a = \_\_a;

donde \_\_a es una variable temporal inaccesible e invisible en caso contrario. La siguiente clase representa un rectángulo creado a partir de dos puntos:

public class Rectangle  
{  
 Point p1, p2;

public Point P1 { get { return p1; } set { p1 = value; } }  
 public Point P2 { get { return p2; } set { p2 = value; } }  
}

Una instancia de Rectangle se puede crear e inicializar de la siguiente forma:

Rectangle r = new Rectangle {  
 P1 = new Point { X = 0, Y = 1 },  
 P2 = new Point { X = 2, Y = 3 }  
};

que tiene el mismo efecto que

Rectangle \_\_r = new Rectangle();  
Point \_\_p1 = new Point();  
\_\_p1.X = 0;  
\_\_p1.Y = 1;  
\_\_r.P1 = \_\_p1;  
Point \_\_p2 = new Point();  
\_\_p2.X = 2;  
\_\_p2.Y = 3;  
\_\_r.P2 = \_\_p2;   
Rectangle r = \_\_r;

donde \_\_r, \_\_p1 y \_\_p2 son variables temporales que, en caso contrario, son inaccesibles e invisibles.

Si el constructor de Rectangle’s asigna las dos instancias de Point incrustadas

public class Rectangle  
{  
 Point p1 = new Point();  
 Point p2 = new Point();

public Point P1 { get { return p1; } }  
 public Point P2 { get { return p2; } }  
}

el siguiente constructor se puede utilizar para inicializar las instancias Point incrustadas en lugar de asignar nuevas instancias:

Rectangle r = new Rectangle {  
 P1 = { X = 0, Y = 1 },  
 P2 = { X = 2, Y = 3 }  
};

que tiene el mismo efecto que

Rectangle \_\_r = new Rectangle();  
\_\_r.P1.X = 0;  
\_\_r.P1.Y = 1;  
\_\_r.P2.X = 2;  
\_\_r.P2.Y = 3;  
Rectangle r = \_\_r;

#### Inicializadores de colección

Un inicializador de colección especifica los elementos de una colección.

collection-initializer:  
{ element-initializer-list }  
{ element-initializer-list , }

element-initializer-list:  
element-initializer  
element-initializer-list , element-initializer

element-initializer:  
non-assignment-expression  
{ expression-list }

expression-list:  
expression  
expression-list , expression

Un inicializador de colección se compone de una secuencia de inicializadores de elemento, que figuran entre los tokens { y } y separados por comas. Cada inicializador de elemento especifica un elemento que se va a agregar al objeto de colección que se inicializa, y se compone de una lista de expresiones entre los tokens { y } y separadas por comas. Un inicializador de elemento de una sola expresión se puede escribir sin corchetes, pero en ese caso no puede ser una expresión de asignación, con el fin de evitar ambigüedad con los inicializadores de miembro. La producción de expresiones de no asignación (non-assignment-expression) se define en la sección §7.18.

A continuación, se muestra un ejemplo de una expresión de creación de objeto que incluye un inicializador de colección:

List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

El objeto de colección al que se aplica un inicializador de colección debe ser un tipo que implemente System.Collections.IEnumerable o se producirá un error durante la compilación. Para cada elemento especificado en orden, el inicializador de colección invoca un método Add en el objeto de destino con la lista de expresiones del inicializador de elemento como lista de argumentos, aplicando una resolución de sobrecarga normal para cada invocación. Por tanto, el objeto de colección debe contener un método Add aplicable para cada inicializador de elemento.

La siguiente clase representa un contacto con un nombre y una lista de números de teléfono:

public class Contact  
{  
 string name;  
 List<string> phoneNumbers = new List<string>();

public string Name { get { return name; } set { name = value; } }

public List<string> PhoneNumbers { get { return phoneNumbers; } }  
}

List<Contact> se puede crear e inicializar de la siguiente forma:

var contacts = new List<Contact> {  
 new Contact {  
 Name = "Chris Smith",  
 PhoneNumbers = { "206-555-0101", "425-882-8080" }  
 },  
 new Contact {  
 Name = "Bob Harris",  
 PhoneNumbers = { "650-555-0199" }  
 }  
};

que tiene el mismo efecto que

var \_\_clist = new List<Contact>();  
Contact \_\_c1 = new Contact();  
\_\_c1.Name = "Chris Smith";  
\_\_c1.PhoneNumbers.Add("206-555-0101");  
\_\_c1.PhoneNumbers.Add("425-882-8080");  
\_\_clist.Add(\_\_c1);  
Contact \_\_c2 = new Contact();  
\_\_c2.Name = "Bob Harris";  
\_\_c2.PhoneNumbers.Add("650-555-0199");  
\_\_clist.Add(\_\_c2);  
var contacts = \_\_clist;

donde \_\_clist, \_\_c1 y \_\_c2 son variables temporales que, en caso contrario, son inaccesibles e invisibles.

#### Expresiones de creación de matrices

Para crear una nueva instancia de un tipo matricial (array-type), se utiliza una expresión de creación de matrices (array-creation-expression).

array-creation-expression:  
new non-array-type [ expression-list ] rank-specifiersopt array-initializeropt  
new array-type array-initializer   
new rank-specifier array-initializer

Una expresión de creación de matrices de la primera forma asigna una instancia de matriz del tipo que se obtiene al eliminar cada una de las expresiones individuales de la lista de expresiones. Por ejemplo, la expresión de creación de matrices new int[10, 20] produce una instancia de matriz de tipo int[,], y la expresión new int[10][,] produce una matriz de tipo int[][,]. Cada expresión de la lista de expresiones debe ser de tipo int, uint, long o ulong, o bien de un tipo que pueda convertirse implícitamente en uno o más de estos tipos. El valor de cada expresión determina la longitud de la dimensión correspondiente en la instancia de matriz recién asignada. Dado que la longitud de la dimensión de matriz no puede ser negativa, si existe una expresión constante (constant-expression) con un valor negativo en la lista de expresiones, se producirá un error en tiempo de compilación.

Excepto en un contexto no seguro (§18.1), la forma de las matrices no se especifica.

Si una expresión de creación de matriz de la primera forma incluye un inicializador de matriz, cada expresión de la lista de expresiones debe ser una constante, y las longitudes de rango y dimensión especificadas por la lista de expresiones deben coincidir con las del inicializador.

En una expresión de creación de matrices de la segunda o tercera forma, el rango del tipo de matriz especificado o el especificador de rango debe coincidir con el del inicializador. Las longitudes de las dimensiones individuales se infieren del número de elementos en cada uno de los niveles de anidamiento correspondientes del inicializador de matriz. Por lo tanto, la expresión

new int[,] {{0, 1}, {2, 3}, {4, 5}}

se corresponde exactamente con

new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}

Una expresión de creación de matrices de la tercera forma se conoce como expresión de creación de matrices con asignación de tipo implícita. Es similar a la segunda forma, excepto en que el tipo de elemento de la matriz no se da explícitamente, sino que se determina como el mejor tipo común (§7.5.2.14) del conjunto de expresiones del inicializador de matriz. Para una matriz multidimensional, es decir, una donde el especificador de rango (rank-specifier) contiene al menos una coma, este conjunto comprende todas las expresiones (expression) encontradas en los inicializadores de matrices (array-initializer) anidados.

Los inicializadores de matrices se describen más detalladamente en §12.6.

El resultado de evaluar una expresión de creación de matrices se clasifica como un valor, concretamente como una referencia a la instancia de matriz recién asignada. El procesamiento en tiempo de ejecución de una expresión de creación de matrices consta de los siguientes pasos:

* Las expresiones de longitud de dimensión de la lista de expresiones (expression-list) se evalúan por orden, de izquierda a derecha. Después de la evaluación de cada expresión, se realiza una conversión implícita (§6.1) a uno de los tipos siguientes: int, uint, long, ulong. Se elige el primer tipo de esta lista para el cual existe una conversión implícita. Si la evaluación de una expresión o de la conversión implícita posterior causa una excepción, no se evalúan otras expresiones y no se ejecutan nuevos pasos.
* Los valores calculados para las longitudes de dimensión se validan como sigue. Si uno o más valores son menores que cero, se inicia una excepción System.OverflowException y no se ejecutan nuevos pasos.
* Se asigna una instancia de matriz con las longitudes de dimensión dadas. Si no hay memoria disponible suficiente para asignar la nueva instancia, se produce una excepción System.OutOfMemoryException y no se ejecutan más pasos.
* Todos los elementos de la nueva instancia de matriz se inicializan con sus valores predeterminados (§5.2).
* Si la expresión de creación de matriz contiene un inicializador de matriz, entonces se evalúan las expresiones del inicializador y se asignan a sus elemento de matriz correspondientes. Las evaluaciones y asignaciones se llevan a cabo en el orden en que están escritas las expresiones en el inicializador de matriz, es decir, los elementos se inicializan en orden de índice creciente, empezando por la dimensión situada más a la derecha. Si la evaluación de una expresión dada o la posterior asignación al elemento de matriz correspondiente causa una excepción, no se inicializan más elementos (y los elementos restantes conservan sus valores predeterminados).

Una expresión de creación de matriz permite crear una instancia de una matriz con elementos de un tipo de matriz, pero los elementos de una matriz como ésta deben inicializarse de forma manual. Por ejemplo, la instrucción

int[][] a = new int[100][];

crea una matriz de una dimensión con 100 elementos de tipo int[]. El valor inicial de cada elemento es null. Una misma expresión de creación de matrices no puede crear también instancias de las submatrices, y la instrucción

int[][] a = new int[100][5]; // Error

da como resultado un error en tiempo de compilación. La creación de instancias de las submatrices debe realizarse de forma manual, como en

int[][] a = new int[100][];  
for (int i = 0; i < 100; i++) a[i] = new int[5];

Cuando una matriz de matrices tiene forma “rectangular”, es decir, cuando todas las submatrices tienen la misma longitud, es más eficaz utilizar una matriz de varias dimensiones. En el ejemplo anterior, la creación de instancias de la matriz de matrices crea 101 objetos, una matriz externa y 100 submatrices. Por el contrario,

int[,] = new int[100, 5];

sólo crea un objeto, una matriz de dos dimensiones, y realiza la asignación mediante una sola instrucción.

A continuación se muestran ejemplos de expresiones de creación de matrices con asignación de tipo implícita.

var a = new[] { 1, 10, 100, 1000 }; // int[]

var b = new[] { 1, 1.5, 2, 2.5 }; // double[]

var c = new[,] { { "hello", null }, { "world", "!" } }; // string[,]

var d = new[] { 1, "one", 2, "two" }; // Error

La última expresión origina un error durante la compilación porque ni int ni string son implícitamente convertibles entre sí, de modo que no hay un mejor tipo común. En este caso se debe utilizar una expresión de creación de matrices con asignación de tipo implícita, por ejemplo, especificando el tipo para que sea object[]. Alternativamente, uno de los elementos se puede convertir a un tipo base común, que entonces pasaría a ser el tipo de elemento inferido.

Las expresiones de creación de matrices con asignación de tipo implícita se pueden combinar con inicializadores de objetos anónimos (§7.6.10.6) para crear estructuras de datos con asignación de tipo anónima. Por ejemplo:

var contacts = new[] {  
 new {  
 Name = "Chris Smith",  
 PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }  
 },  
 new {  
 Name = "Bob Harris",  
 PhoneNumbers = new[] { "650-555-0199" }  
 }  
};

#### Expresiones de creación de delegados

Una expresión de creación de delegados (delegate-creation-expression) se usa para crear una nueva instancia de un tipo delegado (delegate-type).

delegate-creation-expression:  
new delegate-type ( expression )

El argumento de una expresión de creación de delegados debe ser un grupo de métodos, una función anónima o un valor del tipo dynamic en tiempo de compilación o un tipo delegado (delegate-type). Si el argumento es un conjunto de métodos, identifica el método y, en el caso de un método de instancia, el objeto para el que se crea un delegado. Si el argumento es una función anónima define los parámetros y el cuerpo del método para el destino del delegado. Si el argumento es un valor, identifica una instancia de delegado del que se va crear una copia.

Si la expresión tiene un tipo dynamic en tiempo de compilación, la expresión de creación de delegados (delegate-creation-expression) se enlaza dinámicamente (§7.2.2) y las reglas siguientes se aplican en tiempo de ejecución, usando el tipo en tiempo de ejecución de la expresión. De lo contrario las reglas se aplican en tiempo de compilación.

El procesamiento en tiempo de enlace de una expresión de creación de delegados (delegate-creation-expression) con el formato new D(E), donde D es un tipo delegado (delegate-type) y E es una expresión (expression), se compone de los siguientes pasos:

* Si E es un grupo de métodos, la expresión de creación de delegados se procesa de la misma manera que una conversión de grupo de métodos (§6.6) de E a D.
* Si E es una función anónima, la expresión de creación de delegados se procesa de la misma manera que una conversión de función anónima (§6.5) de E a D.
* Si E es un valor, debe ser compatible (§15.1) con D y el resultado es una referencia a un delegado recién creado del tipo D que hace referencia a la misma lista de invocación que E. Si E no es compatible con D, se genera un error de compilación.

El procesamiento en tiempo de ejecución de una expresión de creación de delegados (delegate-creation-expression) con el formato new D(E), donde D es un tipo delegado (delegate-type) y E es una expresión (expression), se compone de los siguientes pasos:

* Si E es un grupo de métodos, la expresión de creación de delegados se evalúa como una conversión de grupo de métodos (§6.6) de E a D.
* Si E es una función anónima, la expresión de creación de delegados se evalúa como una conversión de función anónima de E a D (§6.5).
* Si E es un valor de un tipo delegado (delegate-type):
* Se evalúa E. Si esta evaluación da lugar a una excepción, no se ejecutan nuevos pasos.
* Si el valor de E es null, se inicia una excepción System.NullReferenceException y no se ejecutan más pasos.
* Se asigna una nueva instancia del tipo delegado D. Si no hay memoria disponible suficiente para asignar la nueva instancia, se produce una excepción System.OutOfMemoryException y no se ejecutan más pasos.
* Se inicializa la nueva instancia de delegado con la misma lista de invocaciones que la instancia de delegado proporcionada por E.

La lista de invocaciones de un delegado se determina cuando se inicializa el delegado y permanece constante durante toda la duración del delegado. Dicho de otra forma: no se podrán cambiar las entidades de destino invocables de un delegado una vez haya sido creado el delegado. Si se combinan dos delegados o se quita uno de otro (§15.1), se produce un nuevo delegado; no se cambia el contenido de ningún delegado existente.

No es posible crear un delegado que haga referencia a una propiedad, un indizador, un operador definido por el usuario, un constructor de instancia, un destructor o un constructor estático.

Como se ha explicado antes, cuando se crea un delegado a partir de un grupo de métodos, la lista de parámetros formales y el tipo de valor devuelto del delegado determinan cuál de los métodos sobrecargados se debe seleccionar. En el ejemplo

delegate double DoubleFunc(double x);

class A  
{  
 DoubleFunc f = new DoubleFunc(Square);

static float Square(float x) {  
 return x \* x;  
 }

static double Square(double x) {  
 return x \* x;  
 }  
}

el campo A.f se inicializa con un delegado que hace referencia al segundo método Square debido a que ese método coincide exactamente con la lista de parámetros formales y el tipo de valor devuelto de DoubleFunc. Si no hubiera existido el segundo método Square, se habría producido un error de compilación.

#### Expresiones de creación de objetos anónimos

Una expresión de creación de objetos anónimos (anonymous-object-creation-expression) se utiliza para crear un objeto de un tipo anónimo.

anonymous-object-creation-expression:  
new anonymous-object-initializer

anonymous-object-initializer:  
{ member-declarator-listopt }  
{ member-declarator-list , }

member-declarator-list:  
member-declarator  
member-declarator-list , member-declarator

member-declarator:  
simple-name  
member-access  
base-access  
identifier = expression

Un inicializador de objeto anónimo declara un tipo anónimo y devuelve una instancia de ese tipo. Un tipo anónimo es un tipo de clase sin nombre que hereda directamente de object. Los miembros de un tipo anónimo son una secuencia de propiedades de sólo lectura inferidas del inicializador de objetos anónimos utilizado para crear una instancia del tipo. Específicamente, un inicializador de objeto anónimo con la estructura

new { p1 = e1 , p2 = e2 , … pn = en }

declara un tipo anónimo con la estructura

class \_\_Anonymous1  
{  
 private readonly T1 f1 ;  
 private readonly T2 f2 ;  
 …  
 private readonly Tn fn ;

public \_\_Anonymous1(T1 a1, T2 a2,…, Tn an) {  
 f1 = a1 ;  
 f2 = a2 ;  
 …  
 fn = an ;  
 }

public T1 p1 { get { return f1 ; } }  
 public T2 p2 { get { return f2 ; } }  
 …  
 public Tn pn { get { return fn ; } }

public override bool Equals(object \_\_o) { … }  
 public override int GetHashCode() { … }  
}

donde cada Tx es el tipo de expresión ex correspondiente. La expresión utilizada en un declarador de miembro (member-declarator) debe tener un tipo. Por tanto, es un error en tiempo de compilación que una expresión de un declarador de miembro (member-declarator) sea nula o una función anónima. También se trata de un error en tiempo de compilación que la expresión tenga un tipo no seguro.

El compilador genera automáticamente los nombres de un tipo anónimo y del parámetro para su método Equals y no se puede hacer referencia a ellos en el texto del programa.

Dentro del mismo programa, dos inicializadores de objetos anónimos que especifiquen una secuencia de propiedades con los mismos nombres y tipos en tiempo de compilación en el mismo orden producirán instancias del mismo tipo anónimo.

En el ejemplo

var p1 = new { Name = "Lawnmower", Price = 495.00 };  
var p2 = new { Name = "Shovel", Price = 26.95 };  
p1 = p2;

la asignación en la última línea se permite porque p1 y p2 son del mismo tipo anónimo.

Los métodos Equals y GetHashcode en tipos anónimos invalidan los métodos heredados de object, y se definen en términos de Equals y GetHashcode de las propiedades, para que las dos instancias del mismo tipo anónimo sean iguales si y sólo si todas sus propiedades son iguales.

Un declarador de miembro se puede abreviar como un nombre simple (§7.5.2), un acceso a miembro (§7.5.4) o un acceso a base (§7.6.8). Es lo que se denomina inicializador de proyección y es la forma abreviada de una declaración de asignación para una propiedad con el mismo nombre. Concretamente, los declaradores de miembros con la estructura

identifier expr . identifier

son precisamente equivalentes a los siguientes, respectivamente:

identifer = identifier identifier = expr . identifier

Por tanto, en un inicializador de proyección el identificador (identifier) selecciona tanto el valor como el campo o propiedad para el que el valor se asigna. Intuitivamente, un inicializador de proyección proyecta no sólo un valor, sino también el nombre de ese valor.

### Operador typeof

El operador typeof se utiliza para obtener el objeto System.Type \b \b para un tipo.

typeof-expression:  
typeof ( type )  
typeof ( unbound-type-name )  
typeof ( void )

unbound-type-name:  
identifier generic-dimension-specifieropt  
identifier :: identifier generic-dimension-specifieropt  
unbound-type-name . identifier generic-dimension-specifieropt

generic-dimension-specifier:  
< commasopt >

commas:  
,  
commas ,

La primera forma de expresión typeof (typeof-expression) consta de una palabra clave typeof seguida de un tipo (type) entre paréntesis. El resultado de una expresión de esta forma es el objeto System.Type del tipo indicado. Solo existe un objeto System.Type para cualquier tipo dado. Esto significa que, para un tipo T, typeof(T) == typeof(T) siempre es true. El tipo no puede ser dynamic.

La segunda forma de la expresión typeof (typeof-expression) consta de una palabra clave typeof seguida de un nombre de tipo sin enlazar (unbound-type-name) entre paréntesis. Un nombre de tipo sin enlazar (unbound-type-name) es muy similar a un nombre de tipo (type-name) (§3.8), salvo que un nombre de tipo sin enlazar contiene especificadores de dimensión genéricos (generic-dimension-specifier) donde un nombre de tipo contiene listas de argumentos de tipo (type-argument-list). Cuando el operando de una expresión typeof (typeof-expression) es una secuencia de tokens que satisface las gramáticas de nombre de tipo sin enlazar (unbound-type-name) y nombre de tipo (type-name), es decir, cuando no contiene un especificador de dimensión genérico (generic-dimension-specifier) ni una lista de argumentos de tipo (type-argument-list), se considera que la secuencia de tokens es un nombre de tipo (type-name). El significado de un nombre de tipo sin enlazar (unbound-type-name) se determina como sigue:

* Convertir la secuencia de tokens en un nombre de tipo (type-name) al reemplazar cada especificador de dimensión genérico (generic-dimension-specifier) por una lista de argumentos de tipo (type-argument-list) con el mismo número de comas y la palabra clave object que cada argumento de tipo (type-argument).
* Evaluar el nombre de tipo (type-name) resultante, al mismo tiempo que se omiten todas las restricciones de los parámetros de tipo.
* El nombre de tipo sin enlazar (unbound-type-name) se resuelve como un tipo genérico sin enlazar asociado al tipo construido resultante (§4.4.3).

El resultado de la expresión typeof (typeof-expression) es el objeto System.Type para el tipo genérico sin enlazar resultante.

La tercera forma de la expresión typeof (typeof-expression) consta de una palabra clave typeof seguida de una palabra clave void entre paréntesis. El resultado de una expresión con esta estructura es el objeto System.Type que representa la ausencia de un tipo. El objeto del tipo devuelto por typeof(void) es exclusivo del objeto del tipo devuelto por cualquier tipo. Este objeto de tipo especial es útil en las bibliotecas de clases que permiten la reflexión en métodos del lenguaje, donde dichos métodos desean tener un medio para representar el tipo de valor devuelto de cualquier método, incluidos los métodos void, con una instancia de System.Type.

El operador typeof se puede utilizar en un parámetro de tipo. El resultado es el objeto System.Type para el tipo en tiempo de ejecución enlazado al parámetro de tipo. El operador typeof también se puede utilizar en un tipo construido o en un tipo genérico no sin enlazar (§4.4.3). El objeto System.Type para el tipo genérico sin enlazar no es el mismo que el objeto System.Type del tipo de instancia. El tipo de instancia siempre es un tipo construido cerrado en tiempo de ejecución, porque el objeto System.Type depende de los argumentos de tipo en tiempo de ejecución que se están utilizando, mientras que el tipo genérico sin enlazar no tiene argumentos de tipo.

El ejemplo

using System;

class X<T>  
{  
 public static void PrintTypes() {  
 Type[] t = {  
 typeof(int),  
 typeof(System.Int32),  
 typeof(string),  
 typeof(double[]),  
 typeof(void),  
 typeof(T),  
 typeof(X<T>),  
 typeof(X<X<T>>),  
 typeof(X<>)  
 };  
 for (int i = 0; i < t.Length; i++) {  
 Console.WriteLine(t[i]);  
 }  
 }  
}

class Test  
{  
 static void Main() {  
 X<int>.PrintTypes();  
 }  
}

produce el resultado:

System.Int32  
System.Int32  
System.String  
System.Double[]  
System.Void  
System.Int32  
X`1[System.Int32]  
X`1[X`1[System.Int32]]  
X`1[T]

Debe tenerse en cuenta que int e System.Int32 son el mismo tipo.

Observe también que el resultado de typeof(X<>) no depende del argumento de tipo, pero el resultado de typeof(X<T>) sí.

### Los operadores checked y unchecked

Los operadores checked y unchecked se utilizan con el fin de controlar el contexto de comprobación de desbordamiento para operaciones aritméticas y conversiones.

checked-expression:  
checked ( expression )

unchecked-expression:  
unchecked ( expression )

El operador checked evalúa la expresión contenida en un contexto comprobado (checked), y el operador unchecked evalúa la expresión contenida en un contexto no comprobado (unchecked). Una expresión checked (checked-expression) o una expresión unchecked (unchecked-expression) se corresponde exactamente con una expresión entre paréntesis (parenthesized-expression) (§7.6.3), excepto porque la expresión contenida se evalúa en el contexto de comprobación de desbordamiento dado.

El contexto de comprobación de desbordamiento también puede controlarse mediante instrucciones checked y unchecked (§8.11).

Las siguientes operaciones resultan afectadas por el contexto de comprobación de desbordamiento establecido por los operadores e instrucciones checked y unchecked:

* Los operadores unarios predefinidos ++ y -- (§7.6.9 y §7.7.5), cuando el operando es de un tipo entero.
* El operador unario predefinido - (§7.7.2), cuando el operando es de un tipo entero.
* Los operadores binarios predefinidos +, -, \* y / (§7.8), cuando los dos operandos son de tipo entero.
* Las conversiones explícitas numéricas (§6.2.1) de un tipo entero a otro, o de float o double a un tipo entero.

Cuando una de las operaciones anteriores produce un resultado demasiado grande para representarlo en el tipo de destino, el contexto en que se ejecuta la operación controla el comportamiento resultante:

* En un contexto checked, si la operación es una expresión constante (§7.19), se produce un error en tiempo de compilación. O bien, si la operación se realiza en tiempo de ejecución, se produce una excepción System.OverflowException.
* En un contexto unchecked, el resultado se trunca, descartándose cualquier bit de orden superior que no quepa en el tipo de destino.

Para expresiones no constantes (expresiones que se evalúan en tiempo de ejecución) no incluidas por operadores o instrucciones checked o unchecked, el contexto de comprobación de desbordamiento predeterminado es unchecked salvo que factores externos (como la configuración de opciones del compilador y del entorno de ejecución) llamen a la evaluación checked.

Para expresiones constantes (expresiones que pueden evaluarse completamente en tiempo de compilación), el contexto de comprobación de desbordamiento predeterminado siempre es checked. Salvo que una expresión constante se coloque explícitamente en un contexto unchecked, los desbordamientos que ocurren durante la evaluación en tiempo de compilación de la expresión siempre causan errores de tiempo de compilación.

El cuerpo de una función anónima no se ve afectado por los contextos checked o unchecked en los que ocurre la función anónima.

En el ejemplo

class Test  
{  
 static readonly int x = 1000000;  
 static readonly int y = 1000000;

static int F() {  
 return checked(x \* y); // Throws OverflowException  
 }

static int G() {  
 return unchecked(x \* y); // Returns -727379968  
 }

static int H() {  
 return x \* y; // Depends on default  
 }  
}

no se notifican errores de tiempo de compilación puesto que ninguna de las expresiones puede evaluarse en tiempo de compilación. En tiempo de ejecución, el método F inicia una excepción System.OverflowException y el método G devuelve –727379968 (los 32 bits menores del resultado fuera de intervalo). El comportamiento del método H depende del contexto de comprobación de desbordamiento predeterminado de la compilación, pero es el mismo que F o el mismo que G.

En el ejemplo

class Test  
{  
 const int x = 1000000;  
 const int y = 1000000;

static int F() {  
 return checked(x \* y); // Compile error, overflow  
 }

static int G() {  
 return unchecked(x \* y); // Returns -727379968  
 }

static int H() {  
 return x \* y; // Compile error, overflow  
 }  
}

los desbordamientos que se producen durante la evaluación de las expresiones constantes en F y H causan la notificación de errores en tiempo de compilación, puesto que las expresiones se evalúan en un contexto checked. También se produce un desbordamiento cuando se evalúa la expresión constante en G, si bien, al tener lugar la evaluación en un contexto unchecked, dicho desbordamiento no se notifica.

Los operadores checked y unchecked solamente afectan al contexto de comprobación de desbordamiento para aquellas operaciones que están contenidas textualmente en los tokens “(” y “)”. Los operadores no tienen ningún efecto sobre los miembros de función que se invocan como consecuencia de la evaluación de la expresión contenida. En el ejemplo

class Test  
{  
 static int Multiply(int x, int y) {  
 return x \* y;  
 }

static int F() {  
 return checked(Multiply(1000000, 1000000));  
 }  
}

el uso de checked en F no afecta a la evaluación de x \* y en Multiply y, por lo tanto, x \* y se evalúa en el contexto de comprobación de desbordamiento predeterminado.

El operador unchecked es práctico cuando se escriben constantes de los tipos enteros con signo en notación hexadecimal. Por ejemplo:

class Test  
{  
 public const int AllBits = unchecked((int)0xFFFFFFFF);

public const int HighBit = unchecked((int)0x80000000);  
}

Las dos constantes hexadecimales anteriores son de tipo uint. Dado que las constantes están fuera del intervalo de int, sin el operador unchecked, las conversiones al tipo int producirían errores en tiempo de compilación.

Las instrucciones y operadores checked y unchecked permiten a los programadores controlar algunos aspectos de determinados cálculos numéricos. No obstante, el comportamiento de algunos operadores numéricos depende de los tipos de datos de sus operandos. Por ejemplo, la multiplicación de dos decimales siempre genera una excepción por desbordamiento, incluso en una construcción unchecked explícita. De igual manera, la multiplicación de dos tipos de punto flotante nunca genera una excepción por desbordamiento, incluso en una construcción checked explícita. Asimismo, otros operadores nunca resultan afectados por el modo de comprobación, ya sea predeterminado o explícito.

### Expresiones de valor predeterminadas

Una expresión de valor predeterminada se utiliza para obtener el valor predeterminado (§5.2) de un tipo. Generalmente, una expresión de valor predeterminada se utiliza para parámetros de tipo puesto que es posible que no se detecte si el parámetro de tipo es un tipo de valor o un tipo de referencia. (No existe una conversión del literal null al parámetro de tipo a menos que éste sea un tipo de referencia.)

default-value-expression:  
default ( type )

Si el tipo (type) de una expresión de valor predeterminada (default-value-expression) se evalúa como un tipo de referencia en tiempo de ejecución, null se convierte en dicho tipo como resultado. Si el tipo (type) de una expresión de valor predeterminada (default-value-expression) se evalúa como un tipo de valor en tiempo de ejecución, se obtiene el valor predeterminado del tipo de valor (value-type) como resultado (§4.1.2).

Una expresión de valor predeterminada (default-value-expression) es una expresión constante (§7.19) si el tipo es un tipo de referencia o un parámetro de tipo que es un tipo de referencia (§10.1.5). Además, una expresión de valor predeterminada (default-value-expression) es una expresión constante si el tipo es uno de los siguientes tipos de valor: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool o cualquier tipo de enumeración.

### Expresiones de métodos anónimos

Una expresión de método anónimo (anonymous-method-expression) es uno de los dos métodos que existen para definir una función anónima. Este tema se describe con más detalle en la sección §7.15.

## Operadores unarios

Los operadores +, -, !, ~, ++, --, await y de conversión se denominan operadores unarios.

unary-expression:  
primary-expression  
+ unary-expression  
- unary-expression  
! unary-expression  
~ unary-expression  
pre-increment-expression  
pre-decrement-expression  
cast-expression  
await-expression

Si el operando de una expresión unaria (unary-expression) tiene el tipo dynamic en tiempo de compilación, se enlaza dinámicamente (§7.2.2). En este caso, el tipo en tiempo de compilación de la expresión unaria (unary-expression) es dynamic y la resolución descrita a continuación se produce en tiempo de compilación usando el tipo en tiempo de compilación del operando.

### Operador unario de signo más

Para una operación con la forma +x, se aplica la resolución de sobrecargas de operadores unarios (§7.3.3) con el fin de seleccionar una implementación de operador concreta. El operando se convierte al tipo de parámetro del operador seleccionado, y el tipo del resultado es el tipo de valor devuelto del operador. Los operadores de signo más unarios predefinidos son:

int operator +(int x);  
uint operator +(uint x);  
long operator +(long x);  
ulong operator +(ulong x);  
float operator +(float x);  
double operator +(double x);  
decimal operator +(decimal x);

Para cada uno de estos operadores, el resultado es sencillamente el valor del operando.

### Operador unario de signo menos

Para una operación con la forma –x, se aplica la resolución de sobrecargas de operadores unarios (§7.3.3) con el fin de seleccionar una implementación de operador concreta. El operando se convierte al tipo de parámetro del operador seleccionado, y el tipo del resultado es el tipo de valor devuelto del operador. Los operadores de negación predefinidos son:

* Negación entera:

int operator –(int x);  
long operator –(long x);

El resultado se calcula restando x de cero. Si el valor de x es el valor menor representable del tipo de operando (−231 para int o −263 para long), la negación matemática de x no se puede representar en este tipo de operando. Si esto ocurre dentro de un contexto checked, se inicia una excepción System.OverflowException; si ocurre dentro de un contexto unchecked, el resultado es el valor del operando y no se informa del desbordamiento.

Si el operando del operador de negación es de tipo uint, se convierte al tipo long y el tipo del resultado es long. Una excepción es la regla que permite escribir el valor int −2147483648 (−231) como un literal entero decimal (§2.4.4.2).

Si el operando del operador de negación es de tipo ulong, se produce un error durante la compilación. Una excepción es la regla que permite escribir el valor long −9223372036854775808 (−263) como un literal entero decimal (§2.4.4.2).

* Negación de punto flotante:

float operator –(float x);  
double operator –(double x);

El resultado es el valor de x con su signo invertido. Si x es NaN, el resultado es también NaN.

* Negación decimal:

decimal operator –(decimal x);

El resultado se calcula restando x de cero. La negación decimal equivale a usar el operador unario de signo menos del tipo System.Decimal.

### Operador de negación lógica

Para una operación con la forma !x, se aplica la resolución de sobrecargas de operadores unarios (§7.3.3) con el fin de seleccionar una implementación de operador concreta. El operando se convierte al tipo de parámetro del operador seleccionado, y el tipo del resultado es el tipo de valor devuelto del operador. Solamente existe un operador de negación lógica predefinido:

bool operator !(bool x);

Este operador calcula la negación lógica del operando: si este es true, el resultado es false. Si el operando es false, el resultado es true.

### Operador de complemento de bit a bit

Para una operación con la forma ~x, se aplica la resolución de sobrecargas de operadores unarios (§7.3.3) con el fin de seleccionar una implementación de operador concreta. El operando se convierte al tipo de parámetro del operador seleccionado, y el tipo del resultado es el tipo de valor devuelto del operador. Los operadores de complemento de bit a bit predefinidos son:

int operator ~(int x);  
uint operator ~(uint x);  
long operator ~(long x);  
ulong operator ~(ulong x);

Para cada uno de estos operadores, el resultado de la operación es el complemento de bit a bit de x.

Todos los tipos de enumeración E proporcionan implícitamente el siguiente operador de complemento de bit a bit:

E operator ~(E x);

El resultado de evaluar ~x, donde x es una expresión de un tipo de enumeración E con un tipo subyacente U, es exactamente el mismo que el de evaluar (E)(~(U)x), excepto que la conversión a E siempre se realiza como dentro de un contexto unchecked (§7.6.12).

### Operadores prefijos de incremento y decremento

pre-increment-expression:  
++ unary-expression

pre-decrement-expression:  
-- unary-expression

El operando de una operación de prefijo de incremento o decremento debe ser una expresión clasificada como un variable, un acceso a propiedad o un acceso a indizador. El resultado de la operación es un valor del mismo tipo que el operando.

Si el operando de una operación de prefijo de incremento o decremento es una propiedad o un acceso a indizador, la propiedad o el indizador debe tener tanto un descriptor de acceso get como set. Si no es éste el caso, se produce un error en tiempo de enlace.

La resolución de sobrecargas de operadores unarios (§7.3.3) se aplica para seleccionar una implementación de operador concreta. Los operadores ++ y -- predefinidos existen para los tipos siguientes: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal y cualquier tipo enum. Los operadores predefinidos ++ devuelven el valor generado al sumar 1 al operando, y los operadores -- predefinidos devuelven el valor generado al restarle 1. En un contexto checked, si el resultado de esta suma o resta se encuentra fuera del intervalo del tipo del resultado y el tipo del resultado es un tipo entero o enum, se inicia una excepción System.OverflowException.

El procesamiento en tiempo de ejecución de una operación de prefijo de incremento o decremento de la forma ++x o --x consta de los pasos siguientes:

* Si x se clasifica como una variable:
* x se evalúa para producir la variable.
* Se invoca el operador seleccionado con el valor de x como argumento.
* El valor devuelto por el operador se almacena en la ubicación dada por la evaluación de x.
* El valor devuelto por el operador es el resultado de la operación.
* Si x se clasifica como una propiedad o un acceso a indizador:
* Se evalúan la expresión de instancia (si x no es static) y la lista de argumentos (si x es un acceso a indizador) asociadas con x, y el resultado se utiliza en las posteriores invocaciones de descriptor de acceso get y set.
* Se invoca el descriptor de acceso get de x.
* Se invoca el operador seleccionado con el valor devuelto por el descriptor de acceso get como argumento.
* Se invoca el descriptor de acceso set de x con el valor devuelto por el operador como argumento value.
* El valor devuelto por el operador es el resultado de la operación.

Los operadores ++ y -- admiten la notación de postfijos (§7.6.9). Normalmente, el resultado de x++ o x-- es el valor de x antes de la operación, mientras que el resultado de ++x o --x es el valor de x después de la operación. En uno u otro caso, x tiene el mismo valor después de la operación.

Una implementación de operator ++ u operator -- puede invocarse mediante la notación de postfijo o prefijo. No es posible contar con implementaciones de operador independientes para las dos notaciones.

### Expresiones de conversión

Una expresión de conversión de tipos (cast-expression) se usa para convertir explícitamente una expresión a un tipo dado.

cast-expression:  
( type ) unary-expression

Una expresión de conversión cast (cast-expression) de la forma (T)E, donde T es un tipo (type) y E es una expresión unaria (unary-expression), realiza una conversión explícita (§6.2) del valor de E al tipo T. Si no existe una conversión explícita del tipo E al tipo T, se produce un error en tiempo de enlace. En caso contrario, el resultado es el valor producido por la conversión explícita. El resultado siempre se clasifica como un valor, aunque E denote una variable.

La gramática de una expresión de conversión de tipos (cast-expression) produce algunas ambigüedades sintácticas. Por ejemplo, la expresión (x)–y podría interpretarse como una expresión de conversión de tipos (cast-expression) (una conversión del tipo –y al tipo x) o como una expresión aditiva (additive-expression) combinada con una expresión entre paréntesis (parenthesized-expression) (que calcula el valor de x – y)).

Para resolver las ambigüedades de las expresiones de conversión de tipos (cast-expression), existe la siguiente regla: una secuencia de uno o más tokens (§2.3.3) encerrados entre paréntesis se considera el inicio de una expresión de conversión de tipos (cast-expression) solo si al menos uno de los siguientes supuestos es cierto:

* La secuencia de tokens es correcta gramaticalmente para un tipo (type), pero no para una expresión (expression).
* La secuencia de tokens es correcta gramaticalmente para un tipo (type), y el token que sigue inmediatamente al paréntesis de cierre es el token “~”, “!” o “(”, un identificador (identifier) (§2.4.1), un literal (§2.4.4) o cualquier palabra clave (keyword) (§2.4.3) excepto as e is.

El término “gramaticalmente correcta” significa que la secuencia de símbolos (tokens) debe ajustarse a la producción gramatical particular de esta clase de expresiones. En concreto, no se considera al significado real de sus identificadores constituyentes. Por ejemplo, si x e y son identificadores, entonces x.y es correcta gramaticalmente para un tipo, aunque x.y no denote realmente un tipo.

De la regla de eliminación de ambigüedades se deduce que, si x e y son identificadores, (x)y, (x)(y) y (x)(-y) son expresiones de conversión de tipos (cast-expressions), pero no así (x)-y, aunque x identifique un tipo. No obstante, si x es una palabra clave que identifica un tipo predefinido (como int), las cuatro formas son expresiones de conversión de tipos (cast-expressions) (porque esta palabra clave nunca podría ser una expresión por sí misma).

### Expresiones await

El operador await se usa para suspender la evaluación de la función asincrónica envolvente hasta que se haya completado la operación asincrónica representada por el operando.

await-expression:  
await unary-expression

Una expresión await (await-expression) solo se permite en el cuerpo de una función asincrónica (§10.14). Dentro de la función asincrónica envolvente más próxima, una expresión await (await-expression) no puede aparecer en estos lugares:

* Dentro de una función anónima (no asincrónica) anidada
* En un bloque catch o finally de una instrucción try (try-statement)
* Dentro de un bloque de una instrucción lock (lock-statement)
* En un contexto no seguro

Debe tenerse en cuenta que una expresión await (await-expression) no puede aparecer en la mayoría de los casos dentro de las expresiones de consulta (query-expression), porque estas se transforman sintácticamente para usar expresiones lambda no asincrónicas.

Dentro de una función asincrónica, await no se puede usar como identificador. Por tanto, no existe ambigüedad sintáctica entre las expresiones await y las diversas expresiones que incluyen identificadores. Aparte de las funciones asincrónicas, await actúa como un identificador normal.

El operando de una expresión await (await-expression) se denomina tarea. Representa una operación asincrónica que puede o no estar finalizada en el momento en que se evalúa la expresión await (await-expression). La finalidad del operador await es suspender la ejecución de la función asincrónica envolvente mientras se espera a que finalice la tarea y, a continuación, obtener su resultado.

#### Expresiones que admiten await

La tarea de una expresión await tiene que admitir await. Una expresión *t* admite await si se cumple alguna de las condiciones siguientes:

* *t* is of compile time type dynamic
* *t* has an accessible instance or extension method called GetAwaiter with no parameters and no type parameters, and a return type *A* for which all of the following hold:
* *A* implementa la interfaz System.Runtime.CompilerServices.INotifyCompletion (de aquí en adelante INotifyCompletion para mayor brevedad)
* *A* tiene una propiedad de instancia accesible y legible IsCompleted de tipo bool
* *A* tiene un método de instancia accesible GetResult sin parámetros ni parámetros de tipo

La finalidad del método GetAwaiter es obtener un awaiter para la tarea. El tipo *A* se denomina tipo awaiter en las expresiones await.

La finalidad de la propiedad IsCompleted es determinar si la tarea ya se ha completado. En tal caso, no es necesario suspender la evaluación.

La finalidad del método INotifyCompletion.OnCompleted es inscribir una “continuación” para la tarea; es decir, un delegado (de tipo System.Action) al que se llamará una vez completada la tarea.

La finalidad del método GetResult es obtener el resultado de la tarea una vez completada. Este resultado puede ser una finalización correcta, posiblemente con un valor de resultado, o puede ser una excepción iniciada por el método GetResult.

#### Clasificaciones de las expresiones await

La expresión await *t* se clasifica de la misma forma que la expresión (*t*).GetAwaiter().GetResult(). Por tanto, si el tipo devuelto de GetResult es void, la expresión await (await-expression) se clasifica como nada. Si tiene un tipo devuelto *T* que no es void, la expresión await (await-expression) se clasifica como un valor de tipo *T*.

#### Evaluación de expresiones await en tiempo de ejecución

En tiempo de ejecución, la expresión await *t* se evalúa de la forma siguiente:

* Se obtiene un awaiter *a* al evaluar la expresión (*t*).GetAwaiter().
* Se obtiene un bool *b* al evaluar la expresión (*a*).IsCompleted.
* Si *b* es false la evaluación depende de si *a* implementa la interfaz System.Runtime.CompilerServices.ICriticalNotifyCompletion (de aquí en adelante ICriticalNotifyCompletion para mayor brevedad). Esta comprobación se realiza en tiempo de enlace; o bien en tiempo de ejecución si *a* tiene el tipo de tiempo de compilación dynamic, y en tiempo de compilación en los demás casos. Supongamos que *r* denota el delegado de reanudación (§10.14):
* Si *a* no implementa ICriticalNotifyCompletion, se evalúa la expresión   
  ((*a*) as INotifyCompletion).OnCompleted(*r*).
* Si *a* implementa ICriticalNotifyCompletion, se evalúa la expresión   
  ((*a*) as ICriticalNotifyCompletion).UnsafeOnCompleted(*r*).
* Después se suspende la evaluación y el control se devuelve al llamador actual de la función asincrónica.
* Ya sea inmediatamente después (si *b* era true), o en una invocación posterior del delegado de reanudación (si *b* era false), se evalúa la expresión (*a*).GetResult(). Si devuelve un valor, ese valor es el resultado de la expresión await (*await-expression*). De lo contrario, el resultado es nada.

Una implementación del awaiter para los métodos de interfaz INotifyCompletion.OnCompleted y ICriticalNotifyCompletion.UnsafeOnCompleted debe causar que se llame al delegado *r* una vez a lo sumo. De lo contrario, el comportamiento de la función asincrónica envolvente está sin definir.

## Operadores aritméticos

Los operadores \*, /, %, + y – se denominan operadores aritméticos.

multiplicative-expression:  
unary-expression  
multiplicative-expression \* unary-expression  
multiplicative-expression / unary-expression  
multiplicative-expression % unary-expression

additive-expression:  
multiplicative-expression  
additive-expression + multiplicative-expression  
additive-expression – multiplicative-expression

Si el operando de una expresión aritmética tiene el tipo dynamic en tiempo de compilación, se enlaza dinámicamente (§7.2.2). En este caso, el tipo en tiempo de compilación de la expresión es dynamic y la resolución descrita a continuación se produce en tiempo de ejecución usando el tipo en tiempo de ejecución de esos operandos que tienen tipo dynamic en tiempo de compilación.

### Operador de multiplicación

Para una operación con la forma x \* y, se aplica la resolución de sobrecargas de operadores binarios (§7.3.4) para seleccionar una implementación de operador concreta. Los operandos se convierten a los tipos de parámetro del operador seleccionado, y el tipo del resultado es el tipo de valor devuelto por el operador.

A continuación se enumeran los operadores de multiplicación predefinidos. Todos los operadores calculan el producto de x e y.

* Multiplicación de enteros:

int operator \*(int x, int y);  
uint operator \*(uint x, uint y);  
long operator \*(long x, long y);  
ulong operator \*(ulong x, ulong y);

En un contexto checked, si el producto está fuera del intervalo del tipo del resultado, se inicia una excepción System.OverflowException. En un contexto unchecked, no se notifican los desbordamientos y se descarta cualquier bit significativo de nivel superior del resultado que esté fuera del intervalo del tipo de resultado.

* Multiplicación de números de punto flotante:

float operator \*(float x, float y);  
double operator \*(double x, double y);

El producto se calcula según las reglas de aritmética IEEE 754. La tabla siguiente muestra los resultados de todas las posibles combinaciones de valores finitos distintos de cero, ceros, infinitos y NaN. En la tabla, x e y son valores finitos positivos. z es el resultado de x \* y. Si el resultado es demasiado grande para el tipo de destino, z es infinito. Si el resultado es demasiado pequeño para el tipo de destino, z es cero.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | +y | –y | +0 | –0 | +∞ | –∞ | NaN |
| +x | +z | –z | +0 | –0 | +∞ | –∞ | NaN |
| –x | –z | +z | –0 | +0 | –∞ | +∞ | NaN |
| +0 | +0 | –0 | +0 | –0 | NaN | NaN | NaN |
| –0 | –0 | +0 | –0 | +0 | NaN | NaN | NaN |
| +∞ | +∞ | –∞ | NaN | NaN | +∞ | –∞ | NaN |
| –∞ | –∞ | +∞ | NaN | NaN | –∞ | +∞ | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Multiplicación de números decimales:

decimal operator \*(decimal x, decimal y);

Si el valor resultante es demasiado grande para representarlo en formato decimal, se produce una excepción System.OverflowException. Si el valor resultante es demasiado pequeño para representarlo en formato decimal, el resultado es cero. La escala del resultado, antes de cualquier redondeo, es la suma de las escalas de los dos operandos.

La multiplicación de decimales equivale al uso del operador de multiplicación de tipo System.Decimal.

### Operador de división

Para una operación con la forma x / y, se aplica la resolución de sobrecargas de operadores binarios (§7.3.4) para seleccionar una implementación de operador concreta. Los operandos se convierten a los tipos de parámetro del operador seleccionado, y el tipo del resultado es el tipo de valor devuelto por el operador.

A continuación se enumeran los operadores de división predefinidos. Todos los operadores calculan el cociente de x e y.

* División de números enteros:

int operator /(int x, int y);  
uint operator /(uint x, uint y);  
long operator /(long x, long y);  
ulong operator /(ulong x, ulong y);

Si el valor del operando derecho es cero, se produce una excepción System.DivideByZeroException.

La división redondea el resultado hacia cero. Por tanto, el valor absoluto del resultado es el entero mayor posible que sea menor o igual que el valor absoluto del cociente de los dos operandos. El resultado es cero o positivo cuando los dos operandos tienen el mismo signo, y cero o negativo si los dos operandos tienen signos opuestos.

Si el operando izquierdo es el int o long representable menor y el derecho es –1, se produce un desbordamiento. En un contexto checked, esto causa que se inicie una excepción System.ArithmeticException (o una subclase de la misma). En un contexto unchecked, la implementación define si se inicia una excepción System.ArithmeticException (o una subclase de la misma) o no se informa del desbordamiento, siendo el valor resultante el del operando izquierdo.

* División de números de punto flotante:

float operator /(float x, float y);  
double operator /(double x, double y);

El cociente se calcula según las reglas de aritmética IEEE 754. La tabla siguiente muestra los resultados de todas las posibles combinaciones de valores finitos distintos de cero, ceros, infinitos y NaN. En la tabla, x e y son valores finitos positivos. z es el resultado de x / y. Si el resultado es demasiado grande para el tipo de destino, z es infinito. Si el resultado es demasiado pequeño para el tipo de destino, z es cero.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | +y | –y | +0 | –0 | +∞ | –∞ | NaN |
| +x | +z | –z | +∞ | –∞ | +0 | –0 | NaN |
| –x | –z | +z | –∞ | +∞ | –0 | +0 | NaN |
| +0 | +0 | –0 | NaN | NaN | +0 | –0 | NaN |
| –0 | –0 | +0 | NaN | NaN | –0 | +0 | NaN |
| +∞ | +∞ | –∞ | +∞ | –∞ | NaN | NaN | NaN |
| –∞ | –∞ | +∞ | –∞ | +∞ | NaN | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* División de números decimales:

decimal operator /(decimal x, decimal y);

Si el valor del operando derecho es cero, se produce una excepción System.DivideByZeroException. Si el valor resultante es demasiado grande para representarlo en formato decimal, se produce una excepción System.OverflowException. Si el valor resultante es demasiado pequeño para representarlo en formato decimal, el resultado es cero. La escala del resultado será la menor escala que conserve un resultado igual al valor decimal representable más cercano al auténtico valor matemático.

La división de decimales equivale al uso del operador de división de tipo System.Decimal.

### Operador de resto

Para una operación con la forma x % y, se aplica la resolución de sobrecargas de operadores binarios (§7.3.4) para seleccionar una implementación de operador concreta. Los operandos se convierten a los tipos de parámetro del operador seleccionado, y el tipo del resultado es el tipo de valor devuelto por el operador.

A continuación se enumeran los operadores de resto predefinidos. Todos los operadores calculan el resto de la división entre x e y.

* Resto de números enteros:

int operator %(int x, int y);  
uint operator %(uint x, uint y);  
long operator %(long x, long y);  
ulong operator %(ulong x, ulong y);

El resultado de x % y es el valor producido por x – (x / y) \* y. Si y es cero, se inicia una excepción System.DivideByZeroException.

Si el operando izquierdo es el valor int o long menor y el operando derecho es -1, se inicia una excepción System.OverflowException. En ningún caso x % y inicia una excepción donde x / y no iniciaría una excepción.

* Resto de números de punto flotante:

float operator %(float x, float y);  
double operator %(double x, double y);

La tabla siguiente muestra los resultados de todas las posibles combinaciones de valores finitos distintos de cero, ceros, infinitos y NaN. En la tabla, x e y son valores finitos positivos. z es el resultado de x % y y se calcula como x – n \* y, donde n es el mayor entero posible que sea menor o igual que x / y. Este método para calcular el resto es análogo al utilizado para los operandos enteros, pero difiere de la definición de IEEE 754 (en la que n es el entero más próximo a x / y).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | +y | –y | +0 | –0 | +∞ | –∞ | NaN |
| +x | +z | +z | NaN | NaN | x | x | NaN |
| –x | –z | –z | NaN | NaN | –x | –x | NaN |
| +0 | +0 | +0 | NaN | NaN | +0 | +0 | NaN |
| –0 | –0 | –0 | NaN | NaN | –0 | –0 | NaN |
| +∞ | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| –∞ | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Resto de números decimales:

decimal operator %(decimal x, decimal y);

Si el valor del operando derecho es cero, se produce una excepción System.DivideByZeroException. La escala del resultado, antes de cualquier redondeo, es la mayor de las escalas de los dos operandos; el signo del resultado, si es distinto de cero, es el mismo que el de x.

El resto decimal equivale al uso del operador de resto de tipo System.Decimal.

### Operador de suma

Para una operación con la forma x + y, se aplica la resolución de sobrecargas de operadores binarios (§7.3.4) para seleccionar una implementación de operador concreta. Los operandos se convierten a los tipos de parámetro del operador seleccionado, y el tipo del resultado es el tipo de valor devuelto por el operador.

A continuación se enumeran los operadores de suma predefinidos. Para tipos numéricos y de enumeración, los operadores de suma predefinidos calculan la suma de los dos operandos. Cuando al menos uno de los operandos es de tipo string, los operadores de suma predefinidos concatenan las representaciones de la cadena de los operandos.

* Suma de números enteros:

int operator +(int x, int y);  
uint operator +(uint x, uint y);  
long operator +(long x, long y);  
ulong operator +(ulong x, ulong y);

En un contexto checked, si la suma está fuera del intervalo del tipo del resultado, se inicia una excepción System.OverflowException. En un contexto unchecked, no se notifican los desbordamientos y se descarta cualquier bit significativo de nivel superior del resultado que esté fuera del intervalo del tipo de resultado.

* Suma de números de punto flotante:

float operator +(float x, float y);  
double operator +(double x, double y);

La suma se calcula según las reglas de aritmética IEEE 754. La tabla siguiente muestra los resultados de todas las posibles combinaciones de valores finitos distintos de cero, ceros, infinitos y NaN. En la tabla, x e y son valores finitos distintos de cero y z es el resultado de x + y. Si x e y tienen la misma magnitud pero signos opuestos, z es cero positivo. Si x + y es demasiado grande para representarlo en el tipo de destino, z es un infinito con el signo de x + y.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | y | +0 | –0 | +∞ | –∞ | NaN |
| x | z | x | x | +∞ | –∞ | NaN |
| +0 | y | +0 | +0 | +∞ | –∞ | NaN |
| –0 | y | +0 | –0 | +∞ | –∞ | NaN |
| +∞ | +∞ | +∞ | +∞ | +∞ | NaN | NaN |
| –∞ | –∞ | –∞ | –∞ | NaN | –∞ | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Suma de números decimales:

decimal operator +(decimal x, decimal y);

Si el valor resultante es demasiado grande para representarlo en formato decimal, se produce una excepción System.OverflowException. La escala del resultado, antes de cualquier redondeo, es la mayor de las escalas de los dos operandos.

La suma de decimales equivale al uso del operador de suma de tipo System.Decimal.

* Suma de enumeraciones. Todos los tipos de enumeración proporcionan de forma implícita los siguientes operadores predefinidos, donde E es el tipo enum y U es el tipo subyacente de E:

E operator +(E x, U y);  
E operator +(U x, E y);

En tiempo de ejecución, los operadores se evalúan exactamente como (E)((U)x + (U)y).

* Concatenación de cadenas:

string operator +(string x, string y);  
string operator +(string x, object y);  
string operator +(object x, string y);

Estas sobrecargas del operador binario + concatenan cadenas. Si un operando de la concatenación de cadenas es null, se sustituye una cadena vacía. O bien, cualquier argumento que no sea de cadena se convierte a su representación en formato de cadena mediante la invocación del método virtual ToString heredado del tipo object. Si ToString devuelve null, se sustituye una cadena vacía.

using System;

class Test  
{  
 static void Main() {  
 string s = null;  
 Console.WriteLine("s = >" + s + "<"); // displays s = ><  
 int i = 1;  
 Console.WriteLine("i = " + i); // displays i = 1  
 float f = 1.2300E+15F;  
 Console.WriteLine("f = " + f); // displays f = 1.23E+15  
 decimal d = 2.900m;  
 Console.WriteLine("d = " + d); // displays d = 2.900  
 }  
}

El resultado del operador de concatenación de cadenas es una cadena formada por los caracteres del operando izquierdo seguidos de los caracteres del operando derecho. El operador de concatenación de cadenas nunca devuelve un valor null. Puede producirse una excepción System.OutOfMemoryException si no hay suficiente memoria libre para asignar la cadena resultante.

* Combinación de delegados. Todos los tipos delegados proporcionan implícitamente el siguiente operador predefinido, donde D es el tipo delegado:

D operator +(D x, D y);

El operador + realiza la combinación de delegados cuando los dos operandos son de un tipo delegado D. Si los operandos tienen tipos de delegado distintos, se produce un error en tiempo de enlace. Si el primer operando es null, el resultado de la operación es el valor del segundo operando (aunque este operando también sea null). En caso contrario, si el segundo operando es null, el resultado de la operación es el valor del primer operando. O bien, el resultado de la operación es una nueva instancia de delegado que, cuando se invoca, llama al primer operando y después al segundo. Para obtener ejemplos de combinación de delegados, vea §7.8.5 y §15.4. Puesto que System.Delegate no es un tipo delegado, no tiene definido operator +.

### Operador de resta

Para una operación con la forma x – y, se aplica la resolución de sobrecargas de operadores binarios (§7.3.4) para seleccionar una implementación de operador concreta. Los operandos se convierten a los tipos de parámetro del operador seleccionado, y el tipo del resultado es el tipo de valor devuelto por el operador.

A continuación se enumeran los operadores de resta predefinidos. Todos los operadores restan y de x.

* Resta de enteros:

int operator –(int x, int y);  
uint operator –(uint x, uint y);  
long operator –(long x, long y);  
ulong operator –(ulong x, ulong y);

En un contexto checked, si la diferencia está fuera del intervalo del tipo del resultado, se inicia una excepción System.OverflowException. En un contexto unchecked, no se notifican los desbordamientos y se descarta cualquier bit significativo de nivel superior del resultado que esté fuera del intervalo del tipo de resultado.

* Resta de números de punto flotante:

float operator –(float x, float y);  
double operator –(double x, double y);

La diferencia se calcula según las reglas de aritmética IEEE 754. La tabla siguiente muestra los resultados de todas las posibles combinaciones de valores finitos distintos de cero, ceros, infinitos y NaN. En la tabla, x e y son valores finitos distintos de cero y z es el resultado de x – y. Si x e y son iguales, z es positivo. Si x – y es demasiado grande para representarlo en el tipo de destino, z es un infinito con el signo de x – y.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | y | +0 | –0 | +∞ | –∞ | NaN |
| x | z | x | x | –∞ | +∞ | NaN |
| +0 | –y | +0 | +0 | –∞ | +∞ | NaN |
| –0 | –y | –0 | +0 | –∞ | +∞ | NaN |
| +∞ | +∞ | +∞ | +∞ | NaN | +∞ | NaN |
| –∞ | –∞ | –∞ | –∞ | –∞ | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Resta de números decimales:

decimal operator –(decimal x, decimal y);

Si el valor resultante es demasiado grande para representarlo en formato decimal, se produce una excepción System.OverflowException. La escala del resultado, antes de cualquier redondeo, es la mayor de las escalas de los dos operandos.

La resta de decimales equivale al uso del operador de resta de tipo System.Decimal.

* Resta de enumeraciones. Todos los tipos de enumeración proporcionan de forma implícita el siguiente operador predefinido, donde E es el tipo enum y U es el tipo subyacente de E:

U operator –(E x, E y);

Este operador se evalúa exactamente como (U)((U)x – (U)y). Es decir, el operador calcula la diferencia entre los valores ordinales de x e y, y el tipo del resultado es el tipo subyacente de la enumeración.

E operator –(E x, U y);

Este operador se evalúa exactamente como (E)((U)x – y). Es decir, el operador resta un valor del tipo subyacente de la enumeración, que produce un valor de la enumeración.

* Eliminación de delegados. Todos los tipos delegados proporcionan implícitamente el siguiente operador predefinido, donde D es el tipo delegado:

D operator –(D x, D y);

El operador binario – realiza la eliminación de delegados cuando los dos operandos son de un tipo delegado D. Si los operandos tienen tipos de delegado distintos, se produce un error en tiempo de enlace. Si el primer operando es null, el resultado de la operación también es null. En caso contrario, si el segundo operando es null, el resultado de la operación es el valor del primer operando. De lo contrario, ambos operandos representan listas de invocaciones (§15.1) que tienen una o más entradas, y el resultado es una nueva lista de invocaciones que se compone de la lista del primer operando con las entradas del segundo operando quitadas de ella, siempre que la lista del segundo operando sea una sublista apropiada contigua al primero. (Para determinar la igualdad de la sublista, las entradas correspondientes se comparan en cuanto al operador de igualdad del delegado (§7.10.8)). En caso contrario, el resultado es el valor del operando izquierdo. En el proceso no se modifica la lista de ninguno de los operandos. Si la lista del segundo operando coincide con varias sublistas de entradas contiguas de la lista del primero, se quita la sublista coincidente situada más a la derecha de las entradas contiguas. Si la eliminación produce una lista vacía, el resultado es null. Por ejemplo:

delegate void D(int x);

class C  
{  
 public static void M1(int i) { /\* … \*/ }  
 public static void M2(int i) { /\* … \*/ }  
}

class Test  
{  
 static void Main() {   
 D cd1 = new D(C.M1);  
 D cd2 = new D(C.M2);  
 D cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd1; // => M1 + M2 + M2

cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd1 + cd2; // => M2 + M1

cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd2 + cd2; // => M1 + M1

cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd2 + cd1; // => M1 + M2

cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd1 + cd1; // => M1 + M2 + M2 + M1  
 }  
}

## Operadores de desplazamiento

Los operadores << y >> permiten realizar operaciones de desplazamiento de bits.

shift-expression:  
additive-expression   
shift-expression << additive-expression  
shift-expression right-shift additive-expression

Si el operando de una expresión de desplazamiento (shift-expression) tiene el tipo dynamic en tiempo de compilación, la expresión se enlaza dinámicamente (§7.2.2). En este caso, el tipo en tiempo de compilación de la expresión es dynamic y la resolución descrita a continuación se produce en tiempo de ejecución usando el tipo en tiempo de ejecución de esos operandos que tienen tipo dynamic en tiempo de compilación.

Para una operación con la forma x << count o x >> count, se aplica la resolución de sobrecargas de operadores binarios (§7.3.4) para seleccionar una implementación de operador concreta. Los operandos se convierten a los tipos de parámetro del operador seleccionado, y el tipo del resultado es el tipo de valor devuelto por el operador.

Cuando se declara un operador de desplazamiento sobrecargado, el tipo del primer operando siempre debe ser la clase o struct que contiene su declaración, mientras que el tipo del segundo operando siempre debe ser int.

A continuación se enumeran los operadores de desplazamiento predefinidos.

* Desplazamiento a la izquierda:

int operator <<(int x, int count);  
uint operator <<(uint x, int count);  
long operator <<(long x, int count);  
ulong operator <<(ulong x, int count);

El operador << desplaza x a la izquierda el número de bits calculados como se explica a continuación.

Los bits de orden superior no comprendidos en el intervalo del tipo de resultado de x se descartan, los bits restantes se desplazan a la izquierda y las posiciones vacías de los bits de orden inferior se establecen en cero.

* Desplazamiento a la derecha:

int operator >>(int x, int count);  
uint operator >>(uint x, int count);  
long operator >>(long x, int count);  
ulong operator >>(ulong x, int count);

El operador >> desplaza x a la derecha el número de bits calculados como se explica a continuación.

Si x es de tipo int o long, los bits de orden inferior de x se descartan, los bits restantes se desplazan a la derecha y las posiciones vacías de los bits de orden superior se establecen en cero si x no es negativo, y en uno si x es negativo.

Si x es de tipo uint o ulong, los bits de orden inferior de x se descartan, los bits restantes se desplazan a la derecha y las posiciones vacías de los bits de orden superior se establecen en cero.

Para los operadores predefinidos, el número de bits del desplazamiento se calcula como se explica a continuación:

* Si el tipo de x es int o uint, el valor del desplazamiento viene dado por los cinco bits de orden inferior de count. Es decir, el recuento del desplazamiento se calcula a partir de count & 0x1F.
* Si el tipo de x es long o ulong, el valor del desplazamiento viene dado por los seis bits de orden inferior de count. Es decir, el recuento del desplazamiento se calcula a partir de count & 0x3F.

Si el valor del desplazamiento resultante es cero, los operadores de desplazamiento sencillamente devuelven el valor de x.

Las operaciones de desplazamiento nunca causan desbordamientos y producen el mismo resultado en los contextos checked y unchecked.

Si el operando izquierdo del operador >> es de un tipo entero con signo, el operador realiza un desplazamiento aritmético a la derecha, en el cual el valor del bit más significativo (el bit de signo) del operando se propaga a las posiciones vacías de los bits de orden superior. Si el operando izquierdo del operador >> es de un tipo entero sin signo, el operador realiza un desplazamiento lógico a la derecha, en el cual las posiciones vacías de los bits de orden superior siempre se establecen en cero. Para realizar la operación opuesta de la inferida a partir del tipo del operando, pueden utilizarse conversiones explícitas. Por ejemplo, si x es una variable de tipo int, la operación unchecked((int)((uint)x >> y)) realiza un desplazamiento lógico a la derecha de x.

## Operadores de comprobación de tipos y relacionales

Los operadores ==, !=, <, >, <=, >=, is y as se denominan operadores relacionales y de comprobación de tipos.

relational-expression:  
shift-expression  
relational-expression < shift-expression  
relational-expression > shift-expression  
relational-expression <= shift-expression  
relational-expression >= shift-expression  
relational-expression is type  
relational-expression as type

equality-expression:  
relational-expression  
equality-expression == relational-expression  
equality-expression != relational-expression

El operador is se describe en §7.10.10, y el operador as en §7.10.11.

Los operadores ==, !=, <, >, <= y >= son operadores de comparación.

Si el operando de un operador de comparación tiene el tipo dynamic en tiempo de compilación, se enlaza dinámicamente (§7.2.2). En este caso, el tipo en tiempo de compilación de la expresión es dynamic y la resolución descrita a continuación se produce en tiempo de ejecución usando el tipo en tiempo de ejecución de esos operandos que tienen tipo dynamic en tiempo de compilación.

Para una operación con la forma x op y, donde op es un operador de comparación, se aplica la resolución de sobrecargas de operadores binarios (§7.3.4) para seleccionar una implementación de operador concreta. Los operandos se convierten a los tipos de parámetro del operador seleccionado, y el tipo del resultado es el tipo de valor devuelto por el operador.

Los operadores de comparación predefinidos se describen en las siguientes secciones. Todos los operadores de comparación predefinidos devuelven un resultado de tipo bool, como se muestra en la tabla siguiente.

|  |  |
| --- | --- |
| **Operación** | **Resultado** |
| x == y | true si x es igual a y, de lo contrario false |
| x != y | true si x no es igual a y, de lo contrario false |
| x < y | true si x es menor que y, de lo contrario false |
| x > y | true si x es mayor que y, de lo contrario false |
| x <= y | true si x es menor o igual que y, de lo contrario false |
| x >= y | true si x es mayor o igual que y, de lo contrario false |

### Operadores de comparación de enteros

Los operadores de comparación de enteros predefinidos son:

bool operator ==(int x, int y);  
bool operator ==(uint x, uint y);  
bool operator ==(long x, long y);  
bool operator ==(ulong x, ulong y);

bool operator !=(int x, int y);  
bool operator !=(uint x, uint y);  
bool operator !=(long x, long y);  
bool operator !=(ulong x, ulong y);

bool operator <(int x, int y);  
bool operator <(uint x, uint y);  
bool operator <(long x, long y);  
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);  
bool operator >(uint x, uint y);  
bool operator >(long x, long y);  
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);  
bool operator <=(uint x, uint y);  
bool operator <=(long x, long y);  
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);  
bool operator >=(uint x, uint y);  
bool operator >=(long x, long y);  
bool operator >=(ulong x, ulong y);

Todos estos operadores comparan los valores numéricos de los dos operandos enteros y devuelven un valor bool que indica si la relación concreta es true o false.

### Operadores de comparación de punto flotante

Los operadores de comparación de punto flotante predefinidos son:

bool operator ==(float x, float y);  
bool operator ==(double x, double y);

bool operator !=(float x, float y);  
bool operator !=(double x, double y);

bool operator <(float x, float y);  
bool operator <(double x, double y);

bool operator >(float x, float y);  
bool operator >(double x, double y);

bool operator <=(float x, float y);  
bool operator <=(double x, double y);

bool operator >=(float x, float y);  
bool operator >=(double x, double y);

Los operadores comparan los operandos según las reglas del estándar IEEE 754:

* Si uno de los operandos es NaN, el resultado es false para todos los operadores excepto !=, cuyo resultado es true. Para dos operandos cualesquiera, x != y siempre produce el mismo resultado que !(x == y). No obstante, si uno o los dos operandos son NaN, los operadores <, >, <= y >= no producen el mismo resultado que la negación lógica del operador opuesto. Por ejemplo, si x e y son NaN, entonces x < y es false, pero !(x >= y) es true.
* Si ninguno de los operandos es NaN, los operadores comparan los valores de los dos operandos de punto flotante con respecto al orden

–∞ < –max < ... < –min < –0.0 == +0.0 < +min < ... < +max < +∞

donde min y max son los valores finitos positivos máximo y mínimo que pueden representarse en el formato de punto flotante. Son efectos notables de este orden:

* El cero negativo y el positivo se consideran iguales.
* Un infinito negativo se considera menor que todos los demás valores, pero igual que otro infinito negativo.
* Un infinito positivo se considera mayor que todos los demás valores, pero igual que otro infinito positivo.

### Operadores de comparación decimales

Los operadores de comparación de decimales predefinidos son:

bool operator ==(decimal x, decimal y);

bool operator !=(decimal x, decimal y);

bool operator <(decimal x, decimal y);

bool operator >(decimal x, decimal y);

bool operator <=(decimal x, decimal y);

bool operator >=(decimal x, decimal y);

Todos estos operadores comparan los valores numéricos de los dos operandos decimales y devuelven un valor bool que indica si la relación concreta es true o false. Cada comparación de decimales equivale al uso del operador relacional o de igualdad correspondiente de tipo System.Decimal.

### Operadores de igualdad booleanos

Los operadores de igualdad booleanos predefinidos son:

bool operator ==(bool x, bool y);

bool operator !=(bool x, bool y);

El resultado de == es true si x e y son true o si x e y son false. De lo contrario, el resultado es false.

El resultado de != es false si x e y son true o si x e y son false. De lo contrario, el resultado es true. Si los operandos son de tipo bool, el operador != produce el mismo resultado que ^.

### Operadores de comparación de tipo de enumeración

Todos los tipos de enumeración proporcionan implícitamente los siguientes operadores de comparación predefinidos:

bool operator ==(E x, E y);

bool operator !=(E x, E y);

bool operator <(E x, E y);

bool operator >(E x, E y);

bool operator <=(E x, E y);

bool operator >=(E x, E y);

El resultado de evaluar x op y, donde x e y son expresiones de un tipo de enumeración E con un tipo subyacente U, y op es uno de los operadores de comparación, es exactamente el mismo que el de evaluar ((U)x) op ((U)y). Esto es lo mismo que decir que los operadores de comparación de tipo de enumeración sencillamente comparan los valores subyacentes integrales de los dos operandos.

### Operadores de igualdad de tipos de referencia

Los operadores de igualdad de tipos de referencia predefinidos son:

bool operator ==(object x, object y);

bool operator !=(object x, object y);

Los operadores devuelven el resultado de comparar la igualdad o desigualdad de las dos referencias.

Dado que los operadores de igualdad de tipos de referencia predefinidos aceptan operandos de tipo object, se aplican a todos los tipos que no declaran miembros aplicables operator == y operator !=. A la inversa, cualquier operador de igualdad aplicable definido por el usuario oculta los operadores de igualdad predefinidos de tipos de referencia.

Los operadores de igualdad de tipos de referencia predefinidos requieren uno de los siguientes puntos:

* Que los dos operandos sean un valor de un tipo conocido que sea de tipo de referencia (reference-type) o el literal null. Además, requieren que exista una conversión de referencia explícita (§6.2.4) del tipo de uno de los operandos al tipo del otro operando.
* Que un operando sea un valor de tipo T donde T es un parámetro de tipo (type-parameter) y el otro operando sea el literal null. Además, T no debe tener la restricción de tipo de valor.

A menos que una de estas dos condiciones sean verdaderas, se producirá un error de en tiempo de enlace. Son implicaciones notables de estas reglas:

* Produce un error en tiempo de enlace el utilizar los operadores de igualdad de tipos de referencia predefinidos para comparar dos referencias de las que se sabe que son diferentes en tiempo de compilación. Por ejemplo, si los tipos de los operandos en tiempo de enlace fueran dos tipos de clases A y B, y si ni A ni B se derivaran del otro, no sería posible que los operandos hicieran referencia al mismo objeto. Por lo tanto, la operación se considera un error en tiempo de enlace.
* Los operadores de igualdad de tipos de referencia predefinidos no permiten la comparación de operandos de tipo de valor. Por lo tanto, salvo que un tipo struct declare sus propios operadores de igualdad, no es posible comparar valores de ese tipo struct.
* Los operadores de igualdad de tipos de referencia predefinidos nunca causan operaciones boxing para sus operandos. No tendría sentido realizar este tipo de operaciones boxing, puesto que las referencias a las instancias convertidas mediante boxing recién asignadas diferirían necesariamente de todas las demás referencias.
* Si un operando de un tipo T de parámetro de tipo se compara con null, y el tipo en tiempo de ejecución de T es un tipo de valor, el resultado de la comparación es false.

En el siguiente ejemplo se comprueba si un argumento de un tipo de parámetro de tipo sin restricciones es null.

class C<T>  
{  
 void F(T x) {  
 if (x == null) throw new ArgumentNullException();  
 ...  
 }  
}

La construcción x == null se permite aunque T represente un tipo de valor y el resultado se defina simplemente como false cuando T es un tipo de valor.

Para una operación con la forma x == y o x != y, si existe un operador operator == o operator != aplicable, las reglas de resolución de sobrecargas de operador (§7.3.4) seleccionan este operador en lugar del operador de igualdad de tipos de referencia predefinido. No obstante, siempre es posible seleccionar el operador de igualdad de tipos de referencia predefinido mediante la conversión explícita de uno o los dos operandos al tipo object. El ejemplo

using System;

class Test  
{  
 static void Main() {  
 string s = "Test";  
 string t = string.Copy(s);  
 Console.WriteLine(s == t);  
 Console.WriteLine((object)s == t);  
 Console.WriteLine(s == (object)t);  
 Console.WriteLine((object)s == (object)t);  
 }  
}

produce el resultado

True  
False  
False  
False

Las variables s y t hacen referencia a las dos instancias únicas de string que contienen los mismos caracteres. La primera comparación produce True a causa de la selección del operador de igualdad de cadenas predefinido (§7.10.7) cuando los dos operandos son de tipo string. Todas las comparaciones restantes producen False a causa de la selección del operador de igualdad de tipos de referencia predefinido cuando uno o los dos operandos son de tipo object.

Téngase en cuenta que la técnica anterior no tiene sentido para los tipos de valor. El ejemplo

class Test  
{  
 static void Main() {  
 int i = 123;  
 int j = 123;  
 System.Console.WriteLine((object)i == (object)j);  
 }  
}

produce False porque las conversiones de tipos crean referencias a dos instancias distintas de valores int convertidos mediante boxing.

### Operadores de igualdad de cadenas

Los operadores de igualdad de cadenas predefinidos son:

bool operator ==(string x, string y);

bool operator !=(string x, string y);

Dos valores string se consideran iguales cuando una de las siguientes condiciones es verdadera:

* Los dos valores son null.
* Los dos valores son referencias no nulas a instancias de cadenas que tienen una longitud idéntica y caracteres idénticos en cada posición de carácter.

Los operadores de igualdad de cadenas comparan valores de cadenas y no referencias a cadenas. Cuando dos instancias de cadena distintas contienen exactamente la misma secuencia de caracteres, los valores de las cadenas son iguales, pero las referencias son diferentes. Como se explica en la §7.10.6, los operadores de igualdad de tipos de referencia sirve para comparar referencias de cadenas en lugar de valores de cadenas.

### Operadores de igualdad de delegados

Los operadores de igualdad de delegados predefinidos son:

bool operator ==(System.Delegate x, System.Delegate y);

bool operator !=(System.Delegate x, System.Delegate y);

Dos instancias de delegados se consideran iguales en los siguientes casos:

* Si una de las instancias de delegado es null, son iguales si y solamente si las dos son null.
* Si los dos delegados tienen un tipo en tiempo de ejecución diferente, no serán nunca iguales.
* Si ambas instancias de delegado tienen una lista de invocaciones (§15.1), dichas instancias son iguales si y solamente si sus listas de invocaciones tienen la misma longitud, y cada entrada de la lista de invocaciones de una es igual a la entrada correspondiente (como se define a continuación), por orden, de la lista de invocaciones de la otra.

Las entradas de las listas de invocaciones deben cumplir las siguientes reglas:

* Si dos entradas de las listas de invocaciones hacen referencia al mismo método estático, entonces las entradas se consideran iguales.
* Si dos entradas de las listas de invocaciones hacen referencia al mismo método no estático en el mismo objeto de destino (tal como definen los operadores de igualdad de referencia), se considera que las entradas son iguales.
* Las entradas de las listas de invocación generadas en la evaluación de expresiones de funciones anónimas (anonymous-function-expression) de semántica idéntica con el mismo conjunto (posiblemente vacío) de instancias de variables externas capturadas pueden ser iguales (aunque no es obligatorio).

### Operadores de igualdad y NULL

Los operadores == y != permiten que un operando sea un valor de un tipo que acepta valores NULL y el otro el literal null, incluso si no existe un operador predefinido ni definido por el usuario (en formato de elevación o de no elevación) para la operación.

Para una operación con las estructuras

x == null null == x x != null null != x

donde x es una expresión de un tipo que acepta valores NULL, si la resolución de sobrecargas de operadores (§7.2.4) no encuentra un operador aplicable, el resultado se calcula a partir de la propiedad HasValue de x. En concreto, las primeras dos estructuras se traducen como !x.HasValue y las últimas dos estructuras se traducen como x.HasValue.

### Operador Is

El operador is se utiliza para comprobar dinámicamente si el tipo en tiempo de ejecución de un objeto es compatible con un tipo dado. El resultado de la operación E is T, donde E es una expresión y T es un tipo, es un valor booleano que indica si E puede convertirse de forma satisfactoria al tipo T mediante una conversión de referencias, una conversión boxing o una conversión unboxing. La operación se evalúa de la siguiente manera, una vez sustituidos los argumentos de tipo por todos los parámetros de tipo:

* Si E es una función anónima, se produce un error durante la compilación
* Si E es un grupo de métodos o el literal null, o si el tipo de E es un tipo de referencia o un tipo que acepta valores NULL y el valor de E es NULL, el resultado es false.
* De lo contrario, D representa el tipo dinámico de E de la siguiente manera:
* Si el tipo de E es un tipo de referencia, D es el tipo en tiempo de ejecución de la instancia a la que E hace referencia.
* Si el tipo de E es un tipo que acepta valores NULL, D es el tipo subyacente del tipo que acepta valores NULL.
* Si el tipo de E es un tipo de valor que no acepta valores NULL, D es el tipo de E.
* El resultado de la operación depende de D y T de la siguiente manera:
* Si T es un tipo de referencia, el resultado es true si D y T son el mismo tipo, si D es un tipo de referencia y existe una conversión de referencia implícita de D a T o si D es un tipo de valor y existe una conversión boxing de D a T.
* Si T es un tipo que acepta valores NULL, el resultado es true si D es el tipo subyacente de T.
* Si T es un tipo de valor que no acepta valores NULL, el resultado es true si D y T son el mismo tipo.
* De lo contrario, el resultado es false.

Observe que el operador is no tiene en cuenta las conversiones definidas por el usuario.

### Operador As

El operador as permite convertir explícitamente un valor a un tipo de referencia dado mediante una conversión de referencia o tipo que acepta valores NULL. A diferencia de una expresión de conversión de tipos (§7.7.6), el operador as nunca inicia una excepción. En lugar de ello, si la conversión indicada no es posible, el valor resultante es null.

En una operación con la estructura E as T, E debe ser una expresión y T debe ser un tipo de referencia, un parámetro de tipo conocido como un tipo de referencia o un tipo que acepta valores NULL. Al menos una de las siguientes afirmaciones debe ser true; de lo contrario, se genera un error en tiempo de compilación:

* Existe una conversión de identidad (§6.1.1), una conversión implícita que acepta valores NULL (§6.1.4), una conversión de referencia implícita (§6.1.6), una conversión boxing (§6.1.7), una conversión explícita que acepta valores NULL (§6.2.3), una conversión de referencia explícita (§6.2.4) o una conversión unboxing (§6.2.5) de E a T.
* El tipo de E o T es un tipo abierto.
* E es el literal null.

Si el tipo de E en tiempo de compilación no es dynamic, la operación E as T produce el mismo resultado que

E is T ? (T)(E) : (T)null

salvo que E solo se evalúa una vez. Se puede esperar que el compilador optimice E as T para realizar como máximo una comprobación de tipo dinámico por oposición a las dos comprobaciones de tipo dinámico implicadas por la expansión anterior.

Si el tipo en tiempo de compilación de E es dynamic, a diferencia del operador de conversión, el operador as no se enlaza dinámicamente (§7.2.2). Por tanto, la expansión en este caso es:

E is T ? (T)(object)(E) : (T)null

Tenga en cuenta que algunas conversiones, como las definidas por el usuario, no son posibles con el operador as y deben realizarse mediante expresiones de conversión de tipos (cast).

En el ejemplo

class X  
{

public string F(object o) {  
 return o as string; // OK, string is a reference type  
 }

public T G<T>(object o) where T: Attribute {  
 return o as T; // Ok, T has a class constraint  
 }

public U H<U>(object o) {  
 return o as U; // Error, U is unconstrained   
 }  
}

el parámetro de tipo T de G es un tipo de referencia, porque tiene la restricción de clase. Sin embargo, el parámetro de tipo U de H no lo es; por lo tanto, no se permite el uso del operador as en H.

## Operadores lógicos

Los operadores &, ^ y | se denominan operadores lógicos.

and-expression:  
equality-expression  
and-expression & equality-expression

exclusive-or-expression:  
and-expression  
exclusive-or-expression ^ and-expression

inclusive-or-expression:  
exclusive-or-expression  
inclusive-or-expression | exclusive-or-expression

Si el operando de una expresión lógica tiene el tipo dynamic en tiempo de compilación, la expresión se enlaza dinámicamente (§7.2.2). En este caso, el tipo en tiempo de compilación de la expresión es dynamic y la resolución descrita a continuación se produce en tiempo de ejecución usando el tipo en tiempo de ejecución de esos operandos que tienen tipo dynamic en tiempo de compilación.

Para una operación con la forma x op y, donde op es uno de los operadores lógicos, se aplica la resolución de sobrecargas de operadores binarios (§7.3.4) para seleccionar una implementación de operador concreta. Los operandos se convierten a los tipos de parámetro del operador seleccionado, y el tipo del resultado es el tipo de valor devuelto por el operador.

Los operadores lógicos predefinidos se describen en las siguientes secciones.

### Operadores lógicos enteros

Los operadores lógicos enteros predefinidos son:

int operator &(int x, int y);  
uint operator &(uint x, uint y);  
long operator &(long x, long y);  
ulong operator &(ulong x, ulong y);

int operator |(int x, int y);  
uint operator |(uint x, uint y);  
long operator |(long x, long y);  
ulong operator |(ulong x, ulong y);

int operator ^(int x, int y);  
uint operator ^(uint x, uint y);  
long operator ^(long x, long y);  
ulong operator ^(ulong x, ulong y);

El operador & calcula la operación lógica AND bit a bit de los dos operandos, el operador | calcula la operación lógica OR bit a bit de los dos operandos y el operador ^ calcula la operación lógica OR exclusiva bit a bit de los dos operandos. Los desbordamientos no son posibles en estas operaciones.

### Operadores lógicos de enumeración

Todo tipo de enumeración E proporciona implícitamente los siguientes operadores lógicos predefinidos:

E operator &(E x, E y);  
E operator |(E x, E y);  
E operator ^(E x, E y);

El resultado de evaluar x op y, donde x e y son expresiones de un tipo de enumeración E con un tipo subyacente U, y op es uno de los operadores lógicos, es exactamente el mismo que el de evaluar (E)((U)x op (U)y). Esto es lo mismo que decir que los operadores lógicos de tipo de enumeración sencillamente ejecutan la operación lógica en el tipo subyacente de los dos operandos.

### Operadores lógicos booleanos

Los operadores lógicos booleanos predefinidos son:

bool operator &(bool x, bool y);

bool operator |(bool x, bool y);

bool operator ^(bool x, bool y);

El resultado de x & y es true si x e y son true. De lo contrario, el resultado es false.

El resultado de x | y es true si x o y es true. De lo contrario, el resultado es false.

El resultado de x ^ y es true si x es true e y es false, o x es false e y es true. De lo contrario, el resultado es false. Si los operandos son de tipo bool, el operador ^ calcula el mismo resultado que el operador !=.

### Operadores lógicos booleanos que aceptan valores NULL

El tipo booleano bool? que acepta valores NULL puede representar tres valores, true, false y null, y es conceptualmente similar al tipo de tres valores utilizado por expresiones booleanas en SQL. Para asegurarse de que los resultados generados por los operadores & y | para los operandos bool? son coherentes con la lógica de tres valores de SQL se proporcionan los siguientes operadores predefinidos:

bool? operator &(bool? x, bool? y);

bool? operator |(bool? x, bool? y);

En la siguiente tabla se enumeran los resultados generados por estos operadores para todas las combinaciones de los valores true, false y null.

|  |  |  |  |
| --- | --- | --- | --- |
| x | y | x & y | x | y |
| true | true | true | true |
| true | false | false | true |
| true | null | null | true |
| false | true | false | true |
| false | false | false | false |
| false | null | false | null |
| null | true | null | true |
| null | false | false | null |
| null | null | null | null |

## Operadores lógicos condicionales

Los operadores && y || se denominan operadores lógicos condicionales. También se conocen como operadores lógicos de evaluación “cortocircuitada”.

conditional-and-expression:  
inclusive-or-expression  
conditional-and-expression && inclusive-or-expression

conditional-or-expression:  
conditional-and-expression  
conditional-or-expression || conditional-and-expression

Los operadores && y || son versiones condicionales de los operadores & y |:

* La operación x && y corresponde a la operación x & y, excepto que y se evalúa solamente si x no es false.
* La operación x || y corresponde a la operación x | y, excepto que y se evalúa solamente si x no es true.

Si el operando de una expresión lógica condicional tiene el tipo dynamic en tiempo de compilación, la expresión se enlaza dinámicamente (§7.2.2). En este caso, el tipo en tiempo de compilación de la expresión es dynamic y la resolución descrita a continuación se produce en tiempo de ejecución usando el tipo en tiempo de ejecución de esos operandos que tienen tipo dynamic en tiempo de compilación.

Una operación de la forma x && y o x || y se procesa mediante la aplicación de la resolución de sobrecargas (§7.3.4) como si la operación estuviera escrita como x & y o x | y. Entonces:

* Si la resolución de sobrecargas no encuentra un solo operador mejor o selecciona uno de los operadores lógicos enteros predefinidos, se produce un error en tiempo de enlace.
* En caso contrario, si el operador seleccionado es uno de los operadores lógicos booleanos predefinidos (§7.11.3) u operadores lógicos booleanos que aceptan valores NULL (§7.11.4), la operación se procesa como se explica en la sección §7.12.1.
* O bien, el operador seleccionado es un operador definido por el usuario y la operación se procesa como se explica en §7.12.2.

No es posible sobrecargar directamente los operadores lógicos condicionales. No obstante, dado que los operadores lógicos condicionales se evalúan en términos de los operadores lógicos regulares, las sobrecargas de éstos, con algunas restricciones, también se consideran sobrecargas de los operadores lógicos condicionales. Esta categoría se explica con más detalle en la sección §7.12.2.

### Operadores lógicos condicionales booleanos

Si los operandos de && o || son de tipo bool, o si los operandos son de tipos que no definen un operator & o un operator | aplicable, pero sí definen conversiones implícitas a bool, la operación se procesa como sigue:

* La operación x && y se evalúa como x ? y : false. En otras palabras, primero se evalúa x y se convierte al tipo bool. Después, si x es true, y se evalúa y se convierte al tipo bool, y se convierte en el resultado de la operación. De lo contrario, el resultado de la operación es false.
* La operación x || y se evalúa como x ? true : y. En otras palabras, primero se evalúa x y se convierte al tipo bool. Después, si el valor de x es true, el resultado de la operación es true. O bien, y se evalúa y se convierte al tipo bool, y pasa a ser el resultado de la operación.

### Operadores lógicos condicionales definidos por el usuario

Si los operandos de && o || son de tipos que declaran un operator & o un operator | aplicable definido por el usuario, las dos siguientes declaraciones deben ser verdaderas, donde T es el tipo en que se declara el operador seleccionado:

* El tipo del valor devuelto y el tipo de todos los parámetros del operador seleccionado debe ser T. Es decir, el operador debe calcular el AND lógico o el OR lógico de los dos operandos de tipo T, y debe devolver un resultado de tipo T.
* T debe contener declaraciones de operator true y operator false.

Si alguno de estos requisitos no se satisface, se produce un error en tiempo de enlace. O bien, la operación && o || se evalúa mediante la combinación del operator true o del operator false definido por el usuario con el operador seleccionado definido por el usuario:

* La operación x && y se evalúa como T.false(x) ? x : T.&(x, y), donde T.false(x) es una invocación del operator false declarado en T, y T.&(x, y) es una invocación del operator & seleccionado. En otras palabras, x es el primero en evaluarse y se invoca operator false en el resultado para averiguar si x es definitivamente false. Después, si x es definitivamente false, el resultado de la operación es el valor previamente calculado para x. De lo contrario, se evalúa y y se invoca el operator & seleccionado en el valor previamente calculado para x y el valor calculado para y, a fin de producir el resultado de la operación.
* La operación x || y se evalúa como T.true(x) ? x : T.|(x, y), donde T.true(x) es una invocación del operator true declarado en T, y T.|(x, y) es una invocación del operator | seleccionado. En otras palabras, x es el primero en evaluarse y se invoca operator true en el resultado para averiguar si x es definitivamente true. Después, si x es definitivamente true, el resultado de la operación es el valor previamente calculado para x. De lo contrario, se evalúa y y se invoca el operator | seleccionado en el valor previamente calculado para x y el valor calculado para y, a fin de producir el resultado de la operación.

En cualquiera de estas operaciones, la expresión dada por x se evalúa una sola vez, y la expresión dada por y no se evalúa o bien se evalúa exactamente una vez.

Para obtener un ejemplo de un tipo que implementa operator true y operator false, vea §11.4.2.

## El operador de incorporación de NULL

El operador ?? se denomina operador de uso combinado de NULL.

null-coalescing-expression:  
conditional-or-expression  
conditional-or-expression ?? null-coalescing-expression

Una expresión de uso combinado de NULL con la forma a ?? b requiere que a sea un tipo que acepta valores NULL o un tipo de referencia. Si a no es NULL, el resultado de a ?? b es a; de lo contrario, el resultado es b. La operación evalúa b solo si a es NULL.

El operador de uso combinado de NULL es asociativo por la derecha, lo que significa que las operaciones se agrupan de derecha a izquierda. Por ejemplo, una expresión con la forma a ?? b ?? c se evalúa como a ?? (b ?? c). En términos generales, una expresión con la forma E1 ?? E2 ?? ... ?? EN devuelve el primero de los operandos que no es NULL o que lo es si todos son NULL.

El tipo de expresión a ?? b depende de qué conversiones implícitas están disponibles en los operandos. En orden de preferencia, el tipo de a ?? b es A0, A o B, donde A es el tipo de a (siempre que a tenga un tipo), B es el tipo de b (siempre que b tenga un tipo), y A0 es un tipo subyacente de A si A es un tipo que acepta valores NULL, o en caso contrario, A. En concreto, a ?? b se procesa de la siguiente manera:

* Si A existe y no es un tipo que acepta valores NULL o un tipo de referencia, se genera un error en tiempo de compilación.
* Si b es una expresión dinámica, el tipo de resultado es dynamic. En tiempo de ejecución, a se evalúa en primer lugar. Si a no es NULL, se convierte en dinámico y esto se convierte en el resultado. De lo contrario, b se evalúa y esto se convierte en el resultado.
* De lo contrario, si A existe y es un tipo que acepta valores NULL y existe una conversión implícita de b a A0, el tipo de resultado es A0. En tiempo de ejecución, a se evalúa en primer lugar. Si a no es NULL, se desajusta en el tipo A0 y este se convierte en el resultado. De lo contrario, b se evalúa y se convierte al tipo A0 y este se convierte en el resultado.
* De lo contrario, si existe A y una conversión implícita de b a A, el tipo de resultado es A. En tiempo de ejecución, a se evalúa en primer lugar. Si a no es NULL, a se convierte en el resultado. De lo contrario, b se evalúa y se convierte al tipo A y este se convierte en el resultado.
* De lo contrario, si A existe y es un tipo que acepta valores NULL, b tiene el tipo B y existe una conversión implícita de A0 a B, el tipo de resultado es B. En tiempo de ejecución, a se evalúa en primer lugar. Si a no es NULL, a se desajusta en el tipo A0 y se convierte en el tipo B, que pasa a ser el resultado. De lo contrario, b se evalúa y se convierte en el resultado.
* De lo contrario, si b tiene un tipo B y existe una conversión implícita de a a B, el tipo de resultado es B. En tiempo de ejecución, a se evalúa en primer lugar. Si a no es NULL, a se convierte en el tipo B y este pasa a ser el resultado. De lo contrario, b se evalúa y se convierte en el resultado.
* De lo contrario, a y b son incompatibles y se genera un error en tiempo de compilación.

## Operador condicional

El operador ?: se denomina operador condicional. A veces también se le denomina operador ternario.

conditional-expression:  
null-coalescing-expression  
null-coalescing-expression ? expression : expression

Una expresión condicional con la forma b ? x : y evalúa en primer lugar la condición b. Entonces, si el valor de b es true, x se evalúa y pasa a ser el resultado de la operación. De lo contrario, se evalúa y, que se convierte en el resultado de la operación. Una expresión condicional nunca evalúa x e y.

El operador condicional es asociativo por la derecha, lo que significa que las operaciones se agrupan de derecha a izquierda. Por ejemplo, una expresión con la forma a ? b : c ? d : e se evalúa como a ? b : (c ? d : e).

El primer operando del operador ?: debe ser una expresión que pueda convertirse implícitamente a bool o una expresión de un tipo que implemente operator true. Si no se cumple ninguna de estas dos condiciones, se producirá un error de compilación.

El segundo y tercer operandos, x e y, del operador ?: controlan el tipo de la expresión condicional.

* Si x tiene el tipo X e y tiene el tipo Y entonces
* Si existe una conversión implícita (§6.1) de X a Y, pero no de Y a X, entonces Y es el tipo de la expresión condicional.
* Si existe una conversión implícita (§6.1) de Y a X, pero no de X a Y, entonces X es el tipo de la expresión condicional.
* En caso contrario, no puede determinarse ninguna expresión y se produce un error en tiempo de compilación.
* Si solo x o y tiene un tipo, y tanto x como y se pueden convertir implícitamente en ese tipo, ese tipo es el de la expresión condicional.
* En caso contrario, no puede determinarse una expresión y se produce un error de compilación.

El procesamiento en tiempo de ejecución de una expresión condicional con la forma b ? x : y consta de los siguientes pasos:

* En primer lugar, se evalúa b y se determina el valor bool de b:
* Si existe una conversión implícita del tipo de b a bool, se ejecuta dicha conversión para generar un valor bool.
* De lo contrario, se invoca el operator true definido por el tipo de b para generar un valor bool.
* Si el valor bool producido por el paso anterior es true, entonces se evalúa x y se convierte al tipo de la expresión condicional, que pasa a ser el resultado de la expresión condicional.
* O bien, y se evalúa y se convierte al tipo de la expresión condicional, y pasa a ser el resultado de dicha de expresión.

## Expresiones de funciones anónimas

Una función anónima es una expresión que representa una definición de método “en línea”. Una función anónima no tiene un valor o tipo de por sí, pero se puede convertir en un tipo de árbol de expresiones o un tipo delegado compatible. La evaluación de una conversión de función anónima depende del tipo de destino de la conversión. Si es un tipo delegado, la conversión se evalúa para un valor delegado que hace referencia al método en el que se define la función anónima. Si es un tipo de árbol de expresiones, la conversión se evalúa para un árbol de expresiones que representa la estructura del método como una estructura de objetos.

Por motivos históricos, hay dos tipos sintácticos de funciones anónimas, en concreto, las expresiones lambda (lambda-expression) y las expresiones de método anónimo (anonymous-method-expression). Para casi todos los fines, las expresiones lambda son más concisas y expresivas que las expresiones de método anónimo, que permanecen en el lenguaje por compatibilidad con versiones anteriores.

lambda-expression:  
async*opt* anonymous-function-signature => anonymous-function-body

anonymous-method-expression:  
async*opt* delegate explicit-anonymous-function-signatureopt block

anonymous-function-signature:  
explicit-anonymous-function-signature   
implicit-anonymous-function-signature

explicit-anonymous-function-signature:  
( explicit-anonymous-function-parameter-list*opt* )

explicit-anonymous-function-parameter-list:  
explicit-anonymous-function-parameter  
explicit-anonymous-function-parameter-list , explicit-anonymous-function-parameter

explicit-anonymous-function-parameter:  
anonymous-function-parameter-modifieropt type identifier

anonymous-function-parameter-modifier:   
ref  
out

implicit-anonymous-function-signature:  
( implicit-anonymous-function-parameter-list*opt* )  
implicit-anonymous-function-parameter

implicit-anonymous-function-parameter-list:  
implicit-anonymous-function-parameter  
implicit-anonymous-function-parameter-list , implicit-anonymous-function-parameter

implicit-anonymous-function-parameter:  
identifier

anonymous-function-body:  
expression  
block

El operador => tiene la misma prioridad que la asignación (=) y es asociativo por la derecha.

Una función anónima con el modificador async es una función asincrónica y sigue las reglas descritas en §10.14.

Los parámetros de una función anónima con la estructura de una expresión lambda (lambda-expression) pueden tener asignación de tipo implícita o explícita. En una lista de parámetros con asignación de tipo explícita, el tipo de cada parámetro se indica de manera explícita. En una lista de parámetros con asignación de tipo implícita, los tipos de los parámetros se infieren del contexto en el que tiene lugar la función anónima. Concretamente, cuando la función anónima se convierte a un tipo delegado compatible o un tipo de árbol de expresiones, ese tipo proporciona los tipos de parámetros (§6.5).

En una función anónima con un solo parámetro con asignación de tipo implícita, los paréntesis de la lista de parámetros se pueden omitir. En otras palabras, una función anónima con la forma

( param ) => expr

se puede abreviar a

param => expr

La lista de parámetros de una función anónima con la forma de una expresión de método anónimo (anonymous-method-expression) es opcional. Los parámetros deben tener asignación de tipo explícita, en caso de tener tipo. Si no, la función anónima es convertible a un delegado con cualquier lista de parámetros que no contenga parámetros out.

Se puede acceder a un cuerpo de bloque de una función anónima (§8.1) a menos que la función anónima tenga lugar dentro de una instrucción inaccesible.

A continuación, se muestran algunos ejemplos de funciones anónimas:

x => x + 1 // Implicitly typed, expression body

x => { return x + 1; } // Implicitly typed, statement body

(int x) => x + 1 // Explicitly typed, expression body

(int x) => { return x + 1; } // Explicitly typed, statement body

(x, y) => x \* y // Multiple parameters

() => Console.WriteLine() // No parameters

async (t1,t2) => await t1 + await t2 // Async

delegate (int x) { return x + 1; } // Anonymous method expression

delegate { return 1 + 1; } // Parameter list omitted

El comportamiento de las expresiones lambda (lambda-expression) y las expresiones de métodos anónimos (anonymous-method-expression) es igual excepto en los siguientes puntos:

* Las expresiones de métodos anónimos (anonymous-method-expression) permiten que una lista de parámetros se omita por completo, lo que tiene como resultado la convertibilidad a tipos delegados de cualquier lista de parámetros de valor.
* Las expresiones lambda (lambda-expression) permiten que los tipos de parámetros se omitan y sean inferidos mientras que las expresiones de métodos anónimos (anonymous-method-expression) requieren que los tipos de parámetros se indiquen de manera explícita.
* El cuerpo de una expresión lambda (lambda-expression) puede ser una expresión o un bloque de instrucciones mientras que el cuerpo de una expresión de métodos anónimos (anonymous-method-expression) debe ser un bloque de instrucciones.
* Solo las expresiones lambda se pueden convertir a tipos de árbol de expresión (§4.6).

### Signaturas de función anónima

La signatura opcional de función anónima (anonymous-function-signature) de una función anónima define los nombres y, opcionalmente, los tipos de los parámetros formales para la función anónima. El ámbito de los parámetros de la función anónima es el cuerpo de la función anónima (anonymous-function-body). (§3.7) Junto con la lista de parámetros (si la hay) el cuerpo de método anónimo constituye un espacio de declaración (§3.3). Por tanto, si el nombre de un parámetro de la función anónima coincide con el nombre de una variable local, una constante o un parámetro local cuyo ámbito incluya la expresión de métodos anónimos (anonymous-method-expression) o la expresión lambda (lambda-expression) se produce un error en tiempo de compilación.

Si una función anónima tiene una signatura de función anónima explícita (explicit-anonymous-function-signature), el conjunto de tipos delegados compatibles y los tipos de árbol de expresiones se limitan a aquellos con los mismos tipos de parámetros y modificadores en el mismo orden. Al contrario de lo que ocurre con las conversiones de grupo de métodos (§6.6), no se admite la contravarianza de tipos de parámetros de función anónima. Si una función anónima no tiene una signatura de función anónima (anonymous-function-signature), el conjunto de tipos delegados compatibles y tipos de árbol de expresiones se limita a aquellos que no tienen parámetros out.

Tenga en cuenta que una signatura de función anónima (anonymous-function-signature) no puede incluir atributos ni una matriz de parámetros. No obstante, una signatura de función anónima (anonymous-function-signature) puede ser compatible con un tipo delegado cuya lista de parámetros contenga una matriz de parámetros.

Tenga en cuenta que una conversión a un tipo de árbol de expresiones, incluso si es compatible, puede no producirse durante la compilación (§4.6).

### Cuerpos de función anónima

El cuerpo (expression o block) de una función anónima está sujeto a las siguientes reglas:

* Si la función anónima incluye una firma, los parámetros especificados en la firma están disponibles en el cuerpo. Si la función anónima no tiene signatura, se puede convertir en un tipo delegado o un tipo de expresión con parámetros (§6.5), pero no se puede obtener acceso a ellos en el cuerpo.
* Excepto por los parámetros ref u out especificados en la signatura (si los hubiera) de la función anónima envolvente más cercana, si el cuerpo obtiene acceso a un parámetro ref u out se generará un error en tiempo de compilación.
* Cuando el tipo de this es un tipo struct y el cuerpo obtiene acceso a this, se generará un error en tiempo de compilación. Esta afirmación es cierta independientemente de si se trata de un acceso explícito (como en this.x) o implícito (como en x, donde x es un miembro de instancia de struct). Esta regla simplemente prohíbe dicho acceso y no afecta si la búsqueda de miembros da como resultado un miembro de struct.
* El cuerpo tiene acceso a las variables externas (§7.15.5) de la función anónima. El acceso de una variable externa hará referencia a la instancia de la variable activa en el momento de la evaluación de la expresión lambda (lambda-expression) o la expresión de método anónimo (anonymous-method-expression) (§7.15.6).
* Si el cuerpo contiene una instrucción goto, break o continue cuyo destino se encuentra fuera o dentro del cuerpo de una función anónima contenida, se generará un error en tiempo de compilación.
* Una instrucción return del cuerpo devuelve el control desde una invocación de la función anónima envolvente más cercana, no desde el miembro de función envolvente. Una expresión especificada en una instrucción return debe poder convertirse implícitamente en el tipo devuelto del tipo de delegado o un tipo de árbol de expresiones al que se convierte la expresión lambda (lambda-expression) o la expresión de método anónimo (anonymous-method-expression) envolvente más cercana (§6.5).

No se especifica explícitamente si se puede ejecutar el bloque de una función anónima de otra manera que no sea mediante la evaluación y la invocación de la expresión lambda (lambda-expression) o la expresión de método anónimo (anonymous-method-expression). En concreto, el compilador puede implementar una función anónima si sintetiza uno o más métodos o tipos con nombre. Los nombres de dichos elementos sintetizados deben tener la forma reservada para uso del compilador.

### Resolución de sobrecargas

Las funciones anónimas de una lista de argumentos participan en la resolución de sobrecargas y en la inferencia de tipos. Para conocer las reglas exactas vea las secciones §7.5.2 y §7.5.3.

El efecto de las funciones anónimas en la resolución de las sobrecargas se ilustra en el siguiente ejemplo:

class ItemList<T>: List<T>  
{  
 public int Sum(Func<T,int> selector) {  
 int sum = 0;  
 foreach (T item in this) sum += selector(item);  
 return sum;  
 }

public double Sum(Func<T,double> selector) {  
 double sum = 0;  
 foreach (T item in this) sum += selector(item);  
 return sum;  
 }  
}

La clase ItemList<T> tiene dos métodos Sum. Cada uno toma un argumento selector, que extrae el valor que se va a sumar de un elemento de la lista. El valor extraído puede ser int o double, y la suma resultante es igualmente int o double.

Los métodos Sum podrían por ejemplo utilizarse para calcular sumas de una lista de líneas de datos en un pedido.

class Detail  
{  
 public int UnitCount;  
 public double UnitPrice;  
 ...  
}

void ComputeSums() {  
 ItemList<Detail> orderDetails = GetOrderDetails(...);  
 int totalUnits = orderDetails.Sum(d => d.UnitCount);  
 double orderTotal = orderDetails.Sum(d => d.UnitPrice \* d.UnitCount);  
 ...  
}

En la primera llamada de orderDetails.Sum, ambos métodos Sum son aplicables porque la función anónima d => d.UnitCount es compatible con Func<Detail,int> y Func<Detail,double>. Sin embargo, la resolución de sobrecargas elige el primer método Sum porque la conversión a Func<Detail,int> es mejor que la conversión a Func<Detail,double>.

En la segunda llamada de orderDetails.Sum, solo es aplicable el segundo método Sum porque la función anónima d => d.UnitPrice \* d.UnitCount produce un valor de tipo double. Por tanto, la resolución de sobrecargas elige el segundo método Sum para esa invocación.

### Funciones anónimas y enlace dinámico

Una función anónima no puede ser receptor, argumento ni operando de una operación enlazada dinámicamente.

### Variables externas

Cualquier variable local, parámetro de valor o matriz de parámetros en cuyo ámbito se incluya la expresión lambda (lambda-expression) o una expresión de método anónimo (anonymous-method-expression) se denomina variable externa de la expresión de función anónima. En un miembro de función de instancia de una clase, el valor this se considera un parámetro de valor y es una variable externa de cualquier expresión de función anónima contenida dentro del miembro de función.

#### Variables externas capturadas

Cuando una función anónima hace referencia a una variable externa, se dice que la función anónima capturó la variable externa. De manera general, la duración de una variable local está limitada por la ejecución del bloque o de una instrucción a la que está asociada (§5.1.7). Sin embargo, la duración de una variable externa capturada se amplía al menos hasta que el delegado o el árbol de expresiones creado a partir de la función anónima pueda convertirse en una recolección de elementos no utilizados.

En el ejemplo

using System;

delegate int D();

class Test  
{  
 static D F() {  
 int x = 0;  
 D result = () => ++x;  
 return result;  
 }

static void Main() {  
 D d = F();  
 Console.WriteLine(d());  
 Console.WriteLine(d());  
 Console.WriteLine(d());  
 }  
}

la función anónima captura la variable local x y el período de duración de x se amplía hasta que el delegado devuelto desde F pasa a formar parte de la recolección de elementos no utilizados (que no ocurre hasta el final del programa). Puesto que todas las invocaciones de la función anónima operan en la misma instancia que x, el resultado de este ejemplo es el siguiente:

1  
2  
3

Cuando una función anónima captura una variable local o un parámetro de valor, la variable local o el parámetro ya no se considera una variable de tipo fijo (§18.3), sino una variable móvil. Por lo tanto, todo código unsafe que adopte la dirección de una variable externa capturada debe primero utilizar la instrucción fixed para fijar la variable.

Tenga en cuenta que, a diferencia de una variable no capturada, una variable local capturada se puede exponer simultáneamente a varios subprocesos de ejecución.

#### Creación de instancias de variables locales

Se considera que se crearon instancias de una variable local cuando la ejecución entra dentro del ámbito de la variable. Por ejemplo, cuando se invoca el siguiente método, se crean instancias de la variable local x, que se inicializa tres veces, una para cada iteración del bucle.

static void F() {  
 for (int i = 0; i < 3; i++) {  
 int x = i \* 2 + 1;  
 ...  
 }  
}

Sin embargo, si la declaración de x se desplaza fuera del bucle, solo se creará una única instancia de x:

static void F() {  
 int x;  
 for (int i = 0; i < 3; i++) {  
 x = i \* 2 + 1;  
 ...  
 }  
}

Cuando no existe captura, no se puede saber con qué frecuencia exacta se crean instancias de una variable local. Puesto que las duraciones de creación de instancias son independientes es posible que cada vez que se crean instancias se utilice la misma ubicación de almacenamiento. Sin embargo, cuando una función anónima captura una variable local, los efectos de la creación de instancias son obvios.

El ejemplo

using System;

delegate void D();

class Test  
{  
 static D[] F() {  
 D[] result = new D[3];  
 for (int i = 0; i < 3; i++) {  
 int x = i \* 2 + 1;  
 result[i] = () => { Console.WriteLine(x); };  
 }  
 return result;  
 }

static void Main() {  
 foreach (D d in F()) d();  
 }  
}

genera el siguiente resultado:

1  
3  
5

Sin embargo, cuando la declaración de x se traslada fuera del bucle:

static D[] F() {  
 D[] result = new D[3];  
 int x;  
 for (int i = 0; i < 3; i++) {  
 x = i \* 2 + 1;  
 result[i] = () => { Console.WriteLine(x); };  
 }  
 return result;  
}

el resultado es:

5  
5  
5

Si un bucle for declara una variable de iteración, la propia variable se considera declarada fuera del bucle. Por tanto, si el ejemplo cambia para capturar la variable de iteración en sí:

static D[] F() {  
 D[] result = new D[3];  
 for (int i = 0; i < 3; i++) {  
 result[i] = () => { Console.WriteLine(i); };  
 }  
 return result;  
}

sólo se captura una instancia de la variable de iteración, que produce el resultado:

3  
3  
3

La función anónima puede compartir algunas de las variables capturadas y tener instancias separadas de otras. Por ejemplo, si F se cambia a

static D[] F() {  
 D[] result = new D[3];  
 int x = 0;  
 for (int i = 0; i < 3; i++) {  
 int y = 0;  
 result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };  
 }  
 return result;  
}

los tres delegados capturan la misma instancia de x y diferentes instancias de y, y el resultado es:

1 1  
2 1  
3 1

Distintas funciones anónimas pueden capturar la misma instancia de una variable externa. En el siguiente ejemplo:

using System;

delegate void Setter(int value);

delegate int Getter();

class Test  
{  
 static void Main() {  
 int x = 0;  
 Setter s = (int value) => { x = value; };  
 Getter g = () => { return x; };  
 s(5);  
 Console.WriteLine(g());  
 s(10);  
 Console.WriteLine(g());  
 }  
}

las dos funciones anónimas capturan la misma instancia de la variable local x y, por lo tanto, se pueden “comunicar” a través de dicha variable. El resultado del ejemplo es el siguiente:

5  
10

### Expresiones de evaluación de funciones anónimas

Una función anónima F siempre se debe convertir a un tipo delegado D o un tipo de árbol de expresiones E, ya sea directamente o mediante la ejecución de una expresión de creación de delegado new D(F). Esta conversión determina el resultado de la función anónima, como se describe en la sección §6.5.

## Expresiones de consulta

Las expresiones de consulta proporcionan una sintaxis integrada de lenguajes para las consultas que son similares a los lenguajes de consulta jerárquicos y relacionales, como SQL y XQuery.

query-expression:  
from-clause query-body

from-clause:  
from typeopt identifier in expression

query-body:  
query-body-clausesopt select-or-group-clause query-continuationopt

query-body-clauses:  
query-body-clause  
query-body-clauses query-body-clause

query-body-clause:  
from-clause  
let-clause  
where-clause  
join-clause  
join-into-clause  
orderby-clause

let-clause:  
let identifier = expression

where-clause:  
where boolean-expression

join-clause:  
join typeopt identifier in expression on expression equals expression

join-into-clause:  
join typeopt identifier in expression on expression equals expression into identifier

orderby-clause:  
orderby orderings

orderings:  
ordering  
orderings , ordering

ordering:  
expression ordering-directionopt

ordering-direction:  
ascending  
descending

select-or-group-clause:  
select-clause  
group-clause

select-clause:  
select expression

group-clause:  
group expression by expression

query-continuation:  
into identifier query-body

Una expresión de consulta empieza con una cláusula from y termina con una cláusula select o group. La cláusula from inicial puede ir seguida de cero o más cláusulas from, let, where, join u orderby. Cada cláusula from es un generador que introduce una variable de intervalo en los elementos de una secuencia. Cada cláusula let introduce una variable de intervalo que representa un valor calculado mediante variables de intervalo anteriores. Cada cláusula where es un filtro que excluye elementos del resultado. Cada cláusula join compara las claves especificadas de la secuencia de origen con las claves de otra secuencia, dando como resultado pares coincidentes. Cada cláusula orderby reordena los elementos según criterios especificados. La cláusula select o group final especifica la forma del resultado en términos de variables de intervalo. Por último, una cláusula into se puede usar para “juntar” consultas tratando los resultados de una consulta como un generador en una consulta subsiguiente.

### Ambigüedad en expresiones de consulta

Las expresiones de consulta contienen varias “palabras clave contextuales”, es decir, identificadores que tienen un significado especial en un contexto dado. Específicamente son from, where, join, on, equals, into, let, orderby, ascending, descending, select, group y by. Para evitar ambigüedades en las expresiones de consultas originadas por la mezcla en el uso de estos identificadores como palabras clave o como nombres comunes, los identificadores se consideran palabras clave cuando tienen lugar dentro de una expresión de consulta.

Con este fin, una expresión de consulta siempre empezará con “from identifier” (“identificador from”) seguido de un token especial excepto “;”, “=” o “,”.

Para poder utilizar estas palabras como identificadores dentro de una expresión de consulta, pueden ir prefijadas con “@” (§2.4.2).

### Traducción de expresiones de consulta

El lenguaje C# no especifica la semántica de ejecución en expresiones de consulta. En lugar de eso, las expresiones de consulta se traducen en invocaciones de métodos que se adhieren al patrón de expresiones de consulta (§7.16.3). Concretamente, las expresiones de consulta se traducen en invocaciones de métodos denominados Where, Select, SelectMany, Join, GroupJoin, OrderBy, OrderByDescending, ThenBy, ThenByDescending, GroupBy y Cast. Se espera que estos métodos tengan tipos de resultados y signaturas particulares, tal como se describe en la sección §7.16.3. Estos métodos pueden ser métodos de instancia del objeto sobre el que se realiza la consulta o métodos de extensión externos al objeto, e implementan la ejecución real de la consulta.

La traducción de expresiones de consulta a invocaciones de método es una asignación sintáctica que tiene lugar antes de que se produzca cualquier enlace de tipos o resolución de sobrecargas. Se garantiza que la traducción es correcta desde el punto de vista sintáctico, pero no se garantiza que produzca código de C# correcto desde el punto de vista semántico. Después de la traducción de las expresiones de consulta, las invocaciones de método resultantes se procesan como invocaciones de método comunes, y este procesamiento puede a su vez descubrir errores, por ejemplo si los métodos no existen, si los argumentos tienen tipos erróneos, o si los métodos son genéricos y se produce un error en la interfaz de tipos.

Una expresión de consulta se procesa aplicando repetidamente las siguientes traducciones hasta que no hay más reducciones posibles. Las traducciones se enumeran en orden de aplicación: cada sección asume que las traducciones de las secciones precedentes se han realizado de manera exhaustiva, y una vez agotada, una sección no volverá a ser visitada en el procesamiento de la misma expresión de consulta.

La asignación a variables de intervalo no está permitida en expresiones de consulta. Sin embargo, se permite que una implementación de C# no siempre exija esta restricción porque algunas veces no es posible con el esquema de conversión sintáctico aquí presentado.

Determinadas traducciones insertan variables de intervalo con identificadores transparentes que se denotan mediante \*. Las propiedades especiales de los identificadores transparentes se discuten con más detalle en la sección §7.16.2.7.

#### Cláusulas Select y Groupby con continuaciones

Una expresión de consulta con una continuación

from … into x …

se traduce en

from x in ( from … ) …

Las traducciones en las siguientes secciones asumen que las consultas no tienen continuaciones into.

El ejemplo

from c in customers  
group c by c.Country into g  
select new { Country = g.Key, CustCount = g.Count() }

se traduce en

from g in  
 from c in customers  
 group c by c.Country  
select new { Country = g.Key, CustCount = g.Count() }

cuya traducción final es

customers.  
GroupBy(c => c.Country).  
Select(g => new { Country = g.Key, CustCount = g.Count() })

#### Tipos de variable de intervalo explícitos

Una cláusula from que especifica explícitamente un tipo de variable de intervalo

from T x in e

se traduce en

from x in ( e ) . Cast < T > ( )

Una cláusula join que especifica explícitamente un tipo de variable de intervalo

join T x in e on k1 equals k2

se traduce en

join x in ( e ) . Cast < T > ( ) on k1 equals k2

Las traducciones en las siguientes secciones asumen que las consultas no tienen tipos de variables de intervalo explícitos.

El ejemplo

from Customer c in customers  
where c.City == "London"  
select c

se traduce en

from c in customers.Cast<Customer>()  
where c.City == "London"  
select c

cuya traducción final es

customers.  
Cast<Customer>().  
Where(c => c.City == "London")

Los tipos de variable de intervalo explícitos son útiles para realizar consultas en colecciones que implementan la interfaz IEnumerable no genérica, pero no la interfaz IEnumerable<T> genérica. En el ejemplo anterior, este sería el caso si customers fuera de tipo ArrayList.

#### Expresiones de consulta degeneradas

Una expresión de consulta con la estructura

from x in e select x

se traduce en

( e ) . Select ( x => x )

El ejemplo

from c in customers  
select c

se traduce en

customers.Select(c => c)

Una expresión de consulta degenerada es aquella que selecciona de manera trivial los elementos del origen. En una fase posterior de la traducción se quitan las consultas degeneradas introducidas por otros pasos de traducción reemplazándolas por su origen. Es importante, no obstante, asegurar que el resultado de una expresión de consulta nunca sea el propio objeto de origen, ya que esto revelaría el tipo y la identidad del origen al cliente de la consulta. De este modo, este paso protege las consultas degeneradas escritas directamente en el código de origen llamando a Select en el origen. A partir de ese momento corresponde a los implementadores de Select y los demás operadores de consulta garantizar que estos métodos nunca devuelvan el objeto de origen en sí.

#### Cláusulas from, let, where, join y orderby

Una expresión de consulta con una segunda cláusula from seguida de una cláusula select

from x1 in e1  
from x2 in e2  
select v

se traduce en

( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )

Una expresión de consulta con una segunda cláusula from seguida de una cláusula que no es select:

from x1 in e1  
from x2 in e2  
…

se traduce en

from \* in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )  
…

Una expresión de consulta con una cláusula let

from x in e  
let y = f  
…

se traduce en

from \* in ( e ) . Select ( x => new { x , y = f } )  
…

Una expresión de consulta con una cláusula where

from x in e  
where f  
…

se traduce en

from x in ( e ) . Where ( x => f )  
…

Una expresión de consulta con una cláusula join sin una cláusula into, y seguida de una cláusula select

from x1 in e1  
join x2 in e2 on k1 equals k2  
select v

se traduce en

( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )

Una expresión de consulta con una cláusula join sin una cláusula into, y seguida de una cláusula que no es select

from x1 in e1  
join x2 in e2 on k1 equals k2   
…

se traduce en

from \* in ( e1 ) . Join(  
 e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })  
…

Una expresión de consulta con una cláusula join con una cláusula into, y seguida de una cláusula select

from x1 in e1  
join x2 in e2 on k1 equals k2 into g  
select v

se traduce en

( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )

Una expresión de consulta con una cláusula join con una cláusula into, y seguida de una cláusula que no es select

from x1 in e1  
join x2 in e2 on k1 equals k2 into g  
…

se traduce en

from \* in ( e1 ) . GroupJoin(  
 e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 , g })  
…

Una expresión de consulta con una cláusula orderby

from x in e  
orderby k1 , k2 , … , kn  
…

se traduce en

from x in ( e ) .   
OrderBy ( x => k1 ) .   
ThenBy ( x => k2 ) .  
 … .   
ThenBy ( x => kn )  
…

Si una cláusula de ordenación especifica un indicador de dirección descending, se produce en su lugar una invocación de OrderByDescending o ThenByDescending.

Las siguientes traducciones asumen que no hay ninguna cláusula let, where, join u orderby, y no más de una cláusula from inicial en cada expresión de consulta.

El ejemplo

from c in customers  
from o in c.Orders  
select new { c.Name, o.OrderID, o.Total }

se traduce en

customers.  
SelectMany(c => c.Orders,  
 (c,o) => new { c.Name, o.OrderID, o.Total }  
)

El ejemplo

from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }

se traduce en

from \* in customers.  
 SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }

cuya traducción final es

customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(x => x.o.Total).  
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })

donde x es un identificador generado por el compilador que de otro modo es invisible e inaccesible.

El ejemplo

from o in orders  
let t = o.Details.Sum(d => d.UnitPrice \* d.Quantity)  
where t >= 1000  
select new { o.OrderID, Total = t }

se traduce en

from \* in orders.  
 Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice \* d.Quantity) })  
where t >= 1000   
select new { o.OrderID, Total = t }

cuya traducción final es

orders.  
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice \* d.Quantity) }).  
Where(x => x.t >= 1000).  
Select(x => new { x.o.OrderID, Total = x.t })

donde x es un identificador generado por el compilador que de otro modo es invisible e inaccesible.

El ejemplo

from c in customers  
join o in orders on c.CustomerID equals o.CustomerID  
select new { c.Name, o.OrderDate, o.Total }

se traduce en

customers.Join(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, o) => new { c.Name, o.OrderDate, o.Total })

El ejemplo

from c in customers  
join o in orders on c.CustomerID equals o.CustomerID into co  
let n = co.Count()  
where n >= 10  
select new { c.Name, OrderCount = n }

se traduce en

from \* in customers.  
 GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, co) => new { c, co })  
let n = co.Count()  
where n >= 10   
select new { c.Name, OrderCount = n }

cuya traducción final es

customers.  
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, co) => new { c, co }).  
Select(x => new { x, n = x.co.Count() }).  
Where(y => y.n >= 10).  
Select(y => new { y.x.c.Name, OrderCount = y.n)

donde x e y son identificadores generados por el compilador que de otro modo son invisibles e inaccesibles.

El ejemplo

from o in orders  
orderby o.Customer.Name, o.Total descending  
select o

tiene la traducción final

orders.  
OrderBy(o => o.Customer.Name).  
ThenByDescending(o => o.Total)

#### Cláusulas Select

Una expresión de consulta con la estructura

from x in e select v

se traduce en

( e ) . Select ( x => v )

excepto cuando v es el identificador x, la traducción es simplemente

( e )

Por ejemplo:

from c in customers.Where(c => c.City == “London”)  
select c

se traduce simplemente en

customers.Where(c => c.City == “London”)

#### Cláusulas Groupby

Una expresión de consulta con la estructura

from x in e group v by k

se traduce en

( e ) . GroupBy ( x => k , x => v )

excepto cuando v es el identificador x, la traducción es

( e ) . GroupBy ( x => k )

El ejemplo

from c in customers  
group c.Name by c.Country

se traduce en

customers.  
GroupBy(c => c.Country, c => c.Name)

#### Identificadores transparentes

Determinadas traducciones insertan variables de intervalo con identificadores transparentes que se denotan mediante \*. Los identificadores transparentes no son una característica propia del lenguaje; existen sólo como un paso intermedio en el proceso de traducción de expresiones de consulta.

Cuando una traducción de consulta inserta un identificador transparente, los siguientes pasos de traducción propagan el identificador transparente en funciones anónimas e inicializadores de objetos anónimos. En estos contextos, los identificadores transparentes tienen el siguiente comportamiento:

* Cuando un identificador transparente tiene lugar como parámetro en una función anónima, los miembros del tipo anónimo asociado están automáticamente en el ámbito del cuerpo de la función anónima.
* Cuando un miembro con un identificador transparente está en el ámbito, los miembros de ese miembro están en el ámbito también.
* Cuando un identificador transparente tiene lugar como un declarador de miembro en un inicializador de objeto anónimo, éste introduce un miembro con un identificador transparente.

En los pasos de traducción descritos antes, los identificadores transparentes siempre se introducen junto con tipos anónimos, con la intención de capturar varias variables de intervalo como miembros de un solo objeto. Se permite que una implementación de C# utilice un mecanismo diferente que los tipos anónimos para agrupar juntas varias variables de intervalo. En la siguiente traducción se asume el uso de los tipos anónimos, y se muestra cómo se pueden traducir los identificadores transparentes.

El ejemplo

from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.Total }

se traduce en

from \* in customers.  
 SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.Total }

que se traduce a su vez en

customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(\* => o.Total).  
Select(\* => new { c.Name, o.Total })

que, cuando se eliminan los identificadores transparentes, equivale a

customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(x => x.o.Total).  
Select(x => new { x.c.Name, x.o.Total })

donde x es un identificador generado por el compilador que de otro modo es invisible e inaccesible.

El ejemplo

from c in customers  
join o in orders on c.CustomerID equals o.CustomerID  
join d in details on o.OrderID equals d.OrderID  
join p in products on d.ProductID equals p.ProductID  
select new { c.Name, o.OrderDate, p.ProductName }

se traduce en

from \* in customers.  
 Join(orders, c => c.CustomerID, o => o.CustomerID,   
 (c, o) => new { c, o })  
join d in details on o.OrderID equals d.OrderID  
join p in products on d.ProductID equals p.ProductID  
select new { c.Name, o.OrderDate, p.ProductName }

que a su vez se reduce a

customers.  
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).  
Join(details, \* => o.OrderID, d => d.OrderID, (\*, d) => new { \*, d }).  
Join(products, \* => d.ProductID, p => p.ProductID, (\*, p) => new { \*, p }).  
Select(\* => new { c.Name, o.OrderDate, p.ProductName })

cuya traducción final es

customers.  
Join(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, o) => new { c, o }).  
Join(details, x => x.o.OrderID, d => d.OrderID,  
 (x, d) => new { x, d }).  
Join(products, y => y.d.ProductID, p => p.ProductID,  
 (y, p) => new { y, p }).  
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })

donde x, y y z son identificadores generados por el compilador que de otro modo son invisibles e inaccesibles.

### El patrón de expresiones de consulta

El patrón de expresiones de consulta establece un patrón de métodos que los tipos pueden implementar para poder admitir expresiones. Dado que las expresiones de consulta se traducen en invocaciones de método mediante una asignación sintáctica, los tipos tienen una flexibilidad considerable a la hora de implementar el patrón de expresiones de consulta. Por ejemplo, los métodos de un patrón se pueden implementar como métodos de instancia o como métodos de extensión, porque los dos tienen la misma sintaxis de invocación, y los métodos pueden solicitar delegados o árboles de expresiones, porque las funciones anónimas son convertibles a ambos.

La forma recomendada de un tipo genérico C<T> que admite el patrón de expresiones de consulta se muestra a continuación. El tipo genérico se usa para poder ilustrar la relación adecuada entre parámetro y tipos de resultado, pero es posible implementar el patrón también para tipos no genéricos.

delegate R Func<T1,R>(T1 arg1);

delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);

class C  
{  
 public C<T> Cast<T>();  
}

class C<T> : C  
{  
 public C<T> Where(Func<T,bool> predicate);

public C<U> Select<U>(Func<T,U> selector);

public C<V> SelectMany<U,V>(Func<T,C<U>> selector,  
 Func<T,U,V> resultSelector);

public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,  
 Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);

public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,  
 Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);

public O<T> OrderBy<K>(Func<T,K> keySelector);

public O<T> OrderByDescending<K>(Func<T,K> keySelector);

public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);

public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,  
 Func<T,E> elementSelector);  
}

class O<T> : C<T>  
{  
 public O<T> ThenBy<K>(Func<T,K> keySelector);

public O<T> ThenByDescending<K>(Func<T,K> keySelector);  
}

class G<K,T> : C<T>  
{  
 public K Key { get; }  
}

Los métodos anteriores utilizan tipos delegados genéricos Func<T1, R> y Func<T1, T2, R>, pero igualmente podrían haber usado otros tipos delegados o de árbol de expresiones con las mismas relaciones en parámetro y tipos de resultado.

Tenga en cuenta la relación recomendada entre C<T> y O<T> que asegura que los métodos ThenBy y ThenByDescending están disponibles solo en el resultado de OrderBy u OrderByDescending. También la forma recomendada del resultado de GroupBy (una secuencia de secuencias), donde cada secuencia interna tiene una propiedad Key adicional.

El espacio de nombres System.Linq proporciona una implementación del patrón de operador de consultas para cualquier tipo que implementa la interfaz System.Collections.Generic.IEnumerable<T>.

## Operadores de asignación

Los operadores de asignación se utilizan para asignar un valor nuevo a una variable, propiedad, evento o elemento de indizador.

assignment:  
unary-expression assignment-operator expression

assignment-operator:  
=  
+=  
-=  
\*=  
/=  
%=  
&=  
|=  
^=  
<<=  
right-shift-assignment

El operando izquierdo de una asignación debe ser una expresión clasificada como una variable, un acceso a propiedad, un acceso a indizador o un acceso a evento.

El operador = se denomina operador de asignación simple. Asigna el valor del operando derecho a la variable, propiedad o elemento de indizador dado por el operando izquierdo. El operando de la izquierda del operador de asignación simple puede no ser un acceso a evento (excepto en las condiciones descritas en §10.8.1). El operador de asignación simple se explica en §7.17.1.

Los operadores de asignación distintos del operador = se denominan operadores de asignación compuestos. Dichos operadores ejecutan la operación indicada en los dos operandos y después asignan el valor resultante a la variable, propiedad o elemento de indizador dado por el operando izquierdo. Los operadores de asignación compuesta se explican en §7.17.2.

Los operadores += y -= con una expresión de acceso a evento como operando izquierdo se denominan operadores de asignación de eventos. Ningún otro operador de asignación es válido con un acceso a evento como operando izquierdo. Los operadores de asignación de eventos se explican en §7.17.3.

Los operadores de asignación son asociativos por la derecha, lo que significa que las operaciones se agrupan de derecha a izquierda. Por ejemplo, una expresión con la forma a = b = c se evalúa como a = (b = c).

### Asignación simple

El operador = se denomina operador de asignación simple.

Si el operando izquierdo de una asignación simple tiene la forma E.P o E[Ei] donde E tiene el tipo dynamic en tiempo de compilación, la asignación se enlaza dinámicamente (§7.2.2). En este caso, el tipo en tiempo de compilación de la expresión de asignación es dynamic y la resolución descrita a continuación se produce en tiempo de ejecución usando el tipo en tiempo de ejecución de E.

En una asignación simple, el operando derecho debe ser una expresión que pueda convertirse implícitamente al tipo del operando izquierdo. La operación asigna el valor del operando derecho a la variable, propiedad o elemento de indizador dado por el operando izquierdo.

El resultado de una expresión de asignación simple es el valor asignado al operando izquierdo. El resultado tiene el mismo tipo que el operando izquierdo y siempre se clasifica como un valor.

Si el operando izquierdo es una propiedad o un acceso a indizador, debe tener un descriptor de acceso set. Si no es éste el caso, se produce un error en tiempo de enlace.

El procesamiento en tiempo de ejecución de una asignación simple de la forma x = y consta de los siguientes pasos:

* Si x se clasifica como una variable:
* x se evalúa para producir la variable.
* y se evalúa y, si es necesario, se convierte al tipo de x mediante una conversión implícita (§6.1).
* Si la variable dada por x es un elemento de matriz de un tipo de referencia (reference-type), se lleva a cabo una comprobación en tiempo de ejecución para garantizar que el valor calculado de y es compatible con la instancia de matriz a la cual pertenece x. La comprobación es satisfactoria si y es null o si existe una conversión de referencia implícita (§6.1.6) del tipo real de la instancia a que hace referencia y al tipo del elemento real de la instancia de matriz que contiene x. De lo contrario, se inicia una excepción System.ArrayTypeMismatchException.
* El valor resultante de la evaluación y conversión de y se almacena en la ubicación dada por la evaluación de x.
* Si x se clasifica como una propiedad o un acceso a indizador:
* Se evalúan la expresión de instancia (si x no es static) y la lista de argumentos (si x es un acceso a indizador) asociadas con x, y el resultado se utiliza en la posterior invocación del descriptor de acceso set.
* y se evalúa y, si es necesario, se convierte al tipo de x mediante una conversión implícita (§6.1).
* Se invoca el descriptor de acceso set de x con el valor calculado para y como su argumento value.

Las reglas de covarianza matricial (§12.5) permiten que un valor de un tipo de matriz A[] se trate como una referencia a una instancia de un tipo matricial B[], siempre que exista una conversión implícita de referencias de B a A. Debido a estas reglas, la asignación de un tipo de referencia (reference-type) a un elemento de matriz requiere una comprobación en tiempo de ejecución para garantizar que el valor asignado es compatible con la instancia de matriz. En el ejemplo

string[] sa = new string[10];  
object[] oa = sa;

oa[0] = null; // Ok  
oa[1] = "Hello"; // Ok  
oa[2] = new ArrayList(); // ArrayTypeMismatchException

la última asignación hace que se genere la excepción System.ArrayTypeMismatchException debido a que una instancia de ArrayList no se puede almacenar en un elemento de string[].

Cuando una propiedad o un indizador declarado en un tipo struct (struct-type) es el destino de una asignación, la expresión de instancia asociada al acceso a propiedad o a indizador debe estar clasificada como una variable. Si la expresión de instancia está clasificada como un valor, se produce un error de enlace. Conforme a la §7.6.4, la misma regla también es aplicable a los campos.

Dadas las declaraciones:

struct Point  
{  
 int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }

public int X {  
 get { return x; }  
 set { x = value; }  
 }

public int Y {  
 get { return y; }  
 set { y = value; }  
 }  
}

struct Rectangle  
{  
 Point a, b;

public Rectangle(Point a, Point b) {  
 this.a = a;  
 this.b = b;  
 }

public Point A {  
 get { return a; }  
 set { a = value; }  
 }

public Point B {  
 get { return b; }  
 set { b = value; }  
 }  
}

del ejemplo:

Point p = new Point();  
p.X = 100;  
p.Y = 100;  
Rectangle r = new Rectangle();  
r.A = new Point(10, 10);  
r.B = p;

las asignaciones a p.X, p.Y, r.A, y r.B se permiten porque p y r son variables. No obstante, en el ejemplo:

Rectangle r = new Rectangle();  
r.A.X = 10;  
r.A.Y = 10;  
r.B.X = 100;  
r.B.Y = 100;

las asignaciones no son válidas, puesto que r.A y r.B no son variables.

### Asignación compuesta

Si el operando izquierdo de una asignación compuesta tiene la forma E.P o E[Ei] donde E tiene el tipo dynamic en tiempo de compilación, la asignación se enlaza dinámicamente (§7.2.2). En este caso, el tipo en tiempo de compilación de la expresión de asignación es dynamic y la resolución descrita a continuación se produce en tiempo de ejecución usando el tipo en tiempo de ejecución de E.

Una operación de la forma x op= y se procesa mediante la aplicación de la resolución de sobrecargas de operadores binarios (§7.3.4) como si la operación se hubiera escrito (x) op y. Supongamos que R es el tipo devuelto del operador seleccionado y T el tipo de x. Entonces:

* Si existe una conversión implícita de una expresión de tipo R al tipo T, la operación se evalúa como x = (T)((x) op y), excepto en que x se evalúa una sola vez.
* O bien, si el operador seleccionado es un operador predefinido, si R es convertible explícitamente a T y si y es convertible implícitamente a T o el operador es de desplazamiento, entonces la operación se evalúa como x = (T)((x) op y), excepto en que x se evalúa una sola vez.
* O bien, la asignación compuesta no es válida y se produce un error en tiempo de enlace.

La expresión “se evalúa una sola vez” implica que, en la evaluación de x op y, el resultado de cualquier expresión que forme parte de x se guarda temporalmente y se reutiliza cuando se realiza la asignación a x. Por ejemplo, en la asignación A()[B()] += C(), donde A es un método que devuelve int[], y B y C son métodos que devuelven int, los métodos se invocan solamente una vez, en el orden A, B, C.

Si el operando izquierdo de una asignación compuesta es un acceso a propiedad o un acceso a indizador, la propiedad o el indizador debe tener un descriptor de acceso get y un descriptor de acceso set. Si no es éste el caso, se produce un error en tiempo de enlace.

La segunda regla mencionada permite que x op= y se evalúe como x = (T)((x) op y) en ciertos contextos. La regla existe para permitir utilizar los operadores predefinidos como operadores compuestos cuando el operando izquierdo es de tipo sbyte, byte, short, ushort o char. Aunque los dos argumentos sean de uno de estos tipos, los operadores predefinidos producen un resultado de tipo int, como se explica en §7.3.6.2. Por lo tanto, sin una conversión, no sería posible asignar el resultado al operando izquierdo.

El efecto intuitivo de la regla de los operadores predefinidos es, sencillamente, que x op= y está permitido si están permitidos (x) op y y x = y. En el ejemplo

byte b = 0;  
char ch = '\0';  
int i = 0;

b += 1; // Ok  
b += 1000; // Error, b = 1000 not permitted  
b += i; // Error, b = i not permitted  
b += (byte)i; // Ok

ch += 1; // Error, ch = 1 not permitted  
ch += (char)1; // Ok

el motivo intuitivo de cada error es que una asignación simple correspondiente también habría sido un error.

Esto también significa que las operaciones de asignación compuesta admiten operaciones de elevación. En el ejemplo

int? i = 0;  
i += 1; // Ok

se utiliza el operador de elevación +(int?,int?).

### Asignación de eventos

Si el operando izquierdo de un operador += o -= se clasifica como acceso a evento, la expresión se evalúa de la siguiente forma:

* Si existe una expresión de instancia del acceso a evento, se evalúa.
* Se evalúa el operando de la derecha del operador += o -= y, si fuera necesario, se convierte al tipo del operando de la izquierda mediante una conversión implícita (§6.1).
* Después de la evaluación y, si es necesario, también después de la conversión, se invoca un descriptor de acceso del evento, con la lista de argumentos formada por el operando derecho. Si el operador fuera +=, se invoca el descriptor de acceso add; si el operador fuera -=, se invoca el descriptor de acceso remove.

Una expresión de asignación de evento no produce ningún valor. Por tanto, una expresión de asignación de eventos sólo será válida en el contexto de una expresión de instrucción (statement-expression) (§8.6).

## Expresión

Una expresión (expression) es una expresión de no asignación (non-assignment-expression) o una asignación (assignment).

expression:   
non-assignment-expression  
assignment

non-assignment-expression:  
conditional-expression  
lambda-expression  
query-expression

## Expresiones constantes

Una expresión constante (constant-expression) es una expresión que se puede evaluar totalmente en tiempo de compilación.

constant-expression:  
expression

Una expresión constante debe ser el null literal o un valor con uno de los tipos siguientes: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, object, string o cualquier tipo de enumeración. Sólo las siguientes construcciones se permiten en expresiones de constantes:

* Literales (incluido null).
* Referencias a miembros const y tipos de clase y struct.
* Referencias a miembros de tipos de enumeración.
* Referencias a parámetros const o a variables locales
* Subexpresiones entre paréntesis, que son en sí mismas expresiones constantes.
* Expresiones de conversión, siempre que el tipo de destino sea uno de los antes indicados.
* Expresiones checked y unchecked
* Expresiones de valor predeterminadas
* Los operadores unarios predefinidos +, –, ! y ~.
* Los operadores binarios predefinidos +, –, \*, /, %, <<, >>, &, |, ^, &&, ||, ==, !=, <, >, <= y >=, siempre que todos los operandos sean de uno de los tipos indicados anteriormente.
* El operador condicional ?:.

Se permiten las siguientes conversiones en expresiones de constantes:

* Conversiones de identidad
* Conversiones numéricas
* Conversiones de enumeración
* Conversiones de expresión constante
* Conversiones de referencia implícita y explícita, siempre y cuando el origen de las conversiones sea una expresión constante que se evalúe como null.

Otras conversiones, incluidas las conversiones boxing y unboxing, y las conversiones de referencia implícita de valores distintos de null no se permiten en expresiones constantes. Por ejemplo:

class C {  
 const object i = 5; // error: boxing conversion not permitted  
 const object str = “hello”; // error: implicit reference conversion  
}

la inicialización de i es un error porque es necesaria una conversión boxing. La inicialización de str es un error porque es necesaria una conversión de referencia implícita desde un valor distinto de NULL.

Siempre que una expresión cumple los requisitos antes mencionados, la expresión se evalúa en tiempo de compilación. Esto ocurre así aunque la expresión sea una sub-expresión de una expresión mayor que contiene construcciones no constantes.

La evaluación en tiempo de compilación de expresiones constantes está regida por las mismas reglas que la evaluación en tiempo de ejecución de expresiones no constantes, excepto porque donde la evaluación en tiempo de ejecución iniciaría una excepción, la evaluación en tiempo de compilación causa un error de tiempo de compilación.

Salvo que una expresión constante se coloque explícitamente en un contexto unchecked, los desbordamientos que ocurran en las conversiones y operaciones aritméticas de tipo integral durante la evaluación en tiempo de compilación de la expresión siempre causarán errores de tiempo de compilación (§7.19).

Las expresiones constantes ocurren en los contextos que se enumeran a continuación. En estos contextos, se produce un error durante la compilación si una expresión no puede evaluarse totalmente en tiempo de compilación.

* Declaraciones de constantes (§10.4).
* Declaraciones de miembros de enumeración (§14.3).
* Argumentos predeterminados de las listas de parámetros formales (§10.6.1)
* Etiquetas case de una instrucción switch (§8.7.2).
* Instrucciones goto case (§8.9.3).
* Longitudes de dimensión en una expresión de creación de matriz (§7.6.10.4) que incluye un inicializador.
* Atributos (§17).

Una conversión implícita de expresión constante (§6.1.9) permite la conversión de una expresión constante de tipo int al tipo sbyte, byte, short, ushort, uint o ulong, siempre que el valor de la expresión constante quede dentro del intervalo del tipo de destino.

## Expresiones booleanas

Una expresión booleana (boolean-expression) es una expresión que devuelve un resultado de tipo bool; ya sea directamente o a través de la aplicación del operator true en ciertos contextos tal y como se especifica a continuación.

boolean-expression:  
expression

La expresión condicional que controla una instrucción if (if-statement) (§8.7.1), instrucción while (while-statement) (§8.8.1), instrucción do (do-statement) (§8.8.2) o instrucción for (for-statement) (§8.8.3) es una expresión booleana (boolean-expression). La expresión condicional de control del operador ?: (§7.14) sigue las mismas reglas que una expresión booleana (boolean-expression) pero, por motivos de prioridad de operadores, se clasifica como una expresión or condicional (conditional-or-expression).

Una expresión booleana (boolean-expression) E debe poder producir un valor del tipo bool, de la siguiente manera:

* Si E se puede convertir implícitamente a bool, la conversión implícita se aplica en tiempo de ejecución.
* De lo contrario, se utiliza la resolución de sobrecargas de operadores unarios (§7.3.3) con el fin de encontrar la mejor implementación única de operador true en E y se aplica dicha implementación en tiempo de ejecución.
* Si no se encuentra este operador se produce un error en tiempo de enlace.

El tipo struct DBBool de la sección §11.4.2 proporciona un ejemplo de un tipo que implementa operator true y operator false.

# Instrucciones

C# proporciona una gran variedad de instrucciones. La mayoría de ellas son conocidas por los programadores de C y C++.

statement:  
labeled-statement  
declaration-statement  
embedded-statement

embedded-statement:  
block  
empty-statement  
expression-statement  
selection-statement  
iteration-statement  
jump-statement  
try-statement  
checked-statement  
unchecked-statement  
lock-statement  
using-statement   
yield-statement

La instrucción incrustada (embedded-statement) sin terminación se utiliza en instrucciones que aparecen dentro de otras instrucciones. El uso de una instrucción incrustada (embedded-statement) en lugar de una instrucción (statement) permite que no sea necesario utilizar instrucciones de declaración y con etiqueta en dichos contextos. El ejemplo

void F(bool b) {  
 if (b)  
 int i = 44;  
}

da como resultado un error en tiempo de compilación, ya que una instrucción if requiere una instrucción incrustada (embedded-statement) en lugar de una instrucción (statement) para su rama if. Si se admitiera este código, la variable i se declararía pero no se utilizaría nunca. Observe, sin embargo, que si coloca la declaración de i en un bloque, el ejemplo es válido.

## Puntos finales y alcance

Toda instrucción tiene un punto final. De manera intuitiva, el punto final de una instrucción es la ubicación que sigue a la instrucción. Las reglas para la ejecución de instrucciones compuestas (instrucciones que contienen instrucciones incrustadas) especifican las acciones a tomar cuando el control llega al punto final de una instrucción incrustada. Por ejemplo, cuando el control llega al punto final de una instrucción dentro de un bloque, pasa a la siguiente instrucción del bloque.

Si una instrucción tiene posibilidades de ejecutarse, se dice que la instrucción es alcanzable. De manera inversa, si una instrucción no tiene ninguna posibilidad de ejecutarse, se dice que es una instrucción inalcanzable.

En el ejemplo

void F() {  
 Console.WriteLine("reachable");  
 goto Label;  
 Console.WriteLine("unreachable");  
 Label:  
 Console.WriteLine("reachable");  
}

la segunda llamada a Console.WriteLine es inalcanzable porque no hay posibilidad de que se ejecute la instrucción.

Si el compilador determina que existe alguna instrucción inalcanzable, emite una advertencia. Que una instrucción sea inalcanzable no es un error propiamente dicho.

Para determinar si una determinada instrucción o punto final es o no alcanzable, el compilador realiza un análisis del flujo del programa de acuerdo con las reglas de alcance definidas para cada instrucción. El análisis de flujo tiene en cuenta los valores de expresiones de constantes (§7.19) que controlan el comportamiento de las instrucciones, pero no considera los posibles valores de expresiones de variables. En otras palabras, en el análisis de flujo se considera que una expresión de variable de un determinado tipo puede tener cualquier valor posible de dicho tipo.

En el ejemplo

void F() {  
 const int i = 1;  
 if (i == 2) Console.WriteLine("unreachable");  
}

la expresión booleana de la instrucción if es una expresión constante porque los dos operandos del operador == son constantes. La expresión constante se evalúa en tiempo de compilación y, como devuelve el valor false, la llamada a Console.WriteLine se considera inalcanzable. Sin embargo, si i se convierte en una variable local

void F() {  
 int i = 1;  
 if (i == 2) Console.WriteLine("reachable");  
}

la invocación a Console.WriteLine se considera alcanzable, aunque en realidad no se ejecutará nunca.

El bloque (block) de un miembro de función siempre se considera alcanzable. Evaluando sucesivamente las reglas de alcance de cada instrucción de un bloque, puede determinarse el alcance de una determinada instrucción.

En el ejemplo

void F(int x) {  
 Console.WriteLine("start");  
 if (x < 0) Console.WriteLine("negative");  
}

la capacidad de alcanzar la segunda instrucción Console.WriteLine se determina de la forma siguiente:

* La primera instrucción de la expresión Console.WriteLine es alcanzable, ya que el bloque del método F también lo es.
* El punto final de la primera instrucción de expresión Console.WriteLine es alcanzable porque esa instrucción es alcanzable.
* La instrucción if es alcanzable porque el punto final de la primera instrucción Console.WriteLine también lo es.
* Por último, como la expresión booleana de la instrucción Console.WriteLine no tiene el valor constante if, la segunda instrucción false es alcanzable.

Existen dos situaciones en las que supone un error en tiempo de compilación que el punto final de una instrucción sea alcanzable:

* Como la instrucción switch no admite que de una sección “se pase” a la siguiente sección, supone un error en tiempo de compilación que el punto final de la lista de instrucciones de una sección de switch sea alcanzable. Este error suele producirse cuando falta una instrucción break.
* Es un error que sea alcanzable el punto final del bloque de un miembro de función que calcula un valor. Si este error se produce, suele indicar que falta una instrucción return.

## Bloques

Un bloque (block) permite escribir varias instrucciones en contextos donde se permite una única instrucción.

block:  
{ statement-listopt }

Un bloque (block) está formado por una lista de instrucciones (statement-list) opcional (§8.2.1), encerrada entre llaves. Si se omite la lista de instrucciones, se dice que el bloque es un bloque vacío.

Un bloque puede contener instrucciones de declaración (§8.5). El ámbito de una variable o constante local declarada en un bloque es el propio bloque.

Dentro de un bloque, el significado de un nombre utilizado en un contexto de expresión siempre debe ser el mismo (§7.6.2.1).

Un bloque se ejecuta de la siguiente forma:

* Si el bloque está vacío, el control se transfiere al punto final del bloque.
* Si no está vacío, el control se transfiere a la lista de instrucciones. Cuando el control alcanza el punto final de la lista de instrucciones, se transfiere al punto final del bloque.

La lista de instrucciones de un bloque es alcanzable si el propio bloque es alcanzable.

El punto final de un bloque es alcanzable si el bloque está vacío o si el punto final de la lista de instrucciones es alcanzable.

Un bloque (block) que contiene una o más instrucciones yield (§8.14) se denomina bloque de iteradores. Los bloques de iteradores se utilizan para implementar miembros de función como iteradores (§10.14). A los bloques de iteradores se les aplican algunas restricciones adicionales:

* Se trata de un error en tiempo de compilación si una instrucción return aparece en un bloque de iteradores (pero se permiten instrucciones yield return).
* Se trata de un error en tiempo de compilación si un bloque de iteradores contiene contexto no seguro (§18.1). Un bloque de iteradores siempre define un contexto seguro incluso si su declaración está anidada en un contexto que no lo es.

### Listas de instrucciones

Una lista de instrucciones está formada por una o varias instrucciones escritas secuencialmente. Las listas de instrucciones aparecen en bloques (block) (§8.2) y en bloques de modificadores (switch-block) (§8.7.2).

statement-list:  
statement  
statement-list statement

Cuando se ejecuta una lista de instrucciones, se transfiere el control a la primera instrucción. Cuando el control alcanza el punto final de una instrucción, se transfiere a la siguiente instrucción. Cuando el control alcanza el punto final de la última instrucción, se transfiere al punto final de la lista de instrucciones.

Una instrucción que forma parte de una lista de instrucciones es alcanzable si se cumple al menos una de las siguientes condiciones:

* Es la primera instrucción y la propia lista de instrucciones es alcanzable.
* El punto final de la instrucción anterior es alcanzable.
* La instrucción es una instrucción con etiqueta y una instrucción goto alcanzable hace referencia a la etiqueta.

El punto final de una lista de instrucciones es alcanzable si el punto final de la última instrucción de la lista es alcanzable.

## Instrucción vacía

Una instrucción vacía (empty-statement) no hace nada.

empty-statement:  
;

Una instrucción vacía se utiliza cuando no hay operaciones que realizar en un contexto donde se requiere una instrucción.

Cuando se ejecuta una instrucción vacía, simplemente se transfiere el control al punto final de la instrucción. Por lo tanto, el punto final de una instrucción vacía es alcanzable si la instrucción vacía es alcanzable.

Puede utilizar una instrucción vacía cuando escriba una instrucción while sin cuerpo:

bool ProcessMessage() {...}

void ProcessMessages() {  
 while (ProcessMessage())  
 ;  
}

También puede utilizarla para declarar una etiqueta justo antes de la llave de cierre “}” de un bloque:

void F() {  
 ...

if (done) goto exit;  
 ...

exit: ;  
}

## Instrucciones con etiqueta

Una instrucción con etiqueta (labeled-statement) permite agregar una etiqueta por delante de una instrucción. Las instrucciones con etiqueta se pueden incluir en bloques, pero no están permitidas como instrucciones incrustadas.

labeled-statement:  
identifier : statement

Una instrucción con etiqueta declara una etiqueta con el nombre especificado en el identificador (identifier). El ámbito de una etiqueta es el bloque entero en el cual se declara, incluyendo los bloques anidados. Se produce un error en tiempo de compilación cuando se definen dos etiquetas con el mismo nombre para que sus ámbitos se solapen.

Se puede hacer referencia a una etiqueta desde instrucciones goto (§8.9.3) dentro del ámbito de la etiqueta. Esto significa que las instrucciones goto pueden transferir el control dentro y fuera de los bloques, pero nunca en los bloques.

Las etiquetas tienen su propio espacio de declaración y no interfieren con otros identificadores. El ejemplo

int F(int x) {  
 if (x >= 0) goto x;  
 x = -x;  
 x: return x;  
}

es un ejemplo válido que utiliza el nombre x como parámetro y como etiqueta.

Una instrucción con etiqueta se ejecuta tal y como se ejecute la instrucción que sigue a la etiqueta.

Además de la capacidad de alcance que proporciona el flujo normal de control, una instrucción con etiqueta es alcanzable si una instrucción goto alcanzable hace referencia a la etiqueta. (Excepción: si una instrucción goto se encuentra dentro de otra try que incluye un bloque finally, la instrucción con etiqueta está fuera de try y el extremo del bloque finally no es alcanzable, entonces la instrucción con etiqueta tampoco es alcanzable desde esta instrucción goto).

## Instrucciones de declaración

Una instrucción de declaración (declaration-statement) declara una variable o una constante local. Las instrucciones de declaración se pueden incluir en bloques, pero no están permitidas como instrucciones incrustadas.

declaration-statement:  
local-variable-declaration ;  
local-constant-declaration ;

### Declaraciones de variables locales

Una declaración de variable local (local-variable-declaration) declara una o varias variables locales.

local-variable-declaration:  
local-variable-type local-variable-declarators

local-variable-type:  
type  
var

local-variable-declarators:  
local-variable-declarator  
local-variable-declarators , local-variable-declarator

local-variable-declarator:  
identifier  
identifier = local-variable-initializer

local-variable-initializer:  
expression  
array-initializer

El tipo de variable local (local-variable-type) de una declaración de variable local (local-variable-declaration) especifica directamente el tipo de las variables introducidas por la declaración, o indica con el identificador var que el tipo se debería inferir basándose en un inicializador. El tipo aparece seguido de una lista de declaradores de variable local (local-variable-declarator), cada una de las cuales incluye una nueva variable. Un declarador de variable local (local-variable-declarator) está formado por un identificador (identifier) que da nombre a la variable, que opcionalmente puede ir seguido del token “=” y de un inicializador de variable local (local-variable-initializer) que establece el valor inicial de la variable.

En el contexto de una declaración de variable local, el identificador var actúa como una palabra clave (§2.4.3). Cuando el tipo de variable local (local-variable-type) se especifica como var y no hay ningún tipo con nombre var en el ámbito, la declaración es una declaración de variable local tipificada implícitamente, cuyo tipo se infiere del tipo de la expresión de inicializador asociada. Las declaraciones de variable local tipificada implícitamente están sujetas a las siguientes restricciones:

* La declaración de variable local (local-variable-declaration) no puede incluir varios declaradores de variable local (local-variable-declarator).
* El declarador de variable local (local-variable-declarator) debe incluir un inicializador de variable local (local-variable-initializer).
* El inicializador de variable local (local-variable-initializer) debe ser una expresión (expression).
* La expresión (expression) del inicializador debe tener un tipo en tiempo de compilación.
* La expresión (expression) del inicializador no puede hacer referencia a la variable declarada en sí

A continuación, se muestran ejemplos de declaraciones de variable local tipificada implícitamente:

var x; // Error, no initializer to infer type from  
var y = {1, 2, 3}; // Error, array initializer not permitted  
var z = null; // Error, null does not have a type  
var u = x => x + 1; // Error, anonymous functions do not have a type  
var v = v++; // Error, initializer cannot refer to variable itself

Para obtener el valor de una variable local en una expresión se utiliza un nombre simple (simple-name) (§7.6.2) y, para modificarlo, se realiza una asignación (assignment) (§7.17). Una variable local debe estar asignada definitivamente (§5.3) en cada ubicación donde se obtenga su valor.

El ámbito de una variable local declarada en una declaración de variable local (local-variable-declaration) es el bloque donde se produce la declaración. Supone un error hacer referencia a una variable local en una posición textual que precede al declarador de la variable local (local-variable-declarator). Dentro del ámbito de una variable local, supone un error en tiempo de compilación declarar otra variable o constante local con el mismo nombre.

Una declaración de variable local que declara varias variables equivale a varias declaraciones de una sola variable con el mismo tipo. Un inicializador de variable (variable-initializer) en una declaración de variable local es en realidad una instrucción de asignación que se inserta inmediatamente después de la declaración.

El ejemplo

void F() {  
 int x = 1, y, z = x \* 2;  
}

es idéntico a

void F() {  
 int x; x = 1;  
 int y;  
 int z; z = x \* 2;  
}

En una declaración de variable local tipificada implícitamente, se toma el tipo de la variable local que se declara para que sea el mismo tipo de la expresión utilizada para inicializar la variable. Por ejemplo:

var i = 5;  
var s = "Hello";  
var d = 1.0;  
var numbers = new int[] {1, 2, 3};  
var orders = new Dictionary<int,Order>();

Las declaraciones de variable local tipificada implícitamente de arriba son precisamente equivalentes a las siguientes declaraciones con asignación de tipo explícita:

int i = 5;  
string s = "Hello";  
double d = 1.0;  
int[] numbers = new int[] {1, 2, 3};  
Dictionary<int,Order> orders = new Dictionary<int,Order>();

### Declaraciones de constantes locales

Una declaración de constante local (local-variable-declarator) declara una o varias constantes locales.

local-constant-declaration:  
const type constant-declarators

constant-declarators:  
constant-declarator  
constant-declarators , constant-declarator

constant-declarator:  
identifier = constant-expression

El tipo (type) de una declaración de constante local (local-constant-declaration) especifica el tipo de las constantes que se incluyen en la declaración. El tipo viene seguido de una lista de declaradores de constante (constant-declarator), cada uno de los cuales incluye una nueva constante. Un declarador de constante (constant-declarator) consta de un identificador (identifier) que da nombre a la constante, seguido del token “=” y de una expresión constante (constant-expression) (§7.19) que establece el valor de la constante.

El tipo (type) y la expresión constante (constant-expression) de una declaración de constante local deben seguir las reglas de declaración de miembros de constantes (§10.4).

Para obtener el valor de una constante local en una expresión, se utiliza el nombre simple (simple-name) (§7.6.2).

El ámbito de una constante local es el bloque donde se produce la declaración. Es un error hacer referencia a una constante local en una posición textual anterior a su declarador de constante (constant-declarator). Dentro del ámbito de una constante local, supone un error en tiempo de compilación declarar otra variable o constante local con el mismo nombre.

Una declaración de constante local que declara varias constantes equivale a varias declaraciones de una sola constante con el mismo tipo.

## Instrucciones de expresiones

Una instrucción de expresión (expression-statement) evalúa una expresión determinada. El valor calculado por la expresión, en el caso de haber alguno, se descarta.

expression-statement:  
statement-expression ;

statement-expression:  
invocation-expression  
object-creation-expression  
assignment  
post-increment-expression  
post-decrement-expression  
pre-increment-expression  
pre-decrement-expression  
await-expression

No todas las expresiones pueden ser instrucciones. En concreto, expresiones como x + y y x == 1, que simplemente calculan un valor (el cual será descartado), no se admiten como instrucciones.

La ejecución de una instrucción de expresión (expression-statement) evalúa la expresión que contiene y después transfiere el control al punto final de la instrucción de expresión. Por lo tanto, el punto final de una instrucción de expresión (expression-statement) es alcanzable si dicha instrucción también lo es.

## Instrucciones de selección

Las instrucciones de selección (selection-statement) seleccionan una de las instrucciones que se van a ejecutar en función del valor de alguna expresión.

selection-statement:  
if-statement  
switch-statement

### Instrucción If

La instrucción if selecciona una instrucción que se va a ejecutar basándose en el valor de una expresión booleana.

if-statement:  
if ( boolean-expression ) embedded-statement  
if ( boolean-expression ) embedded-statement else embedded-statement

La sección else se asocia a la instrucción if anterior más cercana permitida por la sintaxis. Por lo tanto, una instrucción if con el formato

if (x) if (y) F(); else G();

equivale a

if (x) {  
 if (y) {  
 F();  
 }  
 else {  
 G();  
 }  
}

Una instrucción if se ejecuta de la siguiente forma:

* Se evalúa la expresión booleana (boolean-expression) (§7.20).
* Si la expresión booleana devuelve true, el control se transfiere a la primera instrucción incrustada. Cuando el control alcanza el punto final de dicha instrucción, se transfiere al punto final de la instrucción if.
* Si la expresión booleana devuelve false y existe una sección else, el control se transfiere a la segunda instrucción incrustada. Cuando el control alcanza el punto final de dicha instrucción, se transfiere al punto final de la instrucción if.
* Si la expresión booleana devuelve false y no existe una sección else, el control se transfiere al punto final de la instrucción if.

La primera instrucción incrustada de una instrucción if es alcanzable si la instrucción if es alcanzable y la expresión booleana no tiene el valor constante false.

La segunda instrucción incrustada de una instrucción if, si existe, es alcanzable si la instrucción if es alcanzable y la expresión booleana no tiene el valor constante true.

El punto final de una instrucción if es alcanzable si el punto final de al menos una de sus instrucciones incrustadas es alcanzable. Además, el punto final de una instrucción if que no tiene sección else es alcanzable si la instrucción if es alcanzable y la expresión booleana no tiene el valor constante true.

### Instrucción Switch

La instrucción switch (switch-statement) selecciona una lista de instrucciones que se van a ejecutar que tengan asociada una etiqueta switch (switch-label) que se corresponda con el valor de la expresión switch.

switch-statement:  
switch ( expression ) switch-block

switch-block:  
{ switch-sectionsopt }

switch-sections:  
switch-section  
switch-sections switch-section

switch-section:  
switch-labels statement-list

switch-labels:  
switch-label  
switch-labels switch-label

switch-label:  
case constant-expression :  
default :

Una instrucción switch (switch-statement) está formada por la palabra clave switch, seguida de una expresión entre paréntesis (denominada expresión switch) y de un bloque switch (switch-block). El bloque switch (switch-block) consiste en cero o varias secciones de switch (switch-section), encerradas entre llaves. Cada sección de switch (switch-section) está formada por una o varias etiquetas switch (switch-label) seguidas de una lista de instrucciones (statement-list) (§8.2.1).

El tipo aplicable de una instrucción switch está establecido por la expresión switch.

* Si el tipo de la expresión switch es sbyte, byte, short, ushort, int, uint, long, ulong, bool, char, string o un tipo enum (enum-type), o si es el tipo que acepta valores NULL correspondiente a uno de estos tipos, entonces es el tipo aplicable de la instrucción switch.
* En caso contrario, debe existir una conversión implícita definida por el usuario (§6.4) del tipo de la expresión switch a uno de los posibles tipos aplicables: sbyte, byte, short, ushort, int, uint, long, ulong, char, string o un tipo que acepta valores NULL correspondiente a uno de esos tipos.
* De lo contrario, si no existe tal conversión implícita o existe más de una, se producirá un error en tiempo de compilación.

La expresión constante de cada etiqueta case debe denotar un valor convertible implícitamente (§6.1) en el tipo aplicable de la instrucción switch. Si dos o más etiquetas case de la misma instrucción switch especifican el mismo valor constante, se producirá un error en tiempo de compilación.

Puede existir como máximo una etiqueta default en una instrucción switch.

Una instrucción switch se ejecuta de la siguiente forma:

* Se evalúa la expresión switch y se convierte en el tipo aplicable.
* Si una de las constantes especificadas en una etiqueta case de la misma instrucción switch es igual al valor de la expresión switch, el control se transfiere a la lista de instrucciones que están a continuación de la etiqueta case.
* Si ninguna de las constantes especificadas en las etiquetas case de la misma instrucción switch es igual al valor de la expresión switch y existe una etiqueta default, el control se transfiere a la lista de instrucciones que aparece a continuación de la etiqueta default.
* Si ninguna de las constantes especificadas en las etiquetas case de una instrucción switch es igual al valor de la expresión switch y no existe una etiqueta default, el control se transfiere al punto final de la instrucción switch.

Si el punto final de la lista de instrucciones de una sección de switch es alcanzable, se producirá un error en tiempo de compilación. Esto se conoce como regla “sin paso explícito”. El ejemplo

switch (i) {  
case 0:  
 CaseZero();  
 break;  
case 1:  
 CaseOne();  
 break;  
default:  
 CaseOthers();  
 break;  
}

es válido porque ninguna sección de switch tiene un punto final alcanzable. A diferencia de C y C++, la ejecución de una sección switch no permite el “paso explícito” a la siguiente sección de switch, y el ejemplo

switch (i) {  
case 0:  
 CaseZero();  
case 1:  
 CaseZeroOrOne();  
default:  
 CaseAny();  
}

da como resultado un error en tiempo de compilación. Para ejecutar una sección de switch después de la ejecución de otra sección de switch, debe utilizar una instrucción goto case o goto default explícita:

switch (i) {  
case 0:  
 CaseZero();  
 goto case 1;  
case 1:  
 CaseZeroOrOne();  
 goto default;  
default:  
 CaseAny();  
 break;  
}

Una sección switch (switch-section) admite varias etiquetas. El ejemplo

switch (i) {  
case 0:  
 CaseZero();  
 break;  
case 1:  
 CaseOne();  
 break;  
case 2:  
default:  
 CaseTwo();  
 break;  
}

es válido. El ejemplo no infringe la regla “sin paso explícito” porque las etiquetas case 2: y default: forman parte de la misma sección switch (switch-section).

La regla “sin paso explícito” evita una clase de errores comunes que se producen en C y C++ cuando se omiten involuntariamente instrucciones break. Además, gracias a esta regla, las secciones de switch de una instrucción switch se pueden reorganizar arbitrariamente sin afectar al comportamiento de la instrucción. Por ejemplo, las secciones de la instrucción switch anterior se pueden colocar en orden inverso sin modificar el comportamiento de la instrucción:

switch (i) {  
default:  
 CaseAny();  
 break;  
case 1:  
 CaseZeroOrOne();  
 goto default;  
case 0:  
 CaseZero();  
 goto case 1;  
}

La lista de instrucciones de una sección de switch termina normalmente con una instrucción break, goto case o goto default, pero también se admite cualquier sintaxis que represente el punto final de la lista de instrucciones en inalcanzable. Por ejemplo, una instrucción while controlada mediante la expresión booleana true nunca alcanzará su punto final. Igualmente, una instrucción throw o return siempre transfiere el control a otra parte y nunca alcanza su punto final. Por tanto, el siguiente ejemplo es correcto:

switch (i) {  
case 0:  
 while (true) F();  
case 1:  
 throw new ArgumentException();  
case 2:  
 return;  
}

El tipo aplicable en una instrucción switch puede ser el tipo string. Por ejemplo:

void DoCommand(string command) {  
 switch (command.ToLower()) {  
 case "run":  
 DoRun();  
 break;  
 case "save":  
 DoSave();  
 break;  
 case "quit":  
 DoQuit();  
 break;  
 default:  
 InvalidCommand(command);  
 break;  
 }  
}

De la misma forma que los operadores de igualdad (§7.10.7), la instrucción switch distingue mayúsculas de minúsculas y ejecutará una determinada sección sólo si la cadena de expresión switch coincide exactamente con una constante de etiqueta case.

Cuando el tipo aplicable en una instrucción switch es string, se admite el valor null como constante de etiqueta case.

Las listas de instrucciones (statement-list) de bloque switch (switch-block) pueden contener instrucciones de declaración (§8.5). El ámbito de una variable o constante local declarada en un bloque switch es el propio bloque.

Dentro de un bloque switch, el significado de un nombre utilizado en un contexto de expresión siempre debe ser el mismo (§7.6.2.1).

La lista de instrucciones de una sección de switch determinada es alcanzable si la instrucción switch es alcanzable y se cumple al menos una de las condiciones siguientes:

* La expresión switch no es un valor constante.
* La expresión switch es un valor de constante que coincide con una etiqueta case de la sección switch.
* La expresión switch es un valor de constante que no coincide con ninguna etiqueta case y la sección de switch contiene la etiqueta default.
* Una instrucción default de goto case o goto alcanzable hace referencia a una etiqueta switch de la sección.

El punto final de una instrucción switch es alcanzable si se cumple al menos una de las siguientes condiciones:

* La instrucción switch contiene una instrucción break alcanzable que provoca la salida de la instrucción switch.
* La instrucción switch es alcanzable, la expresión switch no es un valor constante y no existe etiqueta default.
* La instrucción switch es alcanzable, la expresión switch es una constante que no coincide con ninguna etiqueta case y no existe etiqueta default.

## Instrucciones de iteración

Las instrucciones de iteración (iteration-statement) ejecutan repetidas veces una instrucción incrustada.

iteration-statement:  
while-statement  
do-statement  
for-statement  
foreach-statement

### Instrucción While

La instrucción while ejecuta una instrucción incrustada cero o varias veces dependiendo de una condición.

while-statement:  
while ( boolean-expression ) embedded-statement

Una instrucción while se ejecuta de la siguiente forma:

* Se evalúa la expresión booleana (boolean-expression) (§7.20).
* Si la expresión booleana devuelve true, el control se transfiere a la instrucción incrustada. Cuando el control alcanza el punto final de la instrucción incrustada (posiblemente desde la ejecución de una instrucción continue), se transfiere al inicio de la instrucción while.
* Si la expresión booleana devuelve false, el control se transfiere al punto final de la instrucción while.

Dentro de la instrucción incrustada de la instrucción while, puede utilizar una instrucción break (§8.9.1) para transferir el control al punto final de la instrucción while (terminando así la iteración de la instrucción incrustada), y una instrucción continue (§8.9.2) para transferir el control al punto final de la instrucción incrustada (de esta forma se realizará otra iteración de la instrucción while).

La primera instrucción incrustada de una instrucción while es alcanzable si la instrucción while es alcanzable y la expresión booleana no tiene el valor constante false.

El punto final de una instrucción while es alcanzable si se cumple al menos una de las siguientes condiciones:

* La instrucción while contiene una instrucción break alcanzable que provoca la salida de la instrucción while.
* La instrucción while es alcanzable y la expresión booleana no tiene el valor constante true.

### Instrucción Do

La instrucción do ejecuta una instrucción incrustada una o varias veces dependiendo de una condición.

do-statement:  
do embedded-statement while ( boolean-expression ) ;

Una instrucción do se ejecuta de la siguiente forma:

* El control se transfiere a la instrucción incrustada.
* Cuando el control alcanza el punto final de la instrucción incrustada (posiblemente desde la ejecución de una instrucción continue), se evalúa la expresión booleana (boolean-expression) (§7.20). Si la expresión booleana devuelve true, el control se transfiere al principio de la instrucción do. En caso contrario, el control se transfiere al punto final de la instrucción do.

Dentro de la instrucción incrustada de la instrucción do, puede utilizar una instrucción break (§8.9.1) para transferir el control al punto final de la instrucción do (terminando así la iteración de la instrucción incrustada), y una instrucción continue (§8.9.2) para transferir el control al punto final de la instrucción incrustada.

La instrucción incrustada de una instrucción do es alcanzable si la instrucción do es alcanzable.

El punto final de una instrucción do es alcanzable si se cumple al menos una de las siguientes condiciones:

* La instrucción do contiene una instrucción break alcanzable que provoca la salida de la instrucción do.
* El punto final de la instrucción incrustada es alcanzable y la expresión booleana no tiene el valor constante true.

### Instrucción For

La instrucción for evalúa primero una secuencia de expresiones de inicialización y, mientras se cumpla una determinada condición, ejecuta repetidas veces una instrucción incrustada y evalúa una secuencia de expresiones de iteración.

for-statement:  
for ( for-initializeropt ; for-conditionopt ; for-iteratoropt ) embedded-statement

for-initializer:  
local-variable-declaration  
statement-expression-list

for-condition:  
boolean-expression

for-iterator:  
statement-expression-list

statement-expression-list:  
statement-expression  
statement-expression-list , statement-expression

El inicializador for (for-initializer), si existe, es una declaración de variable local (local-variable-declaration) (§8.5.1) o una lista de expresiones de instrucción (statement-expression) (§8.6) separadas por comas. El ámbito de una variable local declarada por un inicializador for (for-initializer) se inicia en el declarador de variable local (local-variable-declarator) y se extiende hasta el final de la instrucción incrustada. El ámbito incluye la condición for (for-condition) y el iterador for (for-iterator).

La condición for (for-condition), si existe, debe ser una expresión booleana (boolean-expression) (§7.20).

El iterador for (for-iterator), si existe, consiste en una lista de expresiones de instrucción (statement-expression) (§8.6) separadas por comas.

Una instrucción for se ejecuta de la siguiente forma:

* Si existe un inicializador for (for-initializer), se ejecutan los inicializadores de variable o las expresiones de instrucción en el orden en el que se hayan codificado. Este paso sólo se realiza una vez.
* Si existe una condición for (for-condition), se evalúa.
* Si no existe una condición for (for-condition), o existe y la evaluación devuelve true, el control se transfiere a la instrucción incrustada. Cuando el control alcanza el punto final de la instrucción incrustada (posiblemente desde la ejecución de una instrucción continue), las expresiones del iterador for (for-iterator), si existen, se evalúan en secuencia y, a continuación, se realiza una nueva iteración empezando por la evaluación de la condición for (for-condition), como se describe en el paso anterior.
* Si existe una condición for (for-condition) y devuelve false, el control se transfiere al punto final de la instrucción for.

Dentro de la instrucción incrustada de una instrucción for, se puede utilizar una instrucción break (§8.9.1) para transferir el control al punto final de la instrucción for (terminando así la iteración de la instrucción incrustada), y una instrucción continue (§8.9.2) para transferir el control al punto final de la instrucción incrustada (de esta manera, se ejecuta el iterador for (for-iterator) y se realiza otra iteración de la instrucción for, empezando por la condición for (for-condition).

La instrucción incrustada de una instrucción for es alcanzable si se cumple alguna de las condiciones siguientes:

* La instrucción for es alcanzable y no hay ninguna condición for (for-condition).
* La instrucción for es alcanzable y hay una condición for (for-condition), pero esta última no tiene el valor constante false.

El punto final de una instrucción for es alcanzable si se cumple al menos una de las siguientes condiciones:

* La instrucción for contiene una instrucción break alcanzable que provoca la salida de la instrucción for.
* La instrucción for es alcanzable y hay una condición for (for-condition), pero esta última no tiene el valor constante true.

### Instrucción Foreach

La instrucción foreach enumera los elementos de una colección, ejecutando una instrucción incrustada para cada elemento de la colección.

foreach-statement:  
foreach ( local-variable-type identifier in expression ) embedded-statement

El tipo (type) y el identificador (identifier) de una instrucción foreach declaran la variable de iteración de la instrucción. Si se da el identificador var como tipo de variable local (local-variable-type) y no hay un tipo denominado var en el ámbito, se dice que la variable de iteración es una variable de iteración con asignación de tipo implícita, y se toma su tipo para que sea el tipo de elemento de la instrucción foreach, como se especifica a continuación. La variable de iteración es una variable local de sólo lectura con un ámbito que se extiende a lo largo de toda la instrucción incrustada. Durante la ejecución de una instrucción foreach, la variable de iteración representa el elemento de colección para el que se está realizando la iteración en ese momento. Se producirá un error en tiempo de compilación si la instrucción incrustada intenta modificar la variable de iteración (por medio de asignación o utilizando los operadores ++ y ‑‑), o bien si la pasa como un parámetro ref u out.

En el ejemplo siguiente, para mayor brevedad, IEnumerable, IEnumerator, IEnumerable<T> y IEnumerator<T> hacen referencia a los tipos correspondientes en los espacios de nombres System.Collections y System.Collections.Generic.

El procesamiento en tiempo de compilación de una instrucción foreach determina primero el tipo de colección, el tipo de enumerador y el tipo de elemento de la expresión. Esta determinación se realiza de la siguiente forma:

* Si el tipo X de expresión (expression) es un tipo de matriz, hay una conversión implícita de X a la interfaz IEnumerable (ya que System.Array implementa esta interfaz). El tipo de colección es la interfaz IEnumerable, el tipo de enumerador es la interfaz IEnumerator y el tipo de elemento es el tipo de elemento del tipo de matriz X.
* Si el tipo X de expresión (expression) es dynamic, hay una conversión implícita de la expresión a la interfaz IEnumerable (§6.1.8). El tipo de colección es la interfaz IEnumerable y el tipo de enumerador es la interfaz IEnumerator. Si se da el identificador var como el tipo de variable local (local-variable-type) el tipo de elemento es dynamic, de lo contrario es object.
* En caso contrario, determine si el tipo X tiene un método GetEnumerator apropiado:
* Realice la búsqueda de miembros en el tipo X con el identificador GetEnumerator y con ningún argumento de tipo. Si la búsqueda de miembros no produce ningún resultado, o si produce una ambigüedad, o un resultado que no es un grupo de métodos, compruebe una interfaz enumerable como se describe más adelante. Es aconsejable que se emita una advertencia si la búsqueda de miembros no produce ningún resultado o si produce resultados pero no son grupos de métodos.
* Realice la resolución de sobrecargas utilizando el grupo de métodos resultante y una lista de argumentos vacía. Si la resolución de sobrecargas no tiene como resultado métodos aplicables, si produce una ambigüedad, o un método individual mejor pero que es estático o no público, compruebe una interfaz enumerable como se describe más adelante. Es aconsejable que se emita una advertencia si la resolución de sobrecargas no produce ningún método aplicable o si produce resultados pero no son métodos de instancia pública sin ambigüedad.
* Si el tipo de resultado E del método GetEnumerator no es una clase, struct o tipo de interfaz, se produce un error y no se realizan más pasos.
* La búsqueda de miembros se realiza en E con el identificador Current y ningún argumento de tipo. Si la búsqueda de miembros no produce ningún resultado, el resultado es un error, o el resultado no es una propiedad de instancia pública que permita la lectura, se produce un error y no se realizan más pasos.
* La búsqueda de miembros se realiza en E con el identificador MoveNext y ningún argumento de tipo. Si la búsqueda de miembros no produce ningún resultado, el resultado es un error, o el resultado no es un grupo de métodos, se produce un error y no se realizan más pasos.
* La resolución de sobrecargas se realiza en el grupo de métodos con una lista de argumentos vacía. Si la resolución de sobrecargas no tiene como resultado métodos aplicables, si produce una ambigüedad, o un método individual mejor pero que es estático o no público, o su tipo devuelto no es bool, se produce un error y no se realizan más pasos.
* El tipo de colección es X, el tipo de enumerador es E, y el tipo de elemento es el tipo de la propiedad Current.
* En caso contrario, compruebe una interfaz enumerable:
* Si entre todos los tipos Ti para los que hay una conversión implícita de X a IEnumerable<Ti>, hay un tipo único T de modo que T no es dynamic y para el resto de Ti hay una conversión implícita de IEnumerable<T> a IEnumerable<Ti>, entonces el tipo de recopilación es la interfaz IEnumerable<T>, el tipo de enumerador es la interfaz IEnumerator<T> y el tipo de elemento es T.
* En caso contrario, si hay más de un tipo T, se produce un error y no se realizan más pasos.
* De lo contrario, si hay una conversión implícita de X a la interfaz System.Collections.IEnumerable, el tipo de colección es esta interfaz, el tipo de enumerador es la interfaz System.Collections.IEnumerator y el tipo de elemento es object.
* En caso contrario, se produce un error y no se realizan más pasos.

Los pasos anteriores, si son correctos, producen un tipo de colección C sin ninguna ambigüedad, un tipo de enumerador E y un tipo de elemento T. Una instrucción foreach con la siguiente estructura

foreach (V v in x) embedded-statement

se expande a:

{  
 E e = ((C)(x)).GetEnumerator();  
 try {  
 while (e.MoveNext()) {  
 V v = (V)(T)e.Current;  
 embedded-statement  
 }  
 }  
 finally {  
 … // Desechar e  
 }  
}

La variable e no es visible ni accesible para la expresión x o la declaración incrustada o cualquier otro código fuente del programa. La variable v es de solo lectura en la instrucción incrustada. Si no hay una conversión explícita (§6.2) de T (tipo de elemento) a V (tipo de variable local (local-variable-type) de la instrucción foreach), se produce un error y no se realizan más pasos. Si x tiene el valor null, se inicia una excepción System.NullReferenceException en tiempo de ejecución.

Se permite que una implementación implemente una instrucción foreach dada de manera diferente, por ejemplo, por motivos de rendimiento, siempre que el comportamiento sea coherente con la expansión anterior.

La posición de v dentro del bucle while es importante para la forma en que lo captura una función anónima que aparece en la instrucción incrustada (embedded-statement).

Por ejemplo:

int[] values = { 7, 9, 13 };  
Action f = null;

foreach (var value in values)  
{  
 if (f == null) f = () => Console.WriteLine("First value: " + value);  
}

f();

Si v se declaró fuera del bucle while, se compartiría entre todas las iteraciones, y su valor después del bucle sería el valor final, 13, que es lo que imprimiría la invocación de f. En su lugar, debido a que cada iteración tiene su propia variable v, la capturada por f en la primera iteración seguirá conteniendo el valor 7, que es lo que se imprimirá. (Nota: las versiones anteriores de C# declaraban v fuera del bucle while).

El cuerpo del bloque finally se construye de acuerdo con los siguientes pasos:

* Si hay una conversión implícita desde E a la interfaz System.IDisposable, entonces
* Si E es un tipo de valor que acepta valores NULL, entonces la cláusula finally se expande hasta el equivalente semántico de:

finally {  
 ((System.IDisposable)e).Dispose();  
}

* De lo contrario, la cláusula finally se expande hasta el equivalente semántico de:

finally {  
 if (e != null) ((System.IDisposable)e).Dispose();  
}

excepto si E es un tipo de valor o un parámetro de tipo cuya instancia se ha creado para un tipo de valor, entonces la conversión de e a System.IDisposable no producirá una conversión boxing.

* En caso contrario, si E es un tipo sealed, la cláusula finally se expande a un bloque vacío:

finally {  
}

* En caso contrario, la cláusula finally se expande a:

finally {  
 System.IDisposable d = e as System.IDisposable;  
 if (d != null) d.Dispose();  
}

La variable local d no es visible ni accesible para cualquier código de usuario. En particular, no entra en conflicto con otras variables cuyo ámbito incluya el bloque finally.

El orden en que foreach recorre los elementos de una matriz es el siguiente: en matrices unidimensionales, los elementos se recorren en orden creciente, empezando por el índice 0 y acabando por el índice Length – 1. Los elementos de matrices multidimensionales se recorren de manera que los índices de la dimensión del extremo derecho se incrementan en primer lugar, a continuación la dimensión situada inmediatamente a su izquierda y así sucesivamente hacia la izquierda.

El código del siguiente ejemplo muestra cada valor de una matriz bidimensional según el orden de los elementos:

using System;

class Test  
{  
 static void Main() {  
 double[,] values = {  
 {1.2, 2.3, 3.4, 4.5},  
 {5.6, 6.7, 7.8, 8.9}  
 };

foreach (double elementValue in values)  
 Console.Write("{0} ", elementValue);

Console.WriteLine();  
 }  
}

El resultado producido es el siguiente:

1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9

En el ejemplo

int[] numbers = { 1, 3, 5, 7, 9 };  
foreach (var n in numbers) Console.WriteLine(n);

el tipo de n se infiere que es int, el tipo de elemento de numbers.

## Instrucciones Jump

Las instrucciones de salto (jump-statement) transfieren el control de forma incondicional.

jump-statement:  
break-statement  
continue-statement  
goto-statement  
return-statement  
throw-statement

La posición a la que transfiere el control una instrucción de salto se denomina destino de la instrucción de salto.

Cuando se ejecuta una instrucción de salto dentro de un bloque y el destino se encuentra fuera del bloque, se dice que la instrucción de salto sale del bloque. Aunque una instrucción de salto puede transferir el control fuera de un bloque, no puede transferirlo dentro del bloque.

La ejecución de las instrucciones de salto se complica cuando intervienen instrucciones try. Si no existen instrucciones try, la instrucción de salto transfiere incondicionalmente el control a su destino. Pero si existen instrucciones try, la ejecución es más compleja. Si la instrucción de salto sale de uno o varios bloques try que llevan asociados bloques finally, el control se transfiere inicialmente al bloque finally de la instrucción try más interna. Cuando el control alcanza el punto final de un bloque finally, el control se transfiere al bloque finally de la siguiente instrucción envolvente try. Este proceso se repite hasta que se ejecuten los bloques finally de todas las instrucciones try que intervienen.

En el ejemplo

using System;

class Test  
{  
 static void Main() {  
 while (true) {  
 try {  
 try {  
 Console.WriteLine("Before break");  
 break;  
 }  
 finally {  
 Console.WriteLine("Innermost finally block");  
 }  
 }  
 finally {  
 Console.WriteLine("Outermost finally block");  
 }  
 }  
 Console.WriteLine("After break");  
 }  
}

los bloques finally asociados a las dos instrucciones try se ejecutan antes de transferir el control al destino de la instrucción de salto.

El resultado producido es el siguiente:

Before break  
Innermost finally block  
Outermost finally block  
After break

### Instrucción Break

La instrucción break sale de la instrucción envolvente switch, while, do, for o foreach más próxima.

break-statement:  
break ;

El destino de una instrucción break es el punto final de la instrucción envolvente switch, while, do, for o foreach más próxima. Si una instrucción break no se encuentra dentro de una instrucción switch, while, do, for o foreach, se produce un error en tiempo de compilación.

Cuando existen varias instrucciones switch, while, do, for o foreach anidadas, la instrucción break se aplica solo a la instrucción más interna. Para transferir el control a través de varios niveles de anidamiento, debe utilizar una instrucción goto (§8.9.3).

Una instrucción break no puede salir de un bloque finally (§8.10). Cuando aparece una instrucción break dentro de un bloque finally, el destino de la instrucción break debe encontrarse dentro del propio bloque finally; en caso contrario, se producirá un error en tiempo de compilación.

Una instrucción break se ejecuta de la siguiente forma:

* Si la instrucción break sale de uno o varios bloques try que llevan asociados bloques finally, el control se transfiere inicialmente al bloque finally de la instrucción try más interna. Cuando el control alcanza el punto final de un bloque finally, el control se transfiere al bloque finally de la siguiente instrucción envolvente try. Este proceso se repite hasta que se ejecuten los bloques finally de todas las instrucciones try que intervienen.
* El control se transfiere al destino de la instrucción break.

Como una instrucción break transfiere incondicionalmente el control a otra parte del código, el punto final de una instrucción break nunca es alcanzable.

### Instrucción continue

La instrucción continue inicia una nueva iteración de la instrucción envolvente while, do, for o foreach más próxima.

continue-statement:  
continue ;

El destino de una instrucción continue es el punto final de la instrucción incrustada de la instrucción envolvente while, do, for o foreach más próxima. Si una instrucción continue no se encuentra dentro de una instrucción while, do, for o foreach, se produce un error en tiempo de compilación.

Cuando existen varias instrucciones while, do, for, foreach o continue anidadas, la instrucción se aplica solo a la instrucción más interna. Para transferir el control a través de varios niveles de anidamiento, debe utilizar una instrucción goto (§8.9.3).

Una instrucción continue no puede salir de un bloque finally (§8.10). Cuando aparece una instrucción continue dentro de un bloque finally, el destino de la instrucción continue debe encontrarse dentro del propio bloque finally; en caso contrario, se producirá un error en tiempo de compilación.

Una instrucción continue se ejecuta de la siguiente forma:

* Si la instrucción continue sale de uno o varios bloques try que llevan asociados bloques finally, el control se transfiere inicialmente al bloque finally de la instrucción try más interna. Cuando el control alcanza el punto final de un bloque finally, el control se transfiere al bloque finally de la siguiente instrucción envolvente try. Este proceso se repite hasta que se ejecuten los bloques finally de todas las instrucciones try que intervienen.
* El control se transfiere al destino de la instrucción continue.

Debido a que una instrucción continue transfiere incondicionalmente el control a otra parte del código, el punto final de una instrucción continue nunca es alcanzable.

### Instrucción Goto

La instrucción goto transfiere el control a una instrucción marcada con una etiqueta.

goto-statement:  
goto identifier ;  
goto case constant-expression ;  
goto default ;

El destino de una instrucción de identificador (identifier) goto es la instrucción con etiqueta que lleva el nombre de dicha etiqueta. Si no existe una etiqueta con dicho nombre en el miembro de función actual, o si la instrucción goto no se encuentra dentro del ámbito de la etiqueta, se produce un error en tiempo de compilación. Esta regla permite el uso de una instrucción goto para transferir el control fuera de un ámbito anidado, pero no dentro de un ámbito anidado. En el ejemplo

using System;

class Test  
{  
 static void Main(string[] args) {  
 string[,] table = {  
 {"Red", "Blue", "Green"},  
 {"Monday", "Wednesday", "Friday"}  
 };

foreach (string str in args) {  
 int row, colm;  
 for (row = 0; row <= 1; ++row)  
 for (colm = 0; colm <= 2; ++colm)  
 if (str == table[row,colm])  
 goto done;

Console.WriteLine("{0} not found", str);  
 continue;  
 done:  
 Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);  
 }  
 }  
}

se utiliza una instrucción goto para transferir el control fuera de un ámbito anidado.

El destino de una instrucción goto case es la lista de instrucciones de la instrucción switch inmediatamente envolvente (§8.7.2), que contiene una etiqueta case con el valor constante dado. Si la instrucción goto case no está dentro de una instrucción switch, la expresión constante (constant-expression) no es convertible implícitamente (§6.1) al tipo aplicable en la instrucción envolvente switch más próxima, o si dicha instrucción switch no contiene una etiqueta case con el valor constante dado, se produce un error en tiempo de compilación.

El destino de una instrucción goto default es la lista de instrucciones de la instrucción switch inmediatamente envolvente (§8.7.2), que contiene una etiqueta default. Si la instrucción goto default no se encuentra dentro de una instrucción switch o la instrucción envolvente switch más próxima no contiene una etiqueta default, se produce un error en tiempo de compilación.

Una instrucción goto no puede salir de un bloque finally (§8.10). Cuando aparece una instrucción goto dentro de un bloque finally, el destino de la instrucción goto debe encontrarse dentro del propio bloque finally; en caso contrario, se producirá un error en tiempo de compilación.

Una instrucción goto se ejecuta de la siguiente forma:

* Si la instrucción goto sale de uno o varios bloques try que llevan asociados bloques finally, el control se transfiere inicialmente al bloque finally de la instrucción try más interna. Cuando el control alcanza el punto final de un bloque finally, el control se transfiere al bloque finally de la siguiente instrucción envolvente try. Este proceso se repite hasta que se ejecuten los bloques finally de todas las instrucciones try que intervienen.
* El control se transfiere al destino de la instrucción goto.

Debido a que una instrucción goto transfiere incondicionalmente el control a otra parte del código, el punto final de una instrucción goto nunca es alcanzable.

### Instrucción Return

La instrucción return devuelve el control al llamador del miembro de función en el que aparece la instrucción return.

return-statement:  
return expressionopt ;

Una instrucción return sin expresión solo puede utilizarse en un miembro de función que no calcule un valor, es decir, un método con el tipo de resultado (§10.6.10) void, el descriptor de acceso set de una propiedad o indizador, descriptores de acceso add y remove de un evento, un constructor de instancia, un constructor estático o un destructor.

Una instrucción return con expresión se puede utilizar sólo en un miembro de función que calcule un valor, es decir, un método con un tipo de resultado que no sea void, el descriptor de acceso get de una propiedad o indizador o un operador definido por el usuario. Debe existir una conversión implícita (§6.1) del tipo de la expresión al tipo de valor devuelto por el miembro de función que la contiene.

Las instrucciones return también pueden usarse en el cuerpo de expresiones de función anónima (§7.15) y pueden participar en la determinación de qué conversiones existen para esas funciones.

Se produce un error en tiempo de compilación cuando aparece una instrucción return en un bloque finally (§8.10).

Una instrucción return se ejecuta de la siguiente forma:

* Si la instrucción return especifica una expresión, se evalúa la expresión y el valor devuelto se convierte al tipo de valor devuelto por el miembro de función que la contiene mediante una conversión implícita. El resultado de la conversión se convierte en el valor de resultado generado por la función.
* Si la instrucción return está encerrada entre uno o varios bloques try o catch que llevan asociados bloques finally, el control se transfiere inicialmente al bloque finally de la instrucción try más interna. Cuando el control alcanza el punto final de un bloque finally, el control se transfiere al bloque finally de la siguiente instrucción envolvente try. Este proceso se repite hasta que se ejecuten los bloques finally de todas las instrucciones envolventes try que intervienen.
* Si la función contenedora no es asincrónica, el control se devuelve al llamador de la función contenedora junto con el valor de resultado, si hay alguno.
* Si la función contenedora es una función asincrónica, el control se devuelve al llamador actual y el valor de resultado, si hay alguno, se registra en la tarea devuelta, tal como se describe en §10.14.1.

Debido a que una instrucción return transfiere incondicionalmente el control a otra parte del código, el punto final de una instrucción return nunca es alcanzable.

### Instrucción throw

La instrucción throw produce una excepción.

throw-statement:  
throw expressionopt ;

Una instrucción throw con una expresión produce el valor resultante de evaluar la expresión. La expresión debe denotar un valor del tipo de clase System.Exception, de un tipo de clase que derive de System.Exception o de un parámetro de tipo cuya clase base efectiva derive de System.Exception. Si la evaluación de la expresión produce null, en su lugar se produce una excepción System.NullReferenceException.

Una instrucción throw sin expresión solo se puede utilizar en un bloque catch, en cuyo caso esta instrucción volverá a provocar la excepción que esté controlando en ese momento el bloque catch.

Debido a que una instrucción throw transfiere incondicionalmente el control a otra parte del código, el punto final de una instrucción throw nunca es alcanzable.

Cuando se produce una excepción E, el control se transfiere a la primera cláusula catch de una instrucción envolvente try que pueda controlar la excepción. El proceso que tiene lugar desde el punto de inicio de la excepción hasta el punto en que se transfiere el control a un controlador de excepciones adecuado es conocido con el nombre de propagación de excepción. La propagación de una excepción consiste en evaluar repetidamente los siguientes pasos hasta encontrar una cláusula catch que coincida con la excepción. En esta descripción, el punto de inicio es la ubicación desde la que se inicia la excepción.

* En el miembro de función actual, se examina cada instrucción try que envuelve al punto de inicio. Se evalúan los siguientes pasos para cada instrucción S, comenzando con la instrucción try más interna y terminando con la más externa:
* Si el bloque try de S encierra al punto de inicio y S tiene una o varias cláusulas catch, se examinan las cláusulas catch en orden de aparición hasta encontrar un controlador adecuado para la excepción. La primera cláusula catch que especifique un tipo de excepción T (o un parámetro de tipo que denote un tipo de excepción T en tiempo de ejecución) de tal modo que el tipo en tiempo de ejecución de E se derive de T se considera una coincidencia. Una cláusula catch general (§8.10) es una coincidencia para cualquier tipo de excepción. Si se encuentra una cláusula catch coincidente, la propagación de excepción se completa y se transfiere el control al bloque de la cláusula catch.
* En caso contrario, si el bloque try o un bloque catch de S encierra el punto de inicio y S tiene un bloque finally, el control se transfiere al bloque finally. Si el bloque finally inicia otra excepción, finaliza el procesamiento de la excepción actual. Si no, cuando el control alcanza el punto final del bloque finally, continúa con el procesamiento de la excepción actual.
* Si no se encontró un controlador de excepciones en la invocación de función actual, finaliza la invocación de la función y ocurre una de las acciones siguientes:
* Si la función actual no es asincrónica, los pasos anteriores se repiten para el llamador de la función con el punto de inicio que corresponda a la instrucción desde la que se invocó al miembro de función.
* Si la función actual es asincrónica y devuelve tareas, la excepción se registra en la tarea devuelta, que se pone en estado de error o cancelado tal como se describe en §10.14.1.
* Si la función actual es asincrónica y devuelve void, el contexto de sincronización del subproceso actual se notifica tal como se describe en §10.14.2.
* Si el procesamiento de la excepción finaliza todas las llamadas a miembros de función del subproceso actual indicando que el subproceso no ha encontrado controlador para la excepción, dicho subproceso finaliza. El impacto de esta terminación se define según la implementación.

## Instrucción try

La instrucción try proporciona un mecanismo para capturar las excepciones que ocurren durante la ejecución de un bloque. La instrucción try permite además especificar un bloque de código que siempre se ejecuta cuando el control abandona la instrucción try.

try-statement:  
try block catch-clauses  
try block finally-clause  
try block catch-clauses finally-clause

catch-clauses:  
specific-catch-clauses general-catch-clauseopt  
specific-catch-clausesopt general-catch-clause

specific-catch-clauses:  
specific-catch-clause  
specific-catch-clauses specific-catch-clause

specific-catch-clause:  
catch ( type identifieropt ) block

general-catch-clause:  
catch block

finally-clause:  
finally block

Existen tres formas posibles de instrucciones try:

* Un bloque try seguido de uno o varios bloques catch.
* Un bloque try seguido de un bloque finally.
* Un bloque try seguido de uno o varios bloques catch seguidos de un bloque finally.

Cuando una cláusula catch especifica un tipo (type), el tipo debe ser System.Exception, un tipo que derive de System.Exception o un tipo de parámetro de tipo cuya clase base efectiva se derive de System.Exception.

Cuando una cláusula catch especifica tanto un tipo de clase (class-type) como un identificador (identifier), se declara una variable de excepción con el nombre y tipo dados. La variable de excepción es una variable local con un ámbito que se extiende a lo largo de todo el bloque catch. Durante la ejecución de un bloque catch, la variable de excepción representa la excepción que se está controlando en ese momento. Desde el punto de vista de comprobación de asignación definitiva, la variable de excepción se considera asignada definitivamente en todo su ámbito.

Si la cláusula catch no incluye el nombre de una variable de excepción, no es posible tener acceso al objeto de excepción en el bloque catch.

Una cláusula catch general es una cláusula catch que no especifica el tipo ni la variable de excepción. Una instrucción try solo puede tener una cláusula catch general y, si existe, debe ser la última cláusula catch.

Algunos lenguajes de programación pueden aceptar excepciones que no son representables como un objeto derivado de System.Exception, aunque estas excepciones nunca pueden ser generadas por código C#. Para detectar este tipo de excepciones se puede utilizar una cláusula catch general. Por lo tanto, una cláusula catch general es semánticamente diferente de otra que especifica el tipo System.Exception, ya que la primera también puede detectar excepciones de otros lenguajes.

Las cláusulas catch se examinan en orden léxico con el fin de encontrar un controlador para la excepción. Si una cláusula catch especifica un tipo que es igual o derivado de un tipo especificado en una cláusula catch anterior de la misma instrucción try, se produce un error en tiempo de compilación. Si no existiera esta restricción, sería posible escribir cláusulas catch inalcanzables.

Dentro de un bloque catch puede utilizar una instrucción throw (§8.9.5) sin expresión para volver a iniciar la excepción que capturó el bloque catch. Las asignaciones a una variable de excepción no modifican la excepción que vuelve a iniciarse.

En el ejemplo

using System;

class Test  
{  
 static void F() {  
 try {  
 G();  
 }  
 catch (Exception e) {  
 Console.WriteLine("Exception in F: " + e.Message);  
 e = new Exception("F");  
 throw; // re-throw  
 }  
 }

static void G() {  
 throw new Exception("G");  
 }

static void Main() {  
 try {  
 F();  
 }  
 catch (Exception e) {  
 Console.WriteLine("Exception in Main: " + e.Message);  
 }  
 }  
}

el método F captura una excepción, escribe información de diagnóstico en la consola, modifica la variable de excepción y vuelve a iniciar la excepción. La excepción que se vuelve a iniciar es la excepción original, de modo que el resultado producido es el siguiente:

Exception in F: G  
Exception in Main: G

Si el primer bloque catch iniciara e en lugar de volver a iniciar la excepción actual, el resultado producido sería el siguiente:

Exception in F: G  
Exception in Main: F

Se produce un error en tiempo de compilación cuando una instrucción break, continue o goto transfiere el control fuera del bloque finally. Cuando aparece una instrucción break, continue o goto dentro de un bloque finally, el destino de la instrucción debe encontrarse dentro del propio bloque finally. En caso contrario, se producirá un error en tiempo de compilación.

Se produce un error en tiempo de compilación cuando aparece una instrucción return en un bloque finally.

Una instrucción try se ejecuta de la siguiente forma:

* El control se transfiere al bloque try.
* Cuando el control alcanza el punto final del bloque try:
* Si la instrucción try tiene un bloque finally, se ejecuta dicho bloque finally.
* El control se transfiere al punto final de la instrucción try.
* Si se propaga una excepción a la instrucción try durante la ejecución del bloque try:
* Si existen cláusulas catch, se examinan en orden de aparición para buscar un controlador adecuado para la excepción. La primera cláusula catch que especifique el tipo de excepción o un tipo base del tipo de excepción se considera una coincidencia. Una cláusula catch general es una coincidencia para cualquier tipo de excepción. Si se encuentra una cláusula catch coincidente:
* Si la cláusula catch coincidente declara una variable de excepción, se asigna el objeto de excepción a dicha variable.
* El control se transfiere al bloque catch coincidente.
* Cuando el control alcanza el punto final del bloque catch:
* Si la instrucción try tiene un bloque finally, se ejecuta dicho bloque finally.
* El control se transfiere al punto final de la instrucción try.
* Si se propaga una excepción a la instrucción try durante la ejecución del bloque catch:
* Si la instrucción try tiene un bloque finally, se ejecuta dicho bloque finally.
* La excepción se propaga a la siguiente instrucción envolvente try.
* Si la instrucción try no tiene cláusulas catch o si ninguna cláusula catch coincide con la excepción:
* Si la instrucción try tiene un bloque finally, se ejecuta dicho bloque finally.
* La excepción se propaga a la siguiente instrucción envolvente try.

Las instrucciones de un bloque finally siempre se ejecutan cuando el control sale de la instrucción try. Esto se cumple cuando el control se transfiere como resultado de una ejecución normal, de la ejecución de una instrucción break, continue, goto o return, o de la propagación de una excepción fuera de la instrucción try.

Si se vuelve a iniciar una excepción durante la ejecución de un bloque finally y no se captura dentro del mismo bloque, la excepción se propaga a la siguiente instrucción envolvente try. Si se estaba propagando otra excepción anterior, ésta se pierde. El proceso de propagación de una excepción se explica con mayor detalle en la descripción de la instrucción throw (§8.9.5).

El bloque try de una instrucción try es alcanzable si la instrucción try es alcanzable.

Un bloque catch de una instrucción try es alcanzable si la instrucción try es alcanzable.

El bloque finally de una instrucción try es alcanzable si la instrucción try es alcanzable.

El punto final de una instrucción try es alcanzable si se cumplen las dos condiciones siguientes:

* El punto final de un bloque try es alcanzable o el punto final de al menos uno de sus bloques catch es alcanzable.
* Si existe un bloque finally, el punto final del bloque finally es alcanzable.

## Instrucciones checked y unchecked

Las instrucciones checked y unchecked se utilizan con el fin de controlar el contexto de comprobación de desbordamiento para operaciones aritméticas y conversiones.

checked-statement:  
checked block

unchecked-statement:  
unchecked block

Con la instrucción checked, todas las expresiones del bloque (block) se evalúan en un contexto de comprobación y, con la instrucción unchecked, en un contexto donde no se realicen comprobaciones.

Las instrucciones checked y unchecked son equivalentes a los operadores checked y unchecked (§7.6.12), con la excepción de que operan con bloques en lugar de con expresiones.

## Instrucción lock

La instrucción lock bloquea un objeto mediante exclusión mutua, ejecuta una instrucción y, a continuación, libera el bloqueo.

lock-statement:  
lock ( expression ) embedded-statement

La expresión de una instrucción lock debe denotar un valor de un tipo que sea un tipo de referencia (reference-type). Nunca se realiza una conversión boxing implícita (§6.1.7) para la expresión de una instrucción lock y, por lo tanto, se produce un error en tiempo de compilación cuando la expresión denota un valor de un tipo de valor (value-type).

Una instrucción lock con la forma

lock (x) ...

donde x es una expresión de tipo de referencia (reference-type), es equivalente a

bool \_\_lockWasTaken = false;  
try {  
 System.Threading.Monitor.Enter(x, ref \_\_lockWasTaken);  
 ...  
}  
finally {  
 if (\_\_lockWasTaken) System.Threading.Monitor.Exit(x);  
}

salvo que x solo se evalúa una vez.

Mientras esté activo un bloqueo de exclusión mutua, el código que está ejecutándose en el mismo subproceso también puede obtener y liberar tal bloqueo. Sin embargo, el código que esté ejecutándose en otros subprocesos no puede recibir el bloqueo hasta que el éste se libere.

Se recomienda no bloquear objetos System.Type para sincronizar el acceso a datos estáticos. Se podría bloquear otro código en el mismo tipo, lo que produciría un interbloqueo. En su lugar, sincronice el acceso a datos estáticos bloqueando un objeto estático privado. Por ejemplo:

class Cache  
{  
 private static readonly object synchronizationObject = new object();

public static void Add(object x) {  
 lock (Cache.synchronizationObject) {  
 ...  
 }  
 }

public static void Remove(object x) {  
 lock (Cache.synchronizationObject) {  
 ...  
 }  
 }  
}

## Instrucción using

La instrucción using obtiene uno o varios recursos, ejecuta una instrucción y, a continuación, elimina el recurso.

using-statement:  
using ( resource-acquisition ) embedded-statement

resource-acquisition:  
local-variable-declaration  
expression

Un recurso es una clase o struct que implementa System.IDisposable, que incluye un único método sin parámetros denominado Dispose. El código que utiliza un recurso puede llamar a Dispose para indicar que ya no lo necesita. Si no llama a Dispose, se elimina temporalmente de forma automática como consecuencia de la recolección de elementos no utilizados.

Si la adquisición de recurso (resource-acquisition) se realiza mediante una declaración de variable local (local-variable-declaration), el tipo de esta última debe ser dynamic o un tipo que pueda ser convertido implícitamente en System.IDisposable. Si la adquisición de recurso (resource-acquisition) se realiza mediante una expresión (expression), esta expresión se debe poder convertir implícitamente en System.IDisposable.

Las variables locales que se declaran en la adquisición de recurso (resource-acquisition) son de solo lectura y deben incluir un inicializador. Se producirá un error en tiempo de compilación si la instrucción incrustada intenta modificar estas variables locales (por medio de asignación o utilizando los operadores ++ y ‑‑) o bien si obtiene su dirección o las pasa como parámetros ref u out.

Una instrucción using se traduce en tres partes: adquisición, uso y eliminación. El uso de un recurso está incluido implícitamente en una instrucción try que contiene una cláusula finally. La cláusula finally elimina el recurso. Si se obtiene un recurso null, no se llama a Dispose y no se inicia ninguna excepción. Si el recurso es de tipo dynamic se convierte dinámicamente mediante una conversión dinámica implícita (§6.1.8) en IDisposable durante la adquisición para asegurar que la conversión es correcta antes del uso y la eliminación.

Una instrucción using con la forma

using (ResourceType resource = expression) statement

corresponde a una de tres posibles expansiones. Cuando ResourceType es un tipo de valor que no acepta valores NULL, la expansión es:

{  
 ResourceType resource = expression;  
 try {  
 statement;  
 }  
 finally {  
 ((IDisposable)resource).Dispose();  
 }  
}

En caso contrario, cuando ResourceType es un tipo de valor que acepta valores NULL o un tipo de referencia diferente de dynamic, la expansión es:

{  
 ResourceType resource = expression;  
 try {  
 statement;  
 }  
 finally {  
 IDisposable d = (IDisposable)resource;  
 if (resource != null) d.Dispose();  
 }  
}

De lo contrario, cuando ResourceType es dynamic, la expansión es

{  
 ResourceType resource = expression;  
 IDisposable d = resource;  
 try {  
 statement;  
 }  
 finally {  
 if (d != null) d.Dispose();  
 }  
}

En cualquiera de las expansiones, la variable resource es de solo lectura en la instrucción incrustada y la variable d es inaccesible e invisible para la instrucción incrustada.

Se permite que una implementación implemente una instrucción using dada de manera diferente, por ejemplo, por motivos de rendimiento, siempre que el comportamiento sea coherente con la expansión anterior.

Una instrucción using con la forma

using (expression) statement

tiene las mismas tres posibles expansiones. En este caso ResourceType es implícitamente el tipo en tiempo de compilación de expression, si lo tiene. De lo contrario, la misma interfaz IDisposable se utiliza como ResourceType. La variable resource es inaccesible en la instrucción incrustada e invisible para esta.

Cuando una adquisición de recurso (resource-acquisition) toma la forma de una declaración de variable local (local-variable-declaration), es posible adquirir múltiples recursos de un tipo dado. Una instrucción using con la forma

using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement

es perfectamente equivalente a una secuencia de instrucciones using anidadas.

using (ResourceType r1 = e1)  
 using (ResourceType r2 = e2)  
 ...  
 using (ResourceType rN = eN)  
 statement

En el ejemplo siguiente se crea un archivo llamado log.txt y se escriben dos líneas de texto en él. Posteriormente, se abre el mismo archivo para leerlo y copiar las líneas de texto que contiene en la consola.

using System;  
using System.IO;

class Test  
{  
 static void Main() {  
 using (TextWriter w = File.CreateText("log.txt")) {  
 w.WriteLine("This is line one");  
 w.WriteLine("This is line two");  
 }

using (TextReader r = File.OpenText("log.txt")) {  
 string s;  
 while ((s = r.ReadLine()) != null) {  
 Console.WriteLine(s);  
 }

}  
 }  
}

Dado que las clases TextWriter y TextReader implementan la interfaz IDisposable, en el ejemplo se pueden utilizar instrucciones using para garantizar que el archivo subyacente esté correctamente cerrado después de las operaciones de lectura o escritura.

## La instrucción yield

En un bloque de iteradores (§8.2), la instrucción yield se utiliza para producir un valor para el objeto del enumerador (§10.14.4) o un objeto enumerable (§10.14.5) de un iterador o para señalar el final de la iteración.

yield-statement:  
yield return expression ;  
yield break ;

yield no es una palabra reservada; sólo adquiere un significado especial cuando se utiliza inmediatamente antes de la palabra clave return o break. En otros contextos, yield se puede utilizar como un identificador.

Existen varias restricciones que limitan los lugares de uso de la instrucción yield como se describe a continuación.

* La instrucción yield (en cualquiera de sus formas) generará un error en tiempo de compilación si aparece fuera de cuerpo del método (method-body), un cuerpo de operador (operator-body) o un cuerpo de descriptor de acceso (accessor-body).
* La instrucción yield (en cualquiera de sus formas) generará un error en tiempo de compilación si aparece dentro de una función anónima.
* La instrucción yield (en cualquiera de sus formas) generará un error en tiempo de compilación si aparece en la cláusula finally de una instrucción try.
* La instrucción yield return generará un error en tiempo de compilación si aparece en cualquier parte dentro de una instrucción try con cláusulas catch.

En el siguiente ejemplo se muestran algunos usos válidos y no válidos de instrucciones yield.

delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {  
 try {  
 yield return 1; // Ok  
 yield break; // Ok  
 }  
 finally {  
 yield return 2; // Error, yield in finally  
 yield break; // Error, yield in finally  
 }

try {  
 yield return 3; // Error, yield return in try...catch  
 yield break; // Ok  
 }  
 catch {  
 yield return 4; // Error, yield return in try...catch  
 yield break; // Ok  
 }

D d = delegate {   
 yield return 5; // Error, yield in an anonymous function  
 };   
}

int MyMethod() {  
 yield return 1; // Error, wrong return type for an iterator block  
}

Debe existir una conversión implícita (§6.1) desde el tipo de la expresión en la instrucción yield return al tipo yield (§10.14.3) del iterador.

Una instrucción yield return se ejecuta de la siguiente forma:

* La expresión de la instrucción se evalúa, se convierte en el tipo yield y se asigna a la propiedad Current del objeto del enumerador.
* Se suspende la ejecución del bloque de iteradores. Si la instrucción yield return se encuentra dentro de uno o más bloques try, los bloques finally asociados no se ejecutan en este momento.
* El método MoveNext del objeto del enumerador devuelve true a su llamador indicando que el objeto avanzó correctamente al siguiente valor.

La siguiente llamada al método MoveNext del objeto del enumerador reanuda la ejecución del bloque de iteradores desde donde se suspendió la última vez.

Una instrucción yield break se ejecuta de la siguiente forma:

* Si la instrucción yield break está encerrada entre uno o varios bloques try que llevan asociados bloques finally, el control se transfiere inicialmente al bloque finally de la instrucción try más interna. Cuando el control alcanza el punto final de un bloque finally, el control se transfiere al bloque finally de la siguiente instrucción envolvente try. Este proceso se repite hasta que se ejecuten los bloques finally de todas las instrucciones envolventes try que intervienen.
* El control vuelve al llamador del bloque de iteradores. Éste es el método MoveNext o el método Dispose del objeto enumerador.

Debido a que una instrucción yield break transfiere incondicionalmente el control a otra parte del código, el punto final de una instrucción yield break nunca es alcanzable.

# Espacios de nombres

Los programas de C# se organizan utilizando espacios de nombres. Los espacios de nombres se utilizan, tanto como sistema de organización “interna” de un programa, como sistema de organización “externa”; es una forma de presentar los elementos de un programa que están expuestos a otros programas.

Las directivas using (§9.4) se proporcionan para facilitar el uso de los espacios de nombres.

## Unidades de compilación

Una unidad de compilación (compilation-unit) define la estructura completa de un archivo de código fuente. Consiste en cero o varias directivas using (using-directives), seguidas de cero o varios atributos globales (global-attributes), seguidos de cero o varias declaraciones de miembros de espacios de nombres (namespace-member-declaration).

compilation-unit:  
extern-alias-directivesopt using-directivesopt global-attributesopt  
 namespace-member-declarationsopt

Un programa de C# consiste en una o varias unidades de compilación, cada una de ellas en un archivo de código fuente independiente. Cuando se compila un programa de C#, todas las unidades de compilación se procesan conjuntamente. Por lo tanto, las unidades de compilación pueden depender unas de otras, posiblemente de forma circular.

Las directivas using (using-directives) de una unidad de compilación afectan a sus atributos globales (global-attributes) y declaraciones de miembros de espacios de nombres (namespace-member-declarations), pero no al resto de unidades de compilación.

Los atributos globales (global-attributes) (§17) de una unidad de compilación permiten especificar los atributos del módulo y del ensamblado de destino. Los ensamblados y los módulos actúan como contenedores físicos de tipos. Un ensamblado puede estar formado por varios módulos separados físicamente.

Las declaraciones de miembros de espacios de nombres (namespace-member-declarations) de cada unidad de compilación de un programa contribuyen con miembros a un único espacio de declaración, denominado espacio de nombres global. Por ejemplo:

Archivo A.cs:

class A {}

Archivo B.cs:

class B {}

Las dos unidades de compilación contribuyen al espacio global de nombres, en este caso, declarando dos clases con los nombres completos A y B. Como las dos unidades de compilación contribuyen al mismo espacio de declaración, si cada una contiene una declaración de miembro con el mismo nombre se producirá un error.

## Declaraciones de espacio de nombres

Una declaración de espacio de nombres (namespace-declaration) consiste en la palabra clave namespace seguida de un nombre y de un cuerpo y, opcionalmente, de un punto y coma.

namespace-declaration:  
namespace qualified-identifier namespace-body ;opt

qualified-identifier:  
identifier  
qualified-identifier . identifier

namespace-body:  
{ extern-alias-directivesopt using-directivesopt namespace-member-declarationsopt }

Una declaración de espacio de nombres (namespace-declaration) puede ocurrir como declaración de alto nivel de una unidad de compilación (compilation-unit) o como declaración de miembro dentro de otra declaración de espacio de nombres. Cuando una declaración de espacio de nombres (namespace-declaration) ocurre como declaración de alto nivel de una unidad de compilación (compilation-unit), el espacio de nombres se convierte en miembro del espacio global de nombres. Cuando una declaración de espacio de nombres (namespace-declaration) ocurre dentro de otra declaración de espacio de nombres, el espacio de nombres más interno se convierte en miembro del espacio de nombres externo. En ambos casos, el nombre del espacio de nombres debe ser único dentro del espacio de nombres que lo contiene.

Los espacios de nombres son public implícitamente y la declaración de un espacio de nombres no puede incluir modificadores de acceso.

Dentro del cuerpo de espacio de nombres (namespace-body), las directivas using (using-directives) opcionales importan los nombres de otros espacios de nombres y tipos, permitiendo que se haga referencia a los mismos directamente y no a través de nombres completos. Las declaraciones de miembros de espacios de nombres (namespace-member-declarations) opcionales contribuyen con miembros al espacio de declaración del espacio de nombres. Observe que todas las directivas using (using-directives) deben aparecer antes de cualquier declaración de miembro.

El identificador calificado (qualified-identifier) de una declaración de espacio de nombres (namespace-declaration) puede ser un único identificador o una secuencia de identificadores separados por tokens “.”. Esta última forma permite que un programa defina un espacio de nombres anidado sin anidar léxicamente varias declaraciones de espacio de nombres. Por ejemplo,

namespace N1.N2  
{  
 class A {}

class B {}  
}

equivale semánticamente a

namespace N1  
{  
 namespace N2  
 {  
 class A {}

class B {}  
 }  
}

Los espacios de nombres tienen extremo abierto, y dos declaraciones de espacio de nombres con el mismo nombre completo contribuyen al mismo espacio de declaración (§3.3). En el ejemplo

namespace N1.N2  
{  
 class A {}  
}

namespace N1.N2  
{  
 class B {}  
}

las dos declaraciones de espacios de nombres anteriores contribuyen al mismo espacio de declaración, en este caso declarando dos clases con los nombres completos N1.N2.A y N1.N2.B. Como las dos declaraciones contribuyen al mismo espacio de declaración, si cada una contiene una declaración de miembro con el mismo nombre se producirá un error.

## Alias extern

Una directiva de alias extern (extern-alias-directive) introduce un identificador que se utiliza como un alias para un espacio de nombres. La especificación del espacio de nombres con alias es externa al código fuente del programa y se aplica también a los espacios de nombres anidados del espacio de nombres con alias.

extern-alias-directives:  
extern-alias-directive  
extern-alias-directives extern-alias-directive

extern-alias-directive:  
extern alias identifier ;

El ámbito de una directiva de alias extern (extern-alias-directive) se extiende sobre las directivas using (using-directives), los atributos globales (global-attributes) y las declaraciones de miembros de espacios de nombres (namespace-member-declarations) de la unidad de compilación o del cuerpo de espacio de nombres inmediato.

Dentro de una unidad de compilación o del cuerpo de espacio de nombres que contiene una directiva de alias extern (extern-alias-directive), el identificador introducido por la directiva de alias extern (extern-alias-directive) se puede utilizar para hacer referencia al espacios de nombres con alias. Se genera un error en tiempo de compilación si el identificador (identifier) es la palabra global.

Una directiva de alias extern (extern-alias-directive) ofrece un alias dentro de una determinada unidad de compilación o cuerpo de espacio de nombres, pero no contribuye con ningún miembro nuevo al espacio de la declaración subyacente. En otras palabras, una directiva de alias extern (extern-alias-directive) no es transitiva, sino que solo afecta a la unidad de compilación o al cuerpo de espacio de nombres en el que se da.

El siguiente programa declara y utiliza dos alias externos, X e Y, cada uno de los cuales representa la ruta de una jerarquía de espacios de nombres diferente.

extern alias X;  
extern alias Y;

class Test  
{  
 X::N.A a;  
 X::N.B b1;  
 Y::N.B b2;  
 Y::N.C c;  
}

El programa declara la existencia de los alias externos X e Y, pero las definiciones reales de los alias son externas al programa. Ahora se puede hacer referencia a las clases N.B denominadas de la misma manera como X.N.B e Y.N.B, o bien utilizando el calificador de alias de espacios de nombres, X::N.B e Y::N.B. Se genera un error si un programa declara un alias extern para el que no se proporciona ninguna definición externa.

## Directivas Using

Las directivas using facilitan el uso de espacios de nombres y tipos definidos en otros espacios de nombres. Las directivas using tienen un impacto en el proceso de resolución de nombres de espacios de nombres o nombres de tipo (namespace-or-type-name) (§3.8) y de nombres simples (simple-name) (§7.6.2), pero, exceptuando las declaraciones, las directivas using no contribuyen con nuevos miembros a los espacios de declaración subyacentes de las unidades de compilación o a los espacios de nombres dentro de los que se utilizan.

using-directives:  
using-directive  
using-directives using-directive

using-directive:  
using-alias-directive  
using-namespace-directive

Una directiva de alias using (using-alias-directive) (§9.4.1) define un alias para un espacio de nombres o un tipo.

Una directiva de espacio de nombres using (using-namespace-directive) (§9.4.2) importa los miembros de tipo de un espacio de nombres.

El ámbito de una directiva using (using-directive) se extiende a lo largo de las declaraciones de miembros de espacios de nombres (namespace-member-declarations) de la unidad de compilación o del cuerpo de espacio de nombres que la contiene inmediatamente. El ámbito de una directiva using (using-directive) no incluye específicamente sus directivas using del mismo nivel. Por consiguiente, las directivas using (using-directive) no se afectan entre sí, y el orden en el que se escriben no tiene importancia.

### Directivas de alias using

Una directiva de alias using (using-alias-directive) define un identificador que funciona como un alias para un espacio de nombres o un tipo dentro de la unidad de compilación o cuerpo de espacio de nombres inmediatamente envolvente.

using-alias-directive:  
using identifier = namespace-or-type-name ;

Dentro de las declaraciones de miembros de una unidad de compilación o del cuerpo de un espacio de nombres que contiene una directiva de alias using (using-alias-directive), se puede utilizar el identificador definido por la directiva de alias using (using-alias-directive) para hacer referencia al espacio de nombres o al tipo. Por ejemplo:

namespace N1.N2  
{  
 class A {}  
}

namespace N3  
{  
 using A = N1.N2.A;

class B: A {}  
}

Dentro de las declaraciones de miembros en el espacio de nombres N3, A es un alias para N1.N2.A y, por tanto, la clase N3.B se deriva de la clase N1.N2.A. Se puede obtener el mismo efecto mediante la creación de un alias R para N1.N2 y, a continuación, haciendo referencia a R.A:

namespace N3  
{  
 using R = N1.N2;

class B: R.A {}  
}

El identificador (identifier) de una directiva de alias using (using-alias-directive) debe ser único dentro del espacio de declaraciones de la unidad de compilación o espacio de nombres que contiene inmediatamente a dicha directiva de alias using. Por ejemplo:

namespace N3  
{  
 class A {}  
}

namespace N3  
{  
 using A = N1.N2.A; // Error, A already exists  
}

En el ejemplo anterior, N3 ya contiene un miembro A, así que es un error que una directiva de alias using (using-alias-directive) utilice ese identificador. También es un error que dos o más directivas de alias using (using-alias-directive) de una misma unidad de compilación o cuerpo de espacio de nombres declaren un alias con el mismo nombre.

Una directiva de alias using (using-alias-directive) ofrece un alias dentro de una determinada unidad de compilación o cuerpo de espacio de nombres, pero no contribuye con ningún miembro nuevo al espacio de declaración subyacente. En otras palabras, una directiva de alias using (using-alias-directive) no es transitiva, sino que solo afecta a la unidad de compilación o cuerpo de espacio de nombres en donde ocurre. En el ejemplo

namespace N3  
{  
 using R = N1.N2;  
}

namespace N3  
{  
 class B: R.A {} // Error, R unknown  
}

el ámbito de la directiva de alias using (using-alias-directive) que define R solo se extiende a las declaraciones de los miembros en el cuerpo de espacio de nombres que lo contiene y, por lo tanto, R es desconocido en la declaración del segundo espacio de nombres. Sin embargo, si se sitúa la directiva de alias using (using-alias-directive) en la unidad de compilación que la contiene, el alias estará disponible para ambas declaraciones de espacios de nombres:

using R = N1.N2;

namespace N3  
{  
 class B: R.A {}  
}

namespace N3  
{  
 class C: R.A {}  
}

Al igual que con los miembros normales, los nombres definidos por las directivas de alias using (using-alias-directive) están ocultos por miembros nombrados de forma similar en ámbitos anidados. En el ejemplo

using R = N1.N2;

namespace N3  
{  
 class R {}

class B: R.A {} // Error, R has no member A  
}

la referencia a R.A en la declaración de B produce un error en tiempo de compilación porque R hace referencia a N3.R, no a N1.N2.

El orden en el que se escriben las directivas de alias using (using-alias-directive) no tiene importancia y la resolución del espacio de nombres o nombre de tipo (namespace-or-type-name) a los que hace referencia una directiva de alias using (using-alias-directive) tampoco se ve afectado por la misma directiva de alias using ni por otras directivas using (using-directive) de la unidad de compilación o cuerpo de espacio de nombres que la contiene inmediatamente. En otras palabras, el espacio de nombres o nombre de tipo (namespace-or-type-name) de una directiva de alias using (using-alias-directive) se resuelve como si la unidad de compilación o el cuerpo de espacio de nombres que la contiene inmediatamente no tuviera ninguna directiva using (using-directive). Sin embargo, una directiva de alias using (using-alias-directive) puede verse afectada por las directivas de alias extern (extern-alias-directives) de la unidad de compilación o el cuerpo de espacio de nombres que la contiene inmediatamente. En el ejemplo

namespace N1.N2 {}

namespace N3  
{  
 extern alias E;

using R1 = E.N; // OK

using R2 = N1; // OK

using R3 = N1.N2; // OK

using R4 = R2.N2; // Error, R2 unknown  
}

la última directiva de alias using (using-alias-directive) da como resultado un error en tiempo de compilación, ya que no se ve afectada por la primera directiva de alias using. La primera directiva de alias using (using-alias-directive) no da como resultado un error puesto que el ámbito del alias externo E incluye la directiva de alias using.

Una directiva de alias using (using-alias-directive) puede crear un alias para cualquier espacio de nombres o tipo, incluyendo el espacio de nombres dentro del que aparece y cualquier espacio de nombres o tipo anidado dentro de ese espacio de nombres.

El acceso a un espacio de nombres o a un tipo mediante un alias da exactamente el mismo resultado que el acceso a ese espacio de nombres o tipo mediante su nombre declarado. Por ejemplo

namespace N1.N2  
{  
 class A {}  
}

namespace N3  
{  
 using R1 = N1;  
 using R2 = N1.N2;

class B  
 {  
 N1.N2.A a; // refers to N1.N2.A  
 R1.N2.A b; // refers to N1.N2.A  
 R2.A c; // refers to N1.N2.A  
 }  
}

los nombres N1.N2.A, R1.N2.A y R2.A son equivalentes y todos se refieren a la clase cuyo nombre completo es N1.N2.A.

Los alias using pueden nombrar un tipo construido cerrado, pero no pueden nombrar una declaración de tipo genérico sin enlazar si no proporcionan argumentos de tipo. Por ejemplo:

namespace N1  
{  
 class A<T>  
 {  
 class B {}  
 }  
}

namespace N2  
{  
 using W = N1.A; // Error, cannot name unbound generic type

using X = N1.A.B; // Error, cannot name unbound generic type

using Y = N1.A<int>; // Ok, can name closed constructed type

using Z<T> = N1.A<T>; // Error, using alias cannot have type parameters  
}

### Directivas Using de espacio de nombres

Una directiva de espacio de nombres using (using-namespace-directive) importa los tipos que contiene un espacio de nombres en la unidad de compilación o cuerpo de espacio de nombres inmediatamente envolvente, permitiendo el uso del identificador sin calificar de cada tipo.

using-namespace-directive:  
using namespace-name ;

Dentro de las declaraciones de miembros de una unidad de compilación o del cuerpo de un espacio de nombres que contiene una directiva de espacio de nombres using (using-namespace-directive), se puede hacer referencia directamente a los tipos que contienen ese espacio de nombres. Por ejemplo:

namespace N1.N2  
{  
 class A {}  
}

namespace N3  
{  
 using N1.N2;

class B: A {}  
}

En el anterior ejemplo, dentro de las declaraciones de miembros en el espacio de nombres N3, los miembros de tipo N1.N2 están directamente disponibles y, por tanto, la clase N3.B se deriva de la clase N1.N2.A.

Una directiva de espacio de nombres using (using-namespace-directive) importa los tipos que contiene un determinado espacio de nombres, pero no importa específicamente los espacios de nombres anidados. En el ejemplo

namespace N1.N2  
{  
 class A {}  
}

namespace N3  
{  
 using N1;

class B: N2.A {} // Error, N2 unknown  
}

la directiva de espacio de nombres using (using-namespace-directive) importa los tipos que contiene N1, pero no los espacios de nombres anidados en N1. Por ello, la referencia a N2.A en la declaración de B produce un error en tiempo de compilación porque no existen miembros con el nombre N2 en el ámbito.

A diferencia de una directiva de alias using (using-alias-directive), una directiva de espacio de nombres using (using-namespace-directive) puede importar tipos cuyos identificadores ya están definidos dentro de la unidad de compilación o cuerpo de espacio de nombres envolvente. En realidad, los nombres importados por una directiva de espacio de nombres using (using-namespace-directive) están ocultos por miembros nombrados de forma similar en la unidad de compilación o cuerpo de espacio de nombres que la contiene. Por ejemplo:

namespace N1.N2  
{  
 class A {}

class B {}  
}

namespace N3  
{  
 using N1.N2;

class A {}  
}

Aquí, dentro de las declaraciones de miembros en el espacio de nombres N3, A hace referencia a N3.A en lugar de a N1.N2.A.

Cuando más de un espacio de nombres importados mediante directivas de espacio de nombres using (using-namespace-directive) de la misma unidad de compilación o cuerpo de espacio de nombres contiene tipos para el mismo nombre, las referencias a dicho nombre se consideran ambiguas. En el ejemplo

namespace N1  
{  
 class A {}  
}

namespace N2  
{  
 class A {}  
}

namespace N3  
{  
 using N1;

using N2;

class B: A {} // Error, A is ambiguous  
}

tanto N1 como N2 contienen un miembro A. Como N3 importa ambas, se produce un error en tiempo de compilación al hacer referencia a A en N3. En este caso, el conflicto se puede resolver mediante la calificación de referencias a A o mediante la introducción de una directiva de alias using (using-alias-directive) que elige un A determinado. Por ejemplo:

namespace N3  
{  
 using N1;

using N2;

using A = N1.A;

class B: A {} // A means N1.A  
}

Al igual que una directiva de alias using (using-alias-directive), una directiva de espacio de nombres using (using-namespace-directive) no contribuye con ningún miembro nuevo al espacio de declaración subyacente de la unidad de compilación o espacio de nombres, sino que afecta sólo a la unidad de compilación o cuerpo de espacio de nombres donde aparece.

El nombre de espacio de nombres (namespace-name) al que hace referencia una directiva de espacio de nombres using (using-namespace-directive) se resuelve de la misma forma que el espacio de nombres o nombre de tipo (namespace-or-type-name) al que hace referencia una directiva de alias using (using-alias-directive). Por lo tanto, las directivas de espacio de nombres using (using-namespace-directives) de una misma unidad de compilación o cuerpo de espacio de nombres no se afectan unas a otras y pueden escribirse en cualquier orden.

## Miembros de espacio de nombres

Una declaración de miembro de espacio de nombres (namespace-member-declaration) puede ser una declaración de espacio de nombres (namespace-declaration) (§9.2) o una declaración de tipo (type-declaration) (§9.6).

namespace-member-declarations:  
namespace-member-declaration  
namespace-member-declarations namespace-member-declaration

namespace-member-declaration:  
namespace-declaration  
type-declaration

Una unidad de compilación o un cuerpo de espacio de nombres puede contener declaraciones de miembros de espacios de nombres (namespace-member-declarations), y dichas declaraciones contribuyen con nuevos miembros al espacio de declaración subyacente de la unidad de compilación o cuerpo de espacio de nombres que las contienen.

## Declaraciones de tipo

Una declaración de tipo (type-declaration) es una declaración de clase (class-declaration) (§10.1), una declaración de struct (struct-declaration) (§11.1), una declaración de interfaz (interface-declaration) (§13.1), una declaración de enumeración (enum-declaration) (§14.1) o una declaración de delegado (delegate-declaration) (§15.1).

type-declaration:  
class-declaration  
struct-declaration  
interface-declaration  
enum-declaration  
delegate-declaration

Una declaración de tipo (type-declaration) puede ocurrir como declaración de alto nivel en una unidad de compilación o como declaración de miembro dentro de un espacio de nombres, de una clase o de un struct.

Cuando ocurre una declaración de tipo para un tipo T como declaración de alto nivel en una unidad de compilación, el nombre completo del nuevo tipo declarado es simplemente T. Cuando una declaración de tipo para un tipo T se ejecuta dentro de un espacio de nombres, de una clase o de un struct, el nombre completo del nuevo tipo declarado es N.T, donde N es el nombre completo del espacio de nombres, de la clase o del struct que lo contiene.

Un tipo declarado dentro de una clase o de un struct se denomina tipo anidado (§10.3.8).

Los modificadores de acceso permitidos y el acceso predeterminado para una declaración de tipo dependen del contexto en el que tenga lugar la declaración (§3.5.1):

* Los tipos declarados en las unidades de compilación o espacios de nombres pueden tener acceso public o internal. El acceso predeterminado es internal.
* Los tipos declarados en clases pueden tener acceso public, protected internal, protected, internal o private. El acceso predeterminado es private.
* Los tipos declarados en structs pueden tener acceso public, internal o private. El acceso predeterminado es private.

## Calificadores de alias de espacios de nombres

El calificador de alias de espacios de nombres :: garantiza que las búsquedas de nombres de tipos no se vean afectadas por la introducción de nuevos tipos y miembros. El calificador de alias de espacios de nombres siempre aparece entre dos identificadores a los que se hace referencia como los identificadores izquierdo y derecho. Al contrario de lo que ocurre con el calificador habitual ., el identificador izquierdo del calificador :: solo se busca como un alias using o extern.

Un miembro de alias calificado (qualified-alias-member) se define de la siguiente manera:

qualified-alias-member:  
identifier :: identifier type-argument-listopt

Se puede utilizar un miembro de alias calificado (qualified-alias-member) como un nombre de espacio de nombres o de tipo (namespace-or-type-name) (§3.8) o como el operando izquierdo en un acceso a miembro (member-access) (§7.6.4).

Un miembro de alias calificado (qualified-alias-member) adquiere una de las estructuras siguientes:

* N::I<A1, ..., AK>, donde N y I representan identificadores, y <A1, ..., AK> es una lista de argumentos de tipo. (K siempre es al menos uno).
* N::I, donde N e I representan identificadores. (En este caso, se considera que K es cero).

Con esta notación, el significado de un miembro de alias calificado (qualified-alias-member) se determina de la siguiente manera:

* Si N es el identificador global, el espacio de nombres global se busca por I:
* Si el espacio de nombres global contiene un espacio de nombres denominado I y K es cero, el miembro de alias calificado (qualified-alias-member) hace referencia a dicho espacio de nombres.
* De lo contrario, si el espacio de nombres global contiene un tipo genérico denominado I y K es cero, el miembro de alias calificado (qualified-alias-member) hace referencia a dicho tipo.
* De lo contrario, si el espacio de nombres global contiene un tipo denominado I con parámetros de tipo K, el miembro de alias calificado (qualified-alias-member) hace referencia a dicho tipo construido con los argumentos de tipo dados.
* De lo contrario, el miembro de alias calificado (qualified-alias-member) no está definido y se genera un error en tiempo de compilación.
* De lo contrario, comenzando por la siguiente declaración de espacio de nombres (§9.2) que contiene el miembro de alias calificado (qualified-alias-member) (si lo hay), continuando por cada una de las declaraciones de espacio de nombres envolventes (si las hay) y terminando por la unidad de compilación que contiene el miembro de alias calificado (qualified-alias-member), se evalúan los siguientes pasos hasta que se detecta una entidad:
* Si la declaración de espacio de nombres o la unidad de compilación contiene una directiva de alias using (using-alias-directive) que asocia N con un tipo, el miembro de alias calificado (qualified-alias-member) no se define y se genera un error en tiempo de compilación.
* De lo contrario, si la declaración de espacio de nombres o la unidad de compilación contiene una directiva de alias extern (extern-alias-directive) o una directiva de alias using (using-alias-directive) que asocia N con un espacio de nombres:
* Si el espacio de nombres asociado con N contiene un espacio de nombres denominado I y K es cero, el miembro de alias calificado (qualified-alias-member) hace referencia a dicho espacio de nombres.
* De lo contrario, si el espacio de nombres asociado con N contiene un tipo no genérico denominado I y K es cero, el miembro de alias calificado (qualified-alias-member) hace referencia a dicho tipo.
* De lo contrario, si el espacio de nombres asociado con N contiene un tipo denominado I con parámetros de tipo K, el miembro de alias calificado (qualified-alias-member) hace referencia a dicho tipo construido con los argumentos de tipo dados.
* De lo contrario, el miembro de alias calificado (qualified-alias-member) no está definido y se genera un error en tiempo de compilación.
* De lo contrario, el miembro de alias calificado (qualified-alias-member) no está definido y se genera un error en tiempo de compilación.

Tenga en cuenta que si utiliza el calificador de alias de espacios de nombres con un alias que haga referencia a un tipo se generará un error en tiempo de compilación. Tenga en cuenta también que si el identificador N es global, se realizará una búsqueda en el espacio de nombres global, incluso si hay un alias using que asocia global con un tipo o un espacio de nombres.

### Unicidad de los alias

Cada unidad de compilación y cada cuerpo de espacio de nombres tiene un espacio de declaración independiente para alias extern y using. Por lo tanto, mientras que el nombre de un alias extern o un alias using debe ser único dentro del conjunto de alias extern y using declarados en la unidad de compilación o cuerpo de espacio de nombres que lo contiene inmediatamente, un alias puede tener el mismo nombre que un tipo o un espacio de nombres siempre que se utilice con el calificador ::.

En el ejemplo

namespace N  
{  
 public class A {}

public class B {}  
}

namespace N  
{  
 using A = System.IO;

class X  
 {  
 A.Stream s1; // Error, A is ambiguous

A::Stream s2; // Ok  
 }  
}

el nombre A tiene dos posibles significados en el segundo cuerpo de espacio de nombres, puesto que tanto la clase A como el alias using A están dentro del ámbito. Por este motivo, el uso de A en el nombre completo A.Stream es ambiguo y causa un error en tiempo de compilación. Sin embargo, el uso de A con el calificador :: no es un error porque A solo se busca como un alias de espacio de nombres.

# Clases

Una clase es una estructura de datos que puede contener miembros de datos (constantes y campos), miembros de función (métodos, propiedades, indizadores, eventos, operadores, constructores de instancia, constructores estáticos y destructores) y tipos anidados. Los tipos de clase admiten la herencia, un mecanismo mediante el cual una clase derivada puede extender y especializar una clase base.

## Declaraciones de clases

Una declaración de clase (class-declaration) es una declaración de tipo (type-declaration) (§9.6) que declara una clase nueva.

class-declaration:  
attributesopt class-modifiersopt partialopt class identifier type-parameter-listopt  
 class-baseopt type-parameter-constraints-clausesopt class-body ;opt

Una declaración de clase (class-declaration) consiste en un conjunto de atributos (attributes) (§17) opcionales, seguido de un conjunto de modificadores de clase (class-modifiers) (§10.1.1), seguido de un modificador partial opcional, seguido de la palabra clave class y un identificador (identifier) que da nombre a la clase, seguido de una especificación opcional de lista de parámetros de tipo (type-parameter-list) (§10.1.3), seguido de una especificación opcional de clase base (class-base) (§10.1.4), seguida de una especificación opcional de cláusulas de restricciones de parámetros de tipo (type-parameter-constraints-clauses) (§10.1.5), seguida de un cuerpo de clase (class-body) (§10.1.6), seguido de un punto y coma opcional.

Una declaración de clase no puede proporcionar cláusulas de restricciones de parámetros de tipo (type-parameter-constraints-clauses) a menos que también proporcione listas de parámetros de tipo (type-parameter-list).

Una declaración de clase que proporciona una lista de parámetros de tipo (type-parameter-list) es una declaración de clase genérica. De manera adicional, cualquier clase anidada dentro de una declaración de clase genérica o una declaración de struct genérica es en sí misma una declaración de clase genérica puesto que se deben proporcionar los parámetros de tipo para el tipo contenedor y así crear un tipo construido.

### Modificadores de clase

Una declaración de clase (class-declaration) puede incluir opcionalmente una secuencia de modificadores de clase:

class-modifiers:  
class-modifier  
class-modifiers class-modifier

class-modifier:  
new  
public  
protected  
internal  
private  
abstract  
sealed  
static

Cuando el mismo modificador aparece varias veces en una declaración de clase, se produce un error en tiempo de compilación.

El modificador new está permitido en clases anidadas. Especifica que la clase oculta un miembro heredado con el mismo nombre, como se describe en §10.3.4. Se produce un error en tiempo de compilación si el modificador new aparece en una declaración de clase no anidada.

Los modificadores public, protected, internal y private controlan la accesibilidad de la clase. Dependiendo del contexto en el que ocurra la declaración de clase, algunos de estos modificadores pueden no estar permitidos (§3.5.1).

Los modificadores abstract, sealed y static se explican en las siguientes secciones.

#### Clases abstractas

El modificador abstract se utiliza para indicar que una clase está incompleta y que sólo se va a utilizar como clase base. Una clase abstracta se diferencia de una clase no abstracta en lo siguiente:

* No se puede crear una instancia de una clase abstracta directamente, y es un error en tiempo de compilación utilizar el operador new en una clase abstracta. Aunque es posible tener variables y valores cuyos tipos en tiempo de compilación sean abstractos, tales variables y valores serán null o contendrán referencias a instancias de clases no abstractas derivadas de los tipos abstractos.
* Se permite que una clase abstracta contenga miembros abstractos, aunque no es necesario.
* No se puede sellar una clase abstracta.

Cuando una clase no abstracta se deriva de una clase abstracta, la clase no abstracta debe incluir implementaciones reales de todos los miembros abstractos heredados; por lo tanto, invalida estos miembros abstractos. En el ejemplo

abstract class A  
{  
 public abstract void F();  
}

abstract class B: A  
{  
 public void G() {}  
}

class C: B  
{  
 public override void F() {  
 // actual implementation of F  
 }  
}

la clase abstracta A define el método abstracto F. La clase B define un método adicional G, pero no proporciona una implementación de F, por lo que B también debe ser declarada como abstracta. La clase C invalida F y proporciona una implementación real. Dado que no hay miembros abstractos en C, está permitido que C sea no abstracta, aunque no es necesario.

#### Clases sealed

El modificador sealed se utiliza para impedir la derivación de una clase. Si se especifica que una clase sealed es la clase base de otra clase, se produce un error en tiempo de compilación.

Una clase sealed no puede ser tampoco una clase abstracta.

El modificador sealed se utiliza principalmente para impedir la derivación no intencionada, pero también permite algunas optimizaciones en tiempo de ejecución. En particular, como una clase sealed no puede tener clases derivadas, es posible transformar las llamadas virtuales a miembros de función en instancias de clase sealed en llamadas no virtuales.

#### Clases estáticas

El modificador static se usa para marcar la clase que se va a declarar como clase estática. No se puede crear una instancia de una clase estática, y una clase estática no se puede utilizar como tipo ni contener sólo miembros estáticos. Solo una clase estática puede contener declaraciones de métodos de extensión (§10.6.9).

Una declaración de clase estática está sujeta a las siguientes restricciones:

* Es posible que una clase estática no incluya un modificador sealed o abstract. Sin embargo, tenga en cuenta que puesto que no se pueden crear instancias de una clase estática ni ésta se puede derivar, dicha clase se comporta como si fuera sealed y abstract.
* Es posible que una clase estática no incluya una especificación de clase base (class-base) (§10.1.4) y no puede especificar de manera explícita una clase base ni una lista de interfaces implementadas. Una clase estática hereda de manera implícita del tipo object.
* Una clase estática sólo puede contener miembros estáticos (§10.3.7). Tenga en cuenta que las constantes y los tipos anidados se clasifican como miembros estáticos.
* Una clase estática no puede tener miembros con accesibilidad declarada protected o protected internal.

El incumplimiento de cualquiera de estas restricciones genera un error en tiempo de compilación.

Una clase estática no tiene constructores de instancia. No es posible declarar un constructor de instancia en una clase estática, y no se proporciona ningún constructor de instancia predeterminado (§10.11.4) para una clase estática.

Los miembros de una clase estática no son automáticamente estáticos y las declaraciones de miembros deben incluir de manera explícita un modificador static (excepto para las constantes y tipos anidados). Cuando una clase se anida dentro de una clase exterior estática, la clase anidada no es estática a menos que incluya explícitamente un modificador static.

##### Referencia a tipos de clases estáticas

Un nombre de espacio de nombres o de tipo (namespace-or-type-name) (§3.8) puede hacer referencia a una clase estática si

* El nombre de espacio de nombres o de tipo (namespace-or-type-name) es T en un nombre de espacio de nombres o de tipo (namespace-or-type-name) con el formato T.I, o bien
* El nombre de espacio de nombres o de tipo (namespace-or-type-name) es T en una expresión typeof (typeof-expression) (§7.5.11) con el formato typeof(T).

Una expresión primaria (primary-expression) (§7.5) puede hacer referencia a una clase estática si

* La expresión primaria (primary-expression) es E en un acceso a miembros (member-access) (§7.5.4) con la forma E.I.

En otros contextos, se genera un error en tiempo de compilación si se hace referencia a una clase estática. Por ejemplo, se genera un error si se utiliza una clase estática como una clase base, un tipo constituyente (§10.3.8) de un miembro, un argumento de tipo genérico o una restricción de parámetro de tipo. De manera similar, una clase estática no se puede utilizar en un tipo de matriz, un tipo de puntero, una expresión new, una expresión de conversión de tipos, una expresión is, una expresión as, una expresión sizeof o una expresión de valor predeterminado.

### Modificador parcial

El modificador partial se usa para indicar que esta declaración de clase (class-declaration) es una declaración de tipo parcial. Varias declaraciones de tipo parciales con el mismo nombre dentro de un espacio de nombres envolvente o una declaración de tipo se combinan para formar una declaración de tipo, siguiendo las reglas especificadas en la sección §10.2.

Hacer que la declaración de una clase se distribuya en segmentos separados del texto de un programa puede ser útil si estos segmentos se producen o mantienen en contextos diferentes. Por ejemplo, una parte de una declaración de clase puede ser generada automáticamente, mientras que la otra se ha creado de manera manual. La separación textual de las dos impide que las actualizaciones de una entren en conflicto con las actualizaciones de la otra.

### Parámetros de tipo

Un parámetro de tipo es un identificador sencillo que denota un marcador de posición para un argumento de tipo proporcionado para crear un tipo construido. Un parámetro de tipo es un marcador de posición formal para un tipo que se proporcionará posteriormente. Por el contrario, un argumento de tipo (§4.4.1) es el tipo real que se sustituye por el parámetro de tipo al crear un tipo construido.

type-parameter-list:  
< type-parameters >

type-parameters:  
attributesopt type-parameter  
type-parameters , attributesopt type-parameter

type-parameter:  
identifier

Cada parámetro de tipo de una declaración de clase define un nombre en el espacio de declaración (§3.3) de dicha clase. Por lo tanto, no puede tener el mismo nombre que otro parámetro de tipo o que un miembro declarado en dicha clase. Un parámetro de tipo no puede tener el mismo nombre que el tipo.

### Especificación de clase base

Una declaración de clase puede incluir una especificación de clase base (class-base), que define la clase base directa de la clase y las interfaces (§13) directamente implementadas por la clase.

class-base:  
: class-type  
: interface-type-list  
: class-type , interface-type-list

interface-type-list:  
interface-type  
interface-type-list , interface-type

La clase base especificada en una declaración de clase se puede ser un tipo de clase construido (§4.4). Una clase base no puede ser un parámetro de tipo en sí mismo, aunque puede implicar a parámetros de tipo que se encuentran dentro del ámbito.

class Extend<V>: V {} // Error, type parameter used as base class

#### Clases base

Cuando se incluye un tipo de clase (class-type) en la clase base (class-base), este especifica la clase base directa de la clase que se declara. Si la declaración de una clase no tiene clase base (class-base) o esta solo proporciona una lista de tipos de interfaz, se supone que la clase base directa es object. Una clase hereda los miembros de su clase base directa, como se describe en §10.3.3.

En el ejemplo

class A {}

class B: A {}

se dice que la clase A es la clase base directa de B y que B se deriva de A. Dado que A no especifica explícitamente una clase base directa, su clase base directa es implícitamente object.

Para un tipo de clase construida, si una clase base se especifica en la declaración de clase genérica, la clase base del tipo construido se obtiene sustituyendo, para cada parámetro de tipo (type-parameter) de la declaración de clase base, el correspondiente argumento de tipo (type-argument) del tipo construido. Dada las declaraciones de clase genérica

class B<U,V> {...}

class G<T>: B<string,T[]> {...}

la clase base del tipo construido G<int> sería B<string,int[]>.

La clase base directa de un tipo de clase debe ser al menos tan accesible como el propio tipo de clase (§3.5.2). Por ejemplo, se produce un error en tiempo de compilación cuando una clase public se deriva de una clase private o internal.

La clase base directa de un tipo de clase no debe ser de ninguno de los tipos siguientes: System.Array, System.Delegate, System.MulticastDelegate, System.Enum o System.ValueType. Es más, una declaración de clase genérica no puede utilizar System.Attribute como una clase base directa o indirecta.

Mientras se determina el significado de la especificación de clase base directa A de una clase B, se supone temporalmente que la clase base directa de B es object. De esta forma, se asegura intuitivamente que el significado de una especificación de clase base no puede depender recursivamente de sí misma. El ejemplo:

class A<T> {

public class B{}

}

class C : A<C.B> {}

es erróneo porque en la especificación de la clase base A<C.B> la clase base directa de C se considera object y, por tanto, (según las reglas de §3.8) C no se considera que tenga un miembro B.

Las clases base de un tipo de clase son la clase base directa y sus clases base. En otras palabras, el conjunto de las clases base es el cierre transitivo de la relación de clase base directa. En el ejemplo anterior, las clases base de B son A y object. En el ejemplo

class A {...}

class B<T>: A {...}

class C<T>: B<IComparable<T>> {...}

class D<T>: C<T[]> {...}

las clases base de D<int> son C<int[]>, B<IComparable<int[]>>, A y object.

Exceptuando la clase object, todos los tipos de clases tienen una clase base directa. La clase object no tiene clase base directa y es la última clase base del resto de clases.

Cuando una clase B se deriva de una clase A, se produce un error si A depende de B. Una clase depende directamente de su clase base directa (si existe), y depende directamente de la clase dentro de la cual está inmediatamente anidada (si es el caso). Dada esta definición, el conjunto completo de clases de las cuales depende una clase es el cierre reflexivo y transitivo de la relación de dependencia directa.

El ejemplo

class A: A {}

es erróneo porque la clase depende de sí misma. Igualmente, el ejemplo

class A: B {}

class B: C {}

class C: A {}

es erróneo, porque las clases tienen una dependencia circular entre ellas. Por último, el ejemplo

class A: B.C {}

class B: A  
{  
 public class C {}  
}

produce un error en tiempo de compilación porque A depende de B.C (su clase base directa), que a su vez depende de B (su clase envolvente inmediata), que depende circularmente de A.

Observe que una clase no depende de las clases anidadas internas. En el ejemplo

class A  
{  
 class B: A {}  
}

B depende de A (porque A es tanto su clase base directa como su clase envolvente inmediata), pero A no depende de B (porque B no es ni una clase base ni una clase envolvente de A). Por tanto, el ejemplo es correcto.

No es posible derivar de una clase sealed. En el ejemplo

sealed class A {}

class B: A {} // Error, cannot derive from a sealed class

la clase B es errónea porque intenta derivarse de la clase sealed A.

#### Implementaciones de interfaces

Una especificación de clase base (class-base) puede incluir una lista de tipos de interfaz, en cuyo caso se dice que la clase implementa directamente esos tipos de interfaz. Las implementaciones de interfaces se explican más detalladamente en §13.4.

### Restricciones de parámetros de tipo

Las declaraciones de tipo y de método genéricos pueden especificar opcionalmente restricciones de parámetros de tipo incluidas las cláusulas de restricciones de parámetros de tipo (type-parameter-constraints-clause).

type-parameter-constraints-clauses:  
type-parameter-constraints-clause  
type-parameter-constraints-clauses type-parameter-constraints-clause

type-parameter-constraints-clause:  
where type-parameter : type-parameter-constraints

type-parameter-constraints:  
primary-constraint  
secondary-constraints  
constructor-constraint  
primary-constraint , secondary-constraints  
primary-constraint , constructor-constraint  
secondary-constraints , constructor-constraint  
primary-constraint , secondary-constraints , constructor-constraint

primary-constraint:  
class-type  
class  
struct

secondary-constraints:  
interface-type  
type-parameter  
secondary-constraints , interface-type  
secondary-constraints , type-parameter

constructor-constraint:  
new ( )

Cada cláusula de restricciones de parámetros de tipo (type-parameter-constraints-clause) consta de un token where, seguido del nombre de un parámetro de tipo, de dos puntos y de la lista de restricciones para dicho parámetro de tipo. Puede haber como máximo una cláusula where para cada parámetro de tipo y las cláusulas where se pueden enumerar en cualquier orden. Al igual que los tokens get y set en un descriptor de acceso a propiedad, el token where no es una palabra clave.

La lista de restricciones proporcionada en una cláusula where puede incluir cualquiera de los siguientes componentes, en este orden: una restricción principal única, una o más restricciones secundarias \b y la restricción de constructor \b \b , new().

Una restricción principal puede ser un tipo de clase o la restricción de tipo de referencia class o la restricción de tipo de valor struct. Una restricción secundaria puede ser un parámetro de tipo (type-parameter) o un tipo de interfaz (interface-type).

La restricción de tipo de referencia especifica que un argumento de tipo utilizado para el parámetro de tipo debe ser un tipo de referencia. Todos los tipos de clase, interfaz, delegado y matriz, y los parámetros de tipo conocidos como un tipo de referencia (tal y como se define a continuación) cumplen con esta restricción.

La restricción de tipo de valor especifica que un argumento de tipo utilizado para el parámetro de tipo debe ser un tipo de valor que acepta valores NULL. Todos los tipos struct que aceptan valores NULL, los tipos de enumeración y los parámetros de tipo que tienen la restricción de tipo de valor satisfacen esta restricción. Tenga en cuenta que aunque se clasifique como un tipo de valor, un tipo de valor que acepte valores NULL (§4.1.10) no cumple con la restricción de tipo de valor. Un parámetro de tipo que tenga la restricción de tipo de valor no puede tener también la restricción de constructor (constructor-constraint).

Los tipos de puntero nunca pueden ser argumentos de tipo y no se considera que cumplen las restricciones de tipo de referencia o de tipo de valor.

Si una restricción es un tipo de clase o de interferencia, o un parámetro de tipo, dicho tipo especifica un “tipo base” mínimo que cada argumento de tipo utilizado para dicho parámetro de tipo debe admitir. Siempre que se utiliza un tipo construido o un método genérico, el argumento de tipo se comprueba con las restricciones en el parámetro de tipo en tiempo de compilación. El argumento de tipo suministrado debe cumplir las condiciones descritas en la sección 4.4.4.

Una restricción de tipo de clase (class-type) debe cumplir las siguientes reglas:

* El tipo debe ser un tipo de clase.
* El tipo no debe ser sealed.
* El tipo no debe ser ninguno de los tipos siguientes: System.Array, System.Delegate, System.Enum o System.ValueType.
* El tipo no debe ser object. Puesto que todos los tipos derivan de object, dicha restricción no tendría efecto si se permitiera.
* Sólo una restricción puede ser un tipo de clase para un parámetro de tipo dado.

Un tipo especificado como una restricción de tipo de interfaz (interface-type) debe cumplir las siguientes reglas:

* El tipo debe ser un tipo de interfaz.
* Un tipo no se debe especificar más de una vez en una cláusula where dada.

En cualquier caso, la restricción puede implicar cualquiera de los parámetros de tipo de la declaración de tipo o método asociada como parte de un tipo construido y puede implicar al tipo que se está declarando.

Cualquier tipo de clase o interfaz especificado como una restricción de parámetro de tipo debe ser al menos tan accesible (§3.5.4) como el tipo o método genérico que se está declarando.

Un tipo especificado como una restricción de parámetro de tipo (type-parameter) debe cumplir las siguientes reglas:

* El tipo debe ser un parámetro de tipo.
* Un tipo no se debe especificar más de una vez en una cláusula where dada.

Además, no debe haber ciclos en el gráfico de dependencia de parámetros de tipo, donde dependencia es una relación transitiva definida por:

* Si un parámetro de tipo T se utiliza como una restricción para el parámetro de tipo S, entonces S depende de T .
* Si un parámetro de tipo S depende de un parámetro de tipo T y T depende de un parámetro de tipo U, entonces S depende de U.

Dada esta relación, es un error de tiempo de compilación si un parámetro de tipo depende de sí mismo (directa o indirectamente).

Cualquier restricción debe ser coherente entre parámetros de tipo dependientes. Si un parámetro de tipo S depende del parámetro de tipo T:

* T no debe tener la restricción de tipo de valor. De lo contrario, T se sella de manera eficaz para forzar a S a ser del mismo tipo que T y eliminar así la necesidad de dos parámetros de tipo.
* Si S tiene la restricción de tipo de valor, T no debe tener una restricción de tipo de clase (class-type).
* Si S tiene una restricción de tipo de clase (class-type) A y T tiene una restricción de tipo de clase (class-type) B, debe haber una conversión de identidad o una conversión de referencia implícita de A a B o una conversión de referencia implícita de B a A.
* Si S también depende del parámetro de tipo U y U tiene una restricción de tipo de clase (class-type) A y T tiene una restricción de tipo de clase (class-type) B, debe haber una conversión de identidad o una conversión de referencia implícita de A a B o una conversión de referencia implícita de B a A.

S puede tener la restricción de tipo de valor y T puede tener la restricción de tipo de referencia. Efectivamente, esto limita T a los tipos System.Object, System.ValueType, System.Enum, y cualquier tipo de interfaz.

Si la cláusula where para un parámetro de tipo incluye una restricción de constructor (con la forma new()), es posible utilizar el operador new para crear instancias del tipo (§7.6.10.1). Cualquier argumento de tipo utilizado para un parámetro de tipo con una restricción de constructor debe tener un constructor público sin parámetros (este constructor existe de manera implícita para cualquier tipo de valor) o ser un parámetro de tipo que tenga la restricción de tipo de valor o la restricción de constructor (consulte la sección §10.1.5 para obtener más detalles).

A continuación se muestran ejemplos de restricciones:

interface IPrintable  
{  
 void Print();  
}

interface IComparable<T>  
{  
 int CompareTo(T value);  
}

interface IKeyProvider<T>  
{

T GetKey();  
}

class Printer<T> where T: IPrintable {...}

class SortedList<T> where T: IComparable<T> {...}

class Dictionary<K,V>  
 where K: IComparable<K>  
 where V: IPrintable, IKeyProvider<K>, new()  
{  
 ...  
}

El siguiente ejemplo genera un error porque provoca una circularidad en el gráfico de dependencia de los parámetros de tipo:

class Circular<S,T>  
 where S: T  
 where T: S // Error, circularity in dependency graph  
{  
 ...  
}

En los siguientes ejemplos se muestran otras situaciones no válidas:

class Sealed<S,T>  
 where S: T  
 where T: struct // Error, T is sealed  
{  
 ...  
}

class A {...}

class B {...}

class Incompat<S,T>  
 where S: A, T  
 where T: B // Error, incompatible class-type constraints  
{  
 ...  
}

class StructWithClass<S,T,U>  
 where S: struct, T  
 where T: U  
 where U: A // Error, A incompatible with struct  
{  
 ...  
}

El **borrado dinámico** de un tipo C es el tipo Co construido de la siguiente forma:

* Si C es un tipo anidado Outer.Inner, entonces Co es un tipo anidado Outero.Innero.
* Si C es un tipo construido G<A1, …, An> con argumentos de tipo A1, …, An entonces Co es el tipo construido G<A1o, …, Ano>.
* Si C es un tipo de matriz E[], entonces Co es el tipo de matriz Eo[].
* Si C es un tipo de puntero E\*, entonces Co es el tipo de puntero Eo\*.
* Si C es dynamic, entonces Co es object.
* De lo contrario, Co es C.

La clase base efectiva de un parámetro de tipo T se define de la siguiente manera:

Supongamos que R es un conjunto de tipos, de modo que:

* Para cada restricción de T que sea un parámetro de tipo (type-parameter), R contiene su clase base efectiva.
* Para cada restricción de T que sea un tipo struct (*struct-type*), R contiene System.ValueType.
* Para cada restricción de T que sea un *tipo enum*, R contiene System.Enum.
* Para cada restricción de T que sea un *tipo de delegado*, R contiene su borrado dinámico.
* Para cada restricción de T que sea un *tipo de matriz*, R contiene System.Array.
* Para cada restricción de T que sea un tipo de clase (class-type), R contiene su borrado dinámico.

Entonces

* Si T tiene la restricción de tipo de valor, su *clase* base efectiva es System.ValueType.
* De lo contrario, si R está vacío, la *clase base efectiva* es object.
* De lo contrario, la clase base efectiva de T es el tipo más abarcado (§6.4.3) del conjunto R. Si dicho conjunto no tiene un tipo más abarcado, la *clase base efectiva* de T es object. Las reglas de coherencia garantizan que exista el tipo más abarcado.

Si el parámetro de tipo es un parámetro de tipo de método cuyas restricciones se heredan del método base, la clase base efectiva se calcula a partir de la sustitución de tipos.

Estas reglas aseguran que la clase base efectiva siempre es un tipo de clase (class-type).

El conjunto de interfaces efectivas de un parámetro de tipo T se define de la siguiente manera:

* Si T no tiene restricciones secundarias (secondary-constraints), su conjunto de interfaces efectivas está vacío.
* Si T tiene restricciones de tipo de interfaz (interface-type), pero no tiene restricciones de parámetro de tipo (type-parameter), su conjunto de interfaces efectivas es el conjunto de borrados dinámicos de sus restricciones de tipo de interfaz (interface-type).
* Si T no tiene restricciones de tipo de interfaz (interface-type), pero tiene restricciones de parámetro de tipo (type-parameter), su conjunto de interfaces efectivas es la unión del conjunto de interfaces efectivas de sus restricciones de parámetro de tipo (type-parameter).
* Si T tiene restricciones de tipo de interfaz (interface-type) y restricciones de parámetro de tipo (type-parameter), su conjunto de interfaces efectivas es la unión del conjunto de borrados dinámicos de sus restricciones de tipo de interfaz (interface-type) y del conjunto de interfaces efectivas de sus restricciones de parámetro de tipo (type-parameter).

Se sabe que un parámetro de tipo es un tipo de referencia si tiene la restricción de tipo de referencia o su clase base efectiva no es object ni System.ValueType.

Se pueden utilizar valores de un tipo de parámetro de tipo restringido para obtener acceso a miembros de instancia implicados en las restricciones. En el ejemplo

interface IPrintable  
{  
 void Print();  
}

class Printer<T> where T: IPrintable  
{  
 void PrintOne(T x) {  
 x.Print();  
 }  
}

los métodos de IPrintable se pueden invocar directamente en x porque es imprescindible que T esté limitado a implementar siempre IPrintable.

### Cuerpo de clase

El cuerpo de clase (class-body) de una clase define los miembros de esa clase.

class-body:  
{ class-member-declarationsopt }

## Tipos parciales

Una declaración de tipo se puede dividir en varias declaraciones de tipo parciales. Una declaración de tipo se construye a partir de sus elementos siguiendo las reglas de esta sección, con lo cual se trata como una única declaración durante el tiempo restante de compilación y procesamiento en tiempo de ejecución del programa.

Una declaración de clase (class-declaration), declaración de struct (struct-declaration) o declaración de interfaz (interface-declaration) representa una declaración de tipo parcial si incluye un modificador partial. Esta palabra, partial, no es una palabra clave, y solo actúa como modificador si aparece inmediatamente antes de una de las palabras clave class, struct o interface en una declaración de tipo, o antes del tipo void en una declaración de método. En otros contextos, se puede utilizar como un identificador normal.

Cada parte de una declaración de tipo parcial debe incluir un modificador partial. Debe tener el mismo nombre y estar declarada en el mismo espacio de nombres o declaración de tipo que las demás partes. El modificador partial indica que es posible que existan partes adicionales de la declaración de tipo en cualquier otra parte, pero la existencia de dichas partes adicionales no es un requisito; la inclusión del modificador partial es válida en un tipo con una única declaración.

Todas las partes de un tipo parcial se deben compilar juntas para combinarlas en tiempo de compilación en una única declaración de tipo. Concretamente, los tipos parciales no permiten ampliar los tipos ya compilados.

Los tipos anidados se pueden declarar en varias partes mediante el modificador partial. Generalmente, el tipo contenedor se declara también con partial y cada parte del tipo anidado se declara en una parte diferente del tipo contenedor.

El modificador partial no se admite en declaraciones de delegado o de enumeración.

### Atributos

Los atributos de un tipo parcial se determinan combinando, en orden no especificado, los atributos de cada una de las partes. Si un atributo se coloca en varias partes, esto equivale a especificar el atributo varias veces en el tipo. Por ejemplo, las dos partes:

[Attr1, Attr2("hello")]  
partial class A {}

[Attr3, Attr2("goodbye")]  
partial class A {}

equivalen a una declaración similar a la siguiente:

[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]  
class A {}

Los atributos en parámetros de tipo se combinan de manera similar.

### Modificadores

Cuando una declaración de tipo parcial incluye una especificación de accesibilidad (los modificadores public, protected, internal y private), debe concordar con las otras partes que incluyen una especificación de accesibilidad. Si ninguna de las partes de un tipo parcial incluye una especificación de accesibilidad, al tipo se le concede la accesibilidad predeterminada apropiada (§3.5.1).

Si una o más declaraciones parciales de un tipo anidado incluyen un modificador new, no se notifica ninguna advertencia si el tipo anidado oculta un miembro heredado (§3.7.1.2).

Si una o más declaraciones parciales de una clase incluyen un modificador abstract, la clase se considera abstracta (§10.1.1.1). De lo contrario, la clase se considera no abstracta.

Si una o más declaraciones parciales de una clase incluyen un modificador sealed, la clase se considera sealed (§10.1.1.2). De lo contrario, la clase se considera unsealed.

Tenga en cuenta que una clase no puede ser abstracta y sealed al mismo tiempo.

Cuando se utiliza el modificador unsafe en una declaración de tipo parcial, sólo dicha parte concreta se considera en un contexto no seguro (§18.1).

### Parámetros de tipo y restricciones

Si un tipo genérico se declara en varias partes, cada una de ellas debe especificar los parámetros de tipo. Cada parte debe tener el mismo número de parámetros de tipo y el mismo nombre para cada parámetro de tipo en orden.

Cuando una declaración de tipo genérico incluye restricciones (cláusulas where), estas deben coincidir con el resto de partes que incluyen restricciones. De manera específica, cada parte que incluye restricciones debe tener restricciones para el mismo conjunto de parámetros de tipo y para cada parámetro de tipo los conjuntos de restricciones principales, secundarias y de constructor deben ser equivalentes. Dos conjuntos de restricciones son equivalentes si contienen los mismos miembros. Si ninguna parte de un tipo genérico parcial especifica las restricciones de parámetros de tipo, se considera que estos no tienen restricciones.

El ejemplo

partial class Dictionary<K,V>  
 where K: IComparable<K>  
 where V: IKeyProvider<K>, IPersistable  
{  
 ...  
}

partial class Dictionary<K,V>  
 where V: IPersistable, IKeyProvider<K>  
 where K: IComparable<K>  
{  
 ...  
}

partial class Dictionary<K,V>  
{  
 ...  
}

es correcto porque aquellas partes que incluyen restricciones (las dos primeras) especifican efectivamente el mismo conjunto de restricciones principales, secundarias y de constructor para el mismo conjunto de parámetros de tipo respectivamente.

### Clase base

Cuando una declaración de clase parcial incluye una especificación de clase base, debe coincidir con el resto de partes que incluyen una especificación de clase base. Si ninguna parte de una clase parcial incluye una especificación de clase base, la clase base se convierte en System.Object (§10.1.4.1).

### Interfaces base

El conjunto de interfaces base para un tipo declarado en varias partes es la unión de las interfaces base especificadas en cada parte. Una interfaz base concreta sólo se puede nombrar una vez en cada parte, pero varias partes sí pueden nombrar la misma interfaz base (o las mismas interfaces base). Sólo debe haber una implementación de los miembros de cualquier interfaz base dada.

En el ejemplo

partial class C: IA, IB {...}

partial class C: IC {...}

partial class C: IA, IB {...}

el conjunto de interfaces base para la clase C es IA, IB e IC.

Generalmente, cada parte proporciona una implementación de las interfaces declaradas en dicha parte; sin embargo, esto no es un requisito. Una parte puede proporcionar la implementación para una interfaz declarada en otra parte diferente:

partial class X  
{  
 int IComparable.CompareTo(object o) {...}  
}

partial class X: IComparable  
{  
 ...  
}

### Miembros

Con la excepción de los métodos parciales (§10.2.7), el conjunto de miembros de un tipo declarado en varias partes es simplemente la unión del conjunto de miembros declarados en cada una de las partes. Los cuerpos de todas las partes de la declaración de tipo comparten el mismo espacio de declaración (§3.3) y el ámbito de cada miembro (§3.7) se extiende a los cuerpos de todas las partes. El dominio de accesibilidad de todo miembro siempre incluye todas las partes del tipo contenedor; se puede obtener acceso libremente a un miembro private declarado en otra parte. Se genera un error en tiempo de compilación si se declara el mismo miembro en más de una parte del tipo a menos que dicho miembro sea un tipo con el modificador partial.

partial class A  
{  
 int x; // Error, cannot declare x more than once

partial class Inner // Ok, Inner is a partial type  
 {  
 int y;  
 }  
}

partial class A  
{  
 int x; // Error, cannot declare x more than once

partial class Inner // Ok, Inner is a partial type  
 {  
 int z;  
 }  
}

Dentro de un tipo el orden de los miembros raramente influye en el código C#, pero puede ser importante a la hora de interactuar con otros lenguajes y entornos. En estos casos, el orden de los miembros dentro de un tipo declarado en varias partes es indefinido.

### Métodos parciales

Los métodos parciales se pueden definir en una parte de una declaración de tipo e implementarse en otra. Esta implementación es opcional; si ninguna parte implementa el método parcial, la declaración de método parcial y todas sus llamadas se quitan de la declaración de tipo resultante de la combinación de las partes.

Los métodos parciales no pueden definir el acceso a los modificadores, pero son implícitamente private. Su tipo de valor devuelto debe ser void, y sus parámetros no pueden tener el modificador out. El identificador partial se reconoce como una palabra clave especial en una declaración de método solo si aparece justo antes del tipo void; de lo contrario se puede usar como un identificador normal. Un método parcial no puede implementar de manera explícita métodos de interfaz.

Hay dos clases de declaraciones de método parciales. Si el cuerpo de la declaración de método es un punto y coma, se dice que la declaración es una declaración de método parcial de definición. Si el cuerpo se da como un bloque, se dice que la declaración es una declaración de método parcial de implementación. En las partes de una declaración de tipo sólo puede haber una declaración de método parcial de definición con una forma dada, y sólo puede haber una declaración de método parcial de implementación con una firma dada. Si se proporciona una declaración de método parcial de implementación, debe existir una declaración de método parcial de definición, y las declaraciones deben coincidir de la manera que se especifica a continuación:

* Las declaraciones deben tener los mismos modificadores (aunque no necesariamente en el mismo orden), el mismo nombre de método, número de parámetros de tipo y número de parámetros.
* Los parámetros correspondientes en las declaraciones deben tener los mismos modificadores (aunque no necesariamente en el mismo orden), y los mismos tipos (diferencias de módulo en los nombres de parámetros de tipo).
* Los parámetros de tipo correspondientes en las declaraciones deben tener las mismas restricciones (diferencias de módulo en los nombres de parámetros de tipo).

Una declaración de método parcial de implementación puede aparecer en la misma parte que la declaración de método parcial de definición correspondiente.

Sólo participa un método parcial de definición en la resolución de sobrecargas. Por lo tanto, tanto si se da como si no se da una declaración de implementación, las expresiones de invocación pueden resolverse en invocaciones de método parcial. Dado que un método parcial siempre devuelve void, dichas expresiones de invocación siempre serán instrucciones de expresión. Es más, debido a que un método parcial es implícitamente private, dichas declaraciones siempre ocurrirán dentro de una de las partes de la declaración de tipo en de la cual se declara el método parcial.

Si ninguna declaración de tipo parcial contiene una declaración de implementación para un método parcial dado, se quitarán de la declaración de tipos combinada todas las instrucciones de expresión que la invoquen. Por lo tanto, la expresión de invocación, incluidas sus expresiones constituyentes, no tiene ningún efecto en tiempo de ejecución. El método parcial en sí también se quitará y no será miembro de la declaración de tipos combinada.

Si existe una declaración de implementación para un método parcial dado, se conservan las invocaciones de los métodos parciales. El método parcial da lugar a una declaración de método similar a la declaración de método parcial de implementación para lo siguiente:

* No se incluye el modificador partial
* Los atributos de la declaración de método resultante son los atributos combinados de la declaración de método parcial de implementación y de definición en un orden no especificado. No se quitan duplicados.
* Los atributos de los parámetros en la declaración de método resultante son los atributos combinados de los parámetros correspondientes en la declaración de método parcial de implementación y de definición en un orden no especificado. No se quitan duplicados.

Si se proporciona una declaración de definición, pero no una declaración de implementación, para un método parcial M, se aplican las siguientes restricciones:

* Es un error en tiempo de compilación crear un delegado para el método (§7.6.10.5).
* Es un error en tiempo de compilación hacer referencia a M dentro de una función anónima que se convierte a un tipo de árbol de expresiones (§6.5.2).
* Las expresiones que ocurren como parte de una invocación de M no afectan al estado de asignación definitivo (§5.3), que puede conducir potencialmente a errores en tiempo de compilación.
* M no puede ser el punto de entrada para una aplicación (§3.1).

Los métodos parciales son útiles para permitir que una parte de una declaración de tipo personalice el comportamiento de otra parte, por ejemplo, una que se genera mediante una herramienta. Considere la siguiente declaración de clase parcial:

partial class Customer  
{  
 string name;

public string Name {

get { return name; }

set {  
 OnNameChanging(value);  
 name = value;  
 OnNameChanged();  
 }

}

partial void OnNameChanging(string newName);

partial void OnNameChanged();  
}

Si esta clase se compila sin ninguna otra parte, se quitarán las declaraciones de método parcial de definición y sus invocaciones, y la declaración de clase combinada resultante será equivalente a la siguiente:

class Customer  
{  
 string name;

public string Name {

get { return name; }

set { name = value; }  
 }  
}

No obstante, si se asume que se proporciona otra parte, que proporciona declaraciones de implementación de métodos parciales:

partial class Customer  
{  
 partial void OnNameChanging(string newName)  
 {  
 Console.WriteLine(“Changing “ + name + “ to “ + newName);  
 }

partial void OnNameChanged()  
 {  
 Console.WriteLine(“Changed to “ + name);  
 }  
}

La declaración de clase combinada resultante será equivalente a la siguiente:

class Customer  
{  
 string name;

public string Name {

get { return name; }

set {  
 OnNameChanging(value);  
 name = value;  
 OnNameChanged();  
 }

}

void OnNameChanging(string newName)  
 {  
 Console.WriteLine(“Changing “ + name + “ to “ + newName);  
 }

void OnNameChanged()  
 {  
 Console.WriteLine(“Changed to “ + name);  
 }  
}

### Enlace de nombres

Aunque cada parte de un tipo extensible se debe declarar dentro del mismo espacio de nombres, las partes generalmente se escriben dentro de diferentes declaraciones de espacios de nombres. Por lo tanto, puede haber diferentes directivas using (§9.4) para cada parte. Al interpretar nombres simples (§7.5.2) dentro de una parte, sólo se consideran las directivas using de las declaraciones de espacios de nombres contenedor. El resultado que se obtenga puede ser el mismo identificador con diferentes significados en diferentes partes:

namespace N  
{  
 using List = System.Collections.ArrayList;

partial class A  
 {  
 List x; // x has type System.Collections.ArrayList  
 }  
}

namespace N  
{  
 using List = Widgets.LinkedList;

partial class A  
 {  
 List y; // y has type Widgets.LinkedList  
 }  
}

## Miembros de clase

Los miembros de una clase se componen de los miembros que introducen las declaraciones de miembros de clase (class-member-declaration) y los miembros que ésta hereda de su clase base directa.

class-member-declarations:  
class-member-declaration  
class-member-declarations class-member-declaration

class-member-declaration:  
constant-declaration  
field-declaration  
method-declaration  
property-declaration  
event-declaration  
indexer-declaration  
operator-declaration  
constructor-declaration  
destructor-declaration  
static-constructor-declaration  
type-declaration

Los miembros de un tipo de clase se dividen en las siguientes categorías:

* Constantes, que representan valores constantes asociados a la clase (§10.4).
* Campos, que son las variables de la clase (§10.5).
* Métodos, que implementan los cálculos y acciones que puede realizar la clase (§10.6).
* Propiedades, que definen características con nombre y las acciones asociadas con la lectura y la escritura de esas características (§10.7).
* Eventos, que definen las notificaciones que puede generar la clase (§10.8).
* Indizadores, que permiten indizar las instancias de la clase de la misma manera (sintácticamente) que las matrices (§10.9).
* Operadores, que definen los operadores de expresión que se pueden aplicar a las instancias de la clase (§10.10).
* Constructores de instancia, que implementan las acciones necesarias para inicializar las instancias de la clase (§10.11).
* Destructores, que implementan las acciones que se deben realizar antes de que las instancias de la clase sean descartadas de forma permanente (§10.13).
* Constructores estáticos, que implementan las acciones necesarias para inicializar la propia clase (§10.12).
* Tipos, que representan los tipos locales de la clase (§10.3.8).

A los miembros que pueden contener código ejecutable se los conoce colectivamente como miembros de función del tipo de clase. Los miembros de función de un tipo de clase son los métodos, propiedades, eventos, indizadores, operadores, constructores de instancia, destructores y constructores estáticos de dicho tipo de clase.

Una declaración de clase (class-declaration) crea un nuevo espacio de declaración (§3.3), y las declaraciones de miembros de clase (class-member-declarations) inmediatamente contenidas en la declaración de clase (class-declaration) introducen nuevos miembros en este espacio de declaración. Se aplican las siguientes reglas a las declaraciones de miembros de clase (class-member-declaration):

* Los constructores de instancia, los constructores estáticos y los destructores deben tener el mismo nombre que su clase envolvente inmediata. El resto de miembros deben tener nombres diferentes al de la clase envolvente inmediata.
* El nombre de una constante, campo, propiedad, evento o tipo debe ser diferente de los nombres del resto de miembros declarados en la misma clase.
* El nombre de un método debe ser diferente de los nombres del resto de miembros que no sean métodos declarados en la misma clase. Además, la signatura (§3.6) de un método debe ser distinta de las signaturas de todos los demás métodos declarados en la misma clase, y dos métodos declarados en la misma clase no pueden tener signaturas que solo se diferencien por ref y out.
* La signatura de un constructor de instancia debe diferir de las signaturas de todos los demás constructores de instancia declaradas en la misma clase, y dos constructores declarados en la misma clase no pueden tener signaturas que solo se diferencien en ref y out.
* La firma de un indizador debe ser diferente de las firmas de los demás indizadores declarados en la misma clase.
* La firma de un operador debe ser diferente de las firmas de los demás operadores declarados en la misma clase.

Los miembros heredados de un tipo de clase (§10.3.3) no forman parte del espacio de declaración de la clase. Por ello, una clase derivada puede declarar un miembro con el mismo nombre o firma que el de un miembro heredado (que en realidad oculta el miembro heredado).

### El tipo de instancia

Cada declaración de clase tiene un tipo enlazado (§4.4.3), el tipo de instancia. Para una declaración de clase genérica, el tipo de instancia se forma creando un tipo construido (§4.4) a partir de la declaración de tipo, siendo cada uno de los argumentos de tipo suministrados el parámetro de tipo equivalente. Puesto que el tipo de instancia utiliza los parámetros de tipo, sólo se puede utilizar donde los parámetros de tipo están dentro del ámbito, es decir, dentro de la declaración de clase. El tipo de instancia es el tipo de this para el código escrito dentro de la declaración de clase. Para las clases no genéricas, el tipo de instancia es simplemente la clase declarada. En el siguiente ejemplo se muestran varias declaraciones de clase junto con sus tipos de instancia:

class A<T> // instance type: A<T>  
{  
 class B {} // instance type: A<T>.B

class C<U> {} // instance type: A<T>.C<U>  
}

class D {} // instance type: D

### Miembros de tipos construidos

Los miembros no heredados de un tipo construido se obtienen sustituyendo, para cada parámetro de tipo (type-parameter) de la declaración de miembro, el argumento de tipo (type-argument) correspondiente del tipo construido. El proceso de sustitución se basa en el significado semántico de las declaraciones de tipo y no se trata simplemente de una sustitución textual.

Por ejemplo, dadas la declaración de clase genérica:

class Gen<T,U>  
{  
 public T[,] a;

public void G(int i, T t, Gen<U,T> gt) {...}

public U Prop { get {...} set {...} }

public int H(double d) {...}  
}

el tipo construido Gen<int[],IComparable<string>> tiene los miembros siguientes:

public int[,][] a;

public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}

public IComparable<string> Prop { get {...} set {...} }

public int H(double d) {...}

El tipo del miembro a de una declaración de clase genérica Gen es una “matriz bidimensional de T”, por lo que el tipo del miembro a del tipo construido mencionado anteriormente es una “matriz bidimensional de una matriz unidimensional de int” o int[,][].

Dentro de los miembros de función de instancia, el tipo de this es el tipo de instancia (§10.3.1) de la declaración contenedora.

Todos los miembros de una clase genérica pueden utilizar parámetros de tipo de cualquier clase envolvente, ya sea directamente o como parte de un tipo construido. Cuando un tipo construido cerrado concreto (§4.4.2) se utiliza en tiempo de ejecución, cada uso de un parámetro de tipo se reemplaza con el argumento de tipo real suministrado al tipo construido. Por ejemplo:

class C<V>  
{  
 public V f1;  
 public C<V> f2 = null;

public C(V x) {  
 this.f1 = x;  
 this.f2 = this;  
 }  
}

class Application  
{  
 static void Main() {  
 C<int> x1 = new C<int>(1);  
 Console.WriteLine(x1.f1); // Prints 1

C<double> x2 = new C<double>(3.1415);  
 Console.WriteLine(x2.f1); // Prints 3.1415  
 }  
}

### Herencia

Una clase hereda los miembros de su tipo de clase base directa. La herencia significa que una clase contiene implícitamente todos los miembros de su tipo de clase base directa, excepto los constructores de instancia, los constructores estáticos y los destructores de la clase base. Estos son algunos aspectos importantes de la herencia:

* La herencia es transitiva. Si C se deriva de B y B se deriva de A, entonces C hereda los miembros declarados en B y los miembros declarados en A.
* Una clase derivada extiende su clase base directa. Una clase derivada puede agregar nuevos miembros a los heredados, pero no puede quitar la definición de un miembro heredado.
* Los constructores de instancia, los constructores estáticos y los destructores no se heredan, pero el resto de miembros sí, independientemente de la accesibilidad que tengan declarada (§3.5). Sin embargo, dependiendo de la accesibilidad que tengan declarada, es posible que los miembros heredados no sean accesibles en una clase derivada.
* Una clase derivada puede ocultar (§3.7.1.2) miembros heredados declarando nuevos miembros con el mismo nombre o signatura. No obstante, tenga en cuenta que ocultar un miembro heredado no significa quitarlo, simplemente hace que el miembro no sea directamente accesible a través de la clase derivada.
* Una instancia de una clase contiene un conjunto de todos los campos de instancia declarados en la clase y en sus clases base y se produce una conversión implícita (§6.1.6) de los tipos de la clase derivada a cualquiera de los tipos de su clase base. Una referencia a una instancia de una clase derivada puede ser tratada como referencia a una instancia de cualquiera de sus clases base.
* Una clase puede declarar métodos, propiedades e indizadores virtuales, y las clases derivadas invalidan la implementación de dichos miembros de función. Esto permite que las clases muestren un comportamiento polimórfico, ya que las acciones que realiza una llamada a un miembro de función varían según el tipo de instancia a través de la que se invoca al miembro de función en tiempo de ejecución.

El miembro heredado de un tipo de clase construido son los miembros del tipo de clase base inmediato (§10.1.4.1), que se halla al sustituir los argumentos de tipo del tipo construido en cada ocurrencia de los parámetros de tipo correspondientes en la especificación de clase base (base-class-specification). A su vez, estos miembros se transforman mediante la sustitución en cada parámetro de tipo (type-parameter) dentro de la declaración de miembro, del argumento de tipo (type-argument) correspondiente de la especificación de clase base (base-class-specification).

class B<U>  
{  
 public U F(long index) {...}  
}

class D<T>: B<T[]>  
{  
 public T G(string s) {...}  
}

En el ejemplo anterior, el tipo construido D<int> tiene un miembro no heredado public int G(string s) que se obtiene sustituyendo el argumento tipo int para el parámetro de tipo T. D<int> también tiene un miembro heredado de la declaración de clase B. Este miembro heredado se determina especificando primero el tipo de la clase base B<int[]> de D<int> mediante la sustitución de int por T en la especificación de la clase base B<T[]>. Entonces, como un argumento de tipo para B, int[] se sustituye por U en public U F(long index), que da como resultado el miembro heredado public int[] F(long index).

### Modificador New

Una declaración de miembro de clase (class-member-declaration) puede declarar un miembro con el mismo nombre o signatura que un miembro heredado. Cuando esto ocurre, se dice que el miembro de la clase derivada oculta el miembro de la clase base. Ocultar un miembro heredado no se considera un error, pero hace que el compilador emita una advertencia. Para suprimir la advertencia, la declaración del miembro de la clase derivada puede incluir un modificador new para indicar que el miembro derivado pretende ocultar el miembro base. Este tema se explica con más detalle en la sección §3.7.1.2.

Si se incluye un modificador new en una declaración que no oculta un miembro heredado, se emite una advertencia a tal efecto. Esta advertencia se evita quitando el modificador new.

### Modificadores de acceso

Una declaración de miembro de clase (class-member-declaration) puede tener una de las cinco clases de accesibilidad declarada (§3.5.1): public, protected internal, protected, internal o private. Excepto en la combinación protected internal, especificar más de un modificador de acceso produce un error en tiempo de compilación. Cuando una declaración de miembro de clase (class-member-declaration) no incluye ningún modificador de acceso, se supone que es private.

### Tipos constituyentes

Los tipos que se utilizan en la declaración de un miembro se denominan tipos constituyentes del miembro. Los posibles tipos constituyentes son los tipos de una constante, de un campo, de una propiedad, de un evento o de un indizador, el tipo de valor devuelto de un método u operador y los tipos de los parámetros de un método, indizador, operador o constructor de instancia. Los tipos constituyentes de un miembro deben ser al menos tan accesibles como el propio miembro (§3.5.4).

### Miembros estáticos y de instancia

Los miembros de una clase son miembros estáticos o miembros de instancia. En términos generales, se podría decir que los miembros estáticos pertenecen a los tipos de clases y los miembros de instancia pertenecen a los objetos (instancias de tipos de clases).

Cuando la declaración de un campo, método, propiedad, evento, operador o constructor incluye un modificador static, está declarando un miembro estático. Una declaración de constante o tipos declara implícitamente un miembro estático. Los miembros estáticos tienen las siguientes características:

* Cuando se hace referencia a un miembro estático M en un acceso a miembro (member-access) (§7.6.4) de la forma E.M, E debe denotar un tipo que contenga M. Se produce un error en tiempo de compilación si E denota una instancia.
* Un campo estático identifica de manera exacta una ubicación de almacenamiento para que sea compartida por todas las instancias de un tipo de clase cerrado dado. Independientemente del número de instancias que se creen de un tipo de clase cerrad determinado, sólo habrá una copia de un campo estático.
* Un miembro de función estático (método, propiedad, evento, operador o constructor) no opera en una instancia específica, y se produce un error en tiempo de compilación si se hace referencia a this en dicho miembro de función.

Cuando la declaración de un campo, método, propiedad, evento, indizador, constructor o destructor no incluye un modificador static, está declarando un miembro de instancia (los miembros de instancia se conocen también como miembros no estáticos). Los miembros de instancia tienen las siguientes características:

* Cuando se hace referencia a un miembro de instancia M en un acceso a miembro (member-access) (§7.6.4) de la forma E.M, E debe denotar una instancia de un tipo que contenga M. Se produce un error en tiempo de enlace si E denota un tipo.
* Cada instancia de una clase contiene un conjunto independiente de todos los campos de instancia de la clase.
* Un miembro de función de instancia (método, propiedad, indizador, constructor de instancia o destructor) opera en una determinada instancia de la clase y es posible tener acceso a dicha instancia con this (§7.6.7).

En el ejemplo siguiente se ilustran las reglas para el acceso a miembros estáticos y de instancia:

class Test  
{  
 int x;  
 static int y;

void F() {  
 x = 1; // Ok, same as this.x = 1  
 y = 1; // Ok, same as Test.y = 1  
 }

static void G() {  
 x = 1; // Error, cannot access this.x  
 y = 1; // Ok, same as Test.y = 1  
 }

static void Main() {  
 Test t = new Test();  
 t.x = 1; // Ok  
 t.y = 1; // Error, cannot access static member through instance  
 Test.x = 1; // Error, cannot access instance member through type  
 Test.y = 1; // Ok  
 }  
}

El método F muestra que se puede utilizar un nombre simple (simple-name) (§7.6.2) en un miembro de función de instancia para tener acceso tanto a miembros de instancia como a miembros estáticos. El método G muestra que en un miembro de función estático, se produce un error en tiempo de compilación si se obtiene acceso a un miembro de instancia a través de un nombre simple (simple-name). El método Main muestra que en un acceso de miembro (member-access) (§7.6.4), se deben utilizar instancias para tener acceso a los miembros de instancia y tipos para tener acceso a los miembros estáticos.

### Tipos anidados

Un tipo declarado dentro de una declaración de struct o clase se denomina tipo anidado. Un tipo declarado dentro de una unidad de compilación o de un espacio de nombres se denomina tipo no anidado.

En el ejemplo

using System;

class A  
{  
 class B  
 {  
 static void F() {  
 Console.WriteLine("A.B.F");  
 }  
 }  
}

la clase B es un tipo anidado porque se declara dentro de la clase A y la clase A es un tipo no anidado porque se declara dentro de una unidad de compilación.

#### Nombre completo

El nombre completo (§3.8.1) de un tipo anidado es S.N, donde S es el nombre completo del tipo en el que se declara el tipo N.

#### Accesibilidad declarada

Los tipos no anidados pueden tener accesibilidad declarada public o internal, aunque la accesibilidad declarada predeterminada es internal. Los tipos anidados también pueden tener esas mismas formas de accesibilidad declarada y otras adicionales, en función de si el tipo que los contiene es una clase o un struct:

* Un tipo anidado declarado en una clase puede tener cualquiera de las cinco formas de accesibilidad declarada (public, protected internal, protected, internal o private) y, como en otros miembros de clase, la accesibilidad declarada predeterminada es private.
* Un tipo anidado declarado en un struct puede tener cualquiera de las tres formas de accesibilidad declarada (public, internal o private) y, como en otros miembros de struct, la accesibilidad declarada predeterminada es private.

El ejemplo

public class List  
{  
 // Private data structure  
 private class Node  
 {   
 public object Data;  
 public Node Next;

public Node(object data, Node next) {  
 this.Data = data;  
 this.Next = next;  
 }  
 }

private Node first = null;  
 private Node last = null;

// Public interface

public void AddToFront(object o) {...}

public void AddToBack(object o) {...}

public object RemoveFromFront() {...}

public object RemoveFromBack() {...}

public int Count { get {...} }  
}

declara una clase anidada privada denominada Node.

#### Ocultar

Un tipo anidado puede ocultar (§3.7.1) un miembro de base. Está permitido utilizar el modificador new en las declaraciones de tipos anidados, por lo que dicha ocultación se puede expresar explícitamente. El ejemplo

using System;

class Base  
{  
 public static void M() {  
 Console.WriteLine("Base.M");  
 }  
}

class Derived: Base   
{  
 new public class M   
 {  
 public static void F() {  
 Console.WriteLine("Derived.M.F");  
 }  
 }  
}

class Test   
{  
 static void Main() {  
 Derived.M.F();  
 }  
}

muestra una clase anidada M que oculta el método M definido en Base.

#### Acceso this

Un tipo anidado y el tipo que lo contiene no tienen una relación especial con respecto al acceso this (this-access) (§7.6.7). En concreto, no se puede utilizar this dentro de un tipo anidado para hacer referencia a miembros de instancia del tipo que lo contiene. En los casos en los que un tipo anidado necesite tener acceso a los miembros de instancia del tipo contenedor, se puede utilizar this para la instancia del tipo contenedor como argumento de constructor del tipo anidado. El siguiente ejemplo

using System;

class C  
{  
 int i = 123;

public void F() {  
 Nested n = new Nested(this);  
 n.G();  
 }

public class Nested  
 {  
 C this\_c;

public Nested(C c) {  
 this\_c = c;  
 }

public void G() {  
 Console.WriteLine(this\_c.i);  
 }  
 }  
}

class Test  
{  
 static void Main() {  
 C c = new C();  
 c.F();  
 }  
}

muestra esta técnica. Una instancia de C crea una instancia de Nested y pasa su propio this al constructor de Nested para proporcionar acceso posterior a los miembros de instancia de C.

#### Acceso a miembros privados y protegidos del tipo contenedor

Un tipo anidado tiene acceso a todos los miembros accesibles a su tipo contenedor, incluyendo miembros que tienen accesibilidad declarada private y protected. El ejemplo

using System;

class C   
{  
 private static void F() {  
 Console.WriteLine("C.F");  
 }

public class Nested   
 {  
 public static void G() {  
 F();  
 }  
 }  
}

class Test   
{  
 static void Main() {  
 C.Nested.G();  
 }  
}

muestra una clase C que contiene una clase anidada Nested. Dentro de Nested, el método G llama al método estático F definido en C, y F tiene accesibilidad declarada privada.

Un tipo anidado también puede tener acceso a miembros protegidos definidos en un tipo base de su tipo contenedor. En el ejemplo

using System;

class Base   
{  
 protected void F() {  
 Console.WriteLine("Base.F");  
 }  
}

class Derived: Base   
{  
 public class Nested   
 {  
 public void G() {  
 Derived d = new Derived();  
 d.F(); // ok  
 }  
 }  
}

class Test   
{  
 static void Main() {  
 Derived.Nested n = new Derived.Nested();  
 n.G();  
 }  
}

la clase anidada Derived.Nested obtiene acceso al método protegido F definido en la clase base de Derived, Base, llamando a través de una instancia de Derived.

#### Tipos anidados en clases genéricas

Una declaración de clase genérica puede contener declaraciones de tipo anidado. Los parámetros de tipo de la clase envolvente se pueden utilizar dentro de los tipos anidados. Una declaración de tipo anidado puede contener parámetros de tipo adicionales que sólo se apliquen al tipo anidado.

Cada declaración de tipo contenida dentro de una declaración de clase genérica es implícitamente una declaración de tipo genérico. Cuando se escribe una referencia en un tipo anidado dentro de un tipo genérico, debe enumerarse el tipo construido contenedor y sus argumentos de tipo. Sin embargo, desde dentro de la clase exterior, el tipo anidado se puede utilizar sin cualificación; el tipo de instancia de la clase exterior se puede utilizar implícitamente al construir el tipo anidado. En el siguiente ejemplo se muestran tres maneras diferentes de hacer referencia a un tipo construido creado desde Inner; los dos primeros son equivalentes:

class Outer<T>  
{  
 class Inner<U>  
 {  
 public static void F(T t, U u) {...}  
 }

static void F(T t) {  
 Outer<T>.Inner<string>.F(t, "abc"); // These two statements have  
 Inner<string>.F(t, "abc"); // the same effect

Outer<int>.Inner<string>.F(3, "abc"); // This type is different

Outer.Inner<string>.F(t, "abc"); // Error, Outer needs type arg  
 }  
}

Aunque el estilo de programación no es correcto, un parámetro de tipo en un tipo anidado puede ocultar un parámetro de miembro o de tipo en el tipo externo:

class Outer<T>  
{  
 class Inner<T> // Valid, hides Outer’s T  
 {  
 public T t; // Refers to Inner’s T  
 }  
}

### Nombres de miembro reservados

Para facilitar la implementación subyacente de C# en tiempo de ejecución, la implementación, para cada declaración de miembro del código fuente que sea una propiedad, un evento o un indizador, debe reservar dos firmas de métodos basadas en el tipo de la declaración de miembro, su nombre y su tipo. Se produce un error en tiempo de compilación cuando un programa declara un miembro cuya firma coincida con una de estas firmas reservadas, incluso aunque la implementación subyacente en tiempo de ejecución no haga uso de dichas reservas.

Los nombres reservados no presentan declaraciones, por lo que no intervienen en la búsqueda de miembros. Sin embargo, las signaturas de métodos reservados asociados a una declaración sí intervienen en la herencia (§10.3.3) y pueden ser ocultados por el modificador new (§10.3.4).

La reserva de estos nombres tiene tres propósitos:

* Permitir que la implementación subyacente utilice un identificador normal como nombre de método para obtener o establecer el acceso a la característica del lenguaje C#.
* Permitir la interoperabilidad con otros lenguajes utilizando un identificador normal como nombre de método para obtener o establecer el acceso a la característica del lenguaje C#.
* Asegurar que el código fuente aceptado por un compilador compatible también es aceptado por otros compiladores, haciendo que las especificaciones de nombres de miembros reservados sean consistentes a través de todas las implementaciones de C#.

La declaración de un destructor (§10.13) también causa la reserva de una signatura (§10.3.9.4).

#### Nombres de miembros reservados para propiedades

Para una propiedad P (§10.7) de tipo T, se reservan las siguientes signaturas:

T get\_P();  
void set\_P(T value);

Se reservan ambas firmas, tanto si la propiedad es de sólo lectura como de sólo escritura.

En el ejemplo

using System;

class A  
{  
 public int P {  
 get { return 123; }  
 }  
}

class B: A  
{  
 new public int get\_P() {  
 return 456;  
 }

new public void set\_P(int value) {  
 }  
}

class Test  
{  
 static void Main() {  
 B b = new B();  
 A a = b;  
 Console.WriteLine(a.P);  
 Console.WriteLine(b.P);  
 Console.WriteLine(b.get\_P());  
 }  
}

una clase A define una propiedad de solo lectura P, con lo que reserva las signaturas para los métodos get\_P y set\_P. Una clase B se deriva de A y oculta ambas signaturas reservadas. El ejemplo produce el resultado:

123  
123  
456

#### Nombres de miembros reservados para eventos

Para un evento E (§10.8) de tipo delegado T, se reservan las siguientes signaturas:

void add\_E(T handler);  
void remove\_E(T handler);

#### Nombres de miembros reservados para indizadores

Para un indizador (§10.9) de tipo T con la lista de parámetros L, se reservan las siguientes signaturas:

T get\_Item(L);  
void set\_Item(L, T value);

Se reservan ambas firmas, tanto si el indizador es de sólo lectura como de sólo escritura.

Además, se reserva el nombre de miembro Item.

#### Nombres de miembros reservados para destructores

Para una clase que contiene un destructor (§10.13), se reserva la siguiente signatura:

void Finalize();

## Constantes

Una constante es un miembro de clase que representa un valor constante: un valor que puede calcularse en tiempo de compilación. Una declaración de constante (constant-declaration) especifica una o varias constantes de un determinado tipo.

constant-declaration:  
attributesopt constant-modifiersopt const type constant-declarators ;

constant-modifiers:  
constant-modifier  
constant-modifiers constant-modifier

constant-modifier:  
new  
public  
protected  
internal  
private

constant-declarators:  
constant-declarator  
constant-declarators , constant-declarator

constant-declarator:  
identifier = constant-expression

Una declaración de constante (constant-declaration) puede incluir un conjunto de atributos (attributes) (§17), un modificador new (§10.3.4) y una combinación válida de los cuatro modificadores de acceso (§10.3.5). Los atributos y modificadores se aplican a todos los miembros declarados en la declaración de constante (constant-declaration). Aunque las constantes se consideran miembros estáticos, una declaración de constante (constant-declaration) no necesita ni permite un modificador static. Cuando el mismo modificador aparece varias veces en una declaración de constante, se produce un error.

El tipo (type) de una declaración de constante (constant-declaration) especifica el tipo de los miembros que se incluyen en la declaración. El tipo va seguido de una lista de declaradores-de-constante (constant-declarator), cada uno de los cuales incluye un nuevo miembro. Un declarador de constante (constant-declarator) consta de un identificador (identifier) que da nombre al miembro, seguido del token “=” y de una expresión constante (constant-expression) (§7.19) que establece el valor del miembro.

El tipo (type) especificado en una declaración de constante debe ser sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, string, un tipo enum (enum-type) o un tipo de referencia (reference-type). Cada expresión constante (constant-expression) debe dar un valor del tipo destino o de un tipo que se pueda convertir al tipo destino mediante una conversión implícita (§6.1).

El tipo (type) de una constante debe ser al menos tan accesible como la propia constante (§3.5.4).

Para obtener el valor de una constante en una expresión se utiliza un nombre simple (simple-name) (§7.6.2) o un acceso a miembro (member-access) (§7.6.4).

Una constante puede participar en una expresión constante (constant-expression). Por lo tanto, se puede utilizar una constante en cualquier constructor que necesite una expresión constante (constant-expression). Ejemplos de este tipo de constructores incluyen las etiquetas case, instrucciones goto case, declaraciones de miembros enum, atributos y otras declaraciones de constantes.

Como se describe en la §7.19, una expresión constante (constant-expression) es una expresión que se puede evaluar completamente en tiempo de compilación. Puesto que la única forma de crear un valor no NULL de un tipo de referencia (reference-type) distinto de string es aplicar el operador new, y como el operador new no está permitido en una expresión constante (constant-expression), el único valor posible para las constantes de tipos de referencia (reference-types) distintos de string es null.

Cuando se desea un nombre simbólico para un valor constante, pero el tipo del valor no se permite en una declaración de constante, o cuando el valor no se puede calcular en tiempo de compilación con una expresión constante (constant-expression), en su lugar puede utilizarse un campo readonly (§10.5.2).

Una declaración de constante que declara varias constantes equivale a varias declaraciones de una sola constante con los mismos atributos, los mismos modificadores y el mismo tipo. Por ejemplo:

class A  
{  
 public const double X = 1.0, Y = 2.0, Z = 3.0;  
}

equivale a

class A  
{  
 public const double X = 1.0;  
 public const double Y = 2.0;  
 public const double Z = 3.0;  
}

Está permitido que las constantes dependan de otras constantes dentro del mismo programa, siempre y cuando las dependencias no sean circulares. El compilador evalúa automáticamente las declaraciones de constantes en el orden apropiado. En el ejemplo

class A  
{  
 public const int X = B.Z + 1;  
 public const int Y = 10;  
}

class B  
{  
 public const int Z = A.Y + 1;  
}

el compilador evalúa primero A.Y, después B.Z y finalmente A.X, que producen los valores 10, 11 y 12. Las declaraciones de constantes pueden depender de constantes de otros programas, pero dichas dependencias sólo son posibles en una dirección. Haciendo referencia al ejemplo anterior, si A y B fueran declaradas en programas independientes, sería posible que A.X dependiera de B.Z, pero entonces B.Z no podría depender simultáneamente de A.Y.

## Campos

Un campo es un miembro que representa una variable asociada a un objeto o una clase. Una declaración de campo (field-declaration) especifica uno o varios campos de un determinado tipo.

field-declaration:  
attributesopt field-modifiersopt type variable-declarators ;

field-modifiers:  
field-modifier  
field-modifiers field-modifier

field-modifier:  
new  
public  
protected  
internal  
private  
static  
readonly  
volatile

variable-declarators:  
variable-declarator  
variable-declarators , variable-declarator

variable-declarator:  
identifier  
identifier = variable-initializer

variable-initializer:  
expression  
array-initializer

Una declaración de campo (field-declaration) puede incluir un conjunto de atributos (attributes) (§17), un modificador new (§10.3.4), una combinación válida de los cuatro modificadores de acceso (§10.3.5) y un modificador static (§10.5.1). Además, una declaración de campo (field-declaration) puede incluir un modificador readonly (§10.5.2) o un modificador volatile (§10.5.3) pero no ambos. Los atributos y modificadores se aplican a todos los miembros declarados en la declaración de campo (field-declaration). Es un error que el mismo modificador aparezca varias veces en una declaración de campo.

El tipo (type) de una declaración de campo (field-declaration) especifica el tipo de los miembros que se incluyen en la declaración. El tipo va seguido de una lista de declaradores de variable (variable-declarator), cada uno de los cuales incluye un nuevo miembro. Un declarador de variable (variable-declarator) está formado por un identificador (identifier) que da nombre al miembro, que opcionalmente puede ir seguido del token “=” y de un inicializador de variable (variable-initializer) (§10.5.5) que establece el valor inicial del miembro.

El tipo (type) de un campo debe ser al menos tan accesible como el propio campo (§3.5.4).

Para obtener el valor de un campo en una expresión se utiliza un nombre simple (simple-name) (§7.6.2) o un acceso a miembro (member-access) (§7.6.4). El valor de un campo que no sea de solo lectura se modifica mediante una asignación (assignment) (§7.17). El valor de un campo que no sea de solo lectura se puede obtener y modificar utilizando operadores de incremento y decremento postfijos (§7.6.9), y operadores de incremento y decremento prefijos (§7.7.5).

Una declaración de campo que declara varios campos equivale a varias declaraciones de un único campo con los mismos atributos, los mismos modificadores y el mismo tipo. Por ejemplo:

class A  
{  
 public static int X = 1, Y, Z = 100;  
}

equivale a

class A  
{  
 public static int X = 1;  
 public static int Y;  
 public static int Z = 100;  
}

### Campos estáticos y de instancia

Cuando una declaración de campo incluye un modificador static, los campos presentados en la declaración son campos estáticos. Cuando no existe un modificador static, los campos presentados en la declaración son campos de instancia. Los campos estáticos y los campos de instancia son dos de los distintos tipos de variables (§5) compatibles con C# y en ocasiones se les llama variables estáticas y variables de instancia, respectivamente.

Un campo estático no forma parte de una instancia específica, sino que es compartido entre todas las instancias de un tipo cerrado (§4.4.2). No importa cuántas instancias de un tipo de clase cerrado se creen, nunca hay más de una copia de un campo estático para el dominio de aplicación asociado.

Por ejemplo:

class C<V>  
{  
 static int count = 0;

public C() {  
 count++;  
 }

public static int Count {  
 get { return count; }  
 }  
}

class Application  
{  
 static void Main() {  
 C<int> x1 = new C<int>();  
 Console.WriteLine(C<int>.Count); // Prints 1

C<double> x2 = new C<double>();  
 Console.WriteLine(C<int>.Count); // Prints 1

C<int> x3 = new C<int>();  
 Console.WriteLine(C<int>.Count); // Prints 2  
 }  
}

Un campo de instancia pertenece siempre a una instancia. De manera específica, cada instancia de una clase contiene un conjunto independiente de todos los campos de instancia de la clase.

Cuando se hace referencia a un campo en un acceso a miembro (member-access) (§7.6.4) con el formato E.M, si M es un campo estático, E debe denotar un tipo que contenga M, y si M es un campo de instancia, E debe denotar una instancia de un tipo que contenga M.

Las diferencias entre miembros estáticos y de instancia se tratan con más detalle en §10.3.7.

### Campos de solo lectura

Cuando una declaración de campo (field-declaration) incluye un modificador readonly, los campos presentados en la declaración son campos de solo lectura. Las asignaciones directas a campos de sólo lectura únicamente pueden formar parte de la declaración, de un constructor de instancia o bien de un constructor estático de la misma clase. (Es posible asignar varias veces un campo de sólo lectura en estos contextos). En concreto, solo se permiten asignaciones directas a un campo readonly en las siguientes situaciones:

* En el declarador de variable (variable-declarator) que introduce la variable; mediante la inclusión de un inicializador de variable (variable-initializer) en la declaración).
* Para un campo de instancia, en los constructores de instancia de la clase que contiene la declaración de campo; para un campo estático, en el constructor estático de la clase que contiene la declaración de campo. Estos son también los únicos contextos en los que es válido pasar un campo readonly como parámetro out o ref.

Se produce un error en tiempo de compilación al intentar hacer una asignación a un campo readonly o pasarlo como parámetro out o ref en cualquier otro contexto.

#### Utilizar campos de sólo lectura estáticos para constantes

Un campo static readonly es útil cuando se desea un nombre simbólico para un valor constante, pero el tipo del valor no se permite en una declaración const, o cuando el valor no se puede calcular en tiempo de compilación. En el ejemplo

public class Color  
{  
 public static readonly Color Black = new Color(0, 0, 0);  
 public static readonly Color White = new Color(255, 255, 255);  
 public static readonly Color Red = new Color(255, 0, 0);  
 public static readonly Color Green = new Color(0, 255, 0);  
 public static readonly Color Blue = new Color(0, 0, 255);

private byte red, green, blue;

public Color(byte r, byte g, byte b) {  
 red = r;  
 green = g;  
 blue = b;  
 }  
}

los miembros Black, White, Red, Green y Blue no pueden declararse como miembros const porque sus valores no se pueden calcular en tiempo de compilación. Sin embargo, produce el mismo efecto declararlos como static readonly.

#### Versiones de constantes y campos de sólo lectura estáticos

Las constantes y los campos de sólo lectura tienen diferentes semánticas para el control de versiones binarias. Cuando una expresión hace referencia a una constante, el valor de la constante se obtiene en tiempo de compilación, pero cuando una expresión hace referencia a un campo de sólo lectura, el valor del campo se obtiene en tiempo de ejecución. Considere una aplicación que conste de dos programas independientes:

using System;

namespace Program1  
{  
 public class Utils  
 {  
 public static readonly int X = 1;  
 }  
}

namespace Program2  
{  
 class Test  
 {  
 static void Main() {  
 Console.WriteLine(Program1.Utils.X);  
 }  
 }  
}

Los espacios de nombres Program1 y Program2 indican dos programas que se compilan de forma independiente. Debido a que el campo Program1.Utils.X se declara como de sólo lectura estático, el resultado de la instrucción Console.WriteLine no se conoce en tiempo de compilación, sino que se obtiene en tiempo de ejecución. Por lo tanto, si se cambia el valor de X y se recompila Program1, la instrucción Console.WriteLine genera el nuevo valor aunque no se recompile Program2. No obstante, si X hubiera sido una constante, el valor de X se habría obtenido al compilar Program2 y no se vería afectado por los cambios realizados en Program1 hasta que se volviera a compilar Program2.

### Campos volatile

Cuando una declaración de campo (field-declaration) incluye un modificador volatile, los campos presentados en la declaración son campos volátiles.

Para los campos no volátiles, las técnicas de optimización que reordenan las instrucciones pueden dar resultados inesperados e impredecibles en programas multiproceso que tengan acceso no sincronizado a campos, como los que proporciona la instrucción lock (lock-statement) (§8.12). Estas optimizaciones pueden ser realizadas por el compilador, el sistema en tiempo de ejecución o el hardware. Para campos volátiles, tales optimizaciones de reordenación están restringidas:

* La lectura de los campos volátiles se denomina lectura volátil. Una lectura volátil tiene “semántica de adquisición”: está garantizado que se produce antes de cualquier referencia a la memoria que tenga lugar con posterioridad a ella en la secuencia de la instrucción.
* La escritura de campos volátiles se denomina escritura volátil. Una escritura volátil tiene “semántica de liberación”: está garantizado que se produce después de cualquier referencia a la memoria que tenga lugar con anterioridad a la instrucción de escritura en la secuencia de la instrucción.

Estas restricciones aseguran que todos los subprocesos reconocerán las escrituras volátiles realizadas por cualquier otro subproceso en el orden en que se realizaron. No se requiere de una implementación correcta que proporcione una sola reordenación total de las escrituras volátiles desde la perspectiva de todos los subprocesos de ejecución. El tipo de un campo volátil debe ser uno de los siguientes:

* Un tipo de referencia (reference-type).
* El tipo byte, sbyte, short, ushort, int, uint, char, float, bool, System.IntPtr o System.UIntPtr.
* Un tipo enum (enum-type) que tiene un tipo base enum de byte, sbyte, short, ushort, int o uint.

El ejemplo

using System;  
using System.Threading;

class Test  
{  
 public static int result;   
 public static volatile bool finished;

static void Thread2() {  
 result = 143;   
 finished = true;   
 }

static void Main() {  
 finished = false;

// Run Thread2() in a new thread  
 new Thread(new ThreadStart(Thread2)).Start();

// Wait for Thread2 to signal that it has a result by setting  
 // finished to true.  
 for (;;) {  
 if (finished) {  
 Console.WriteLine("result = {0}", result);  
 return;  
 }  
 }  
 }  
}

genera el siguiente resultado:

result = 143

En este ejemplo, el método Main inicia un nuevo subproceso que ejecuta el método Thread2. Este método almacena un valor en un campo no volátil denominado result y, a continuación, almacena true en el campo volátil finished. El subproceso principal espera a que se establezca el campo finished en true y, a continuación, lee el campo result. Dado que se declaró finished como volatile, el subproceso principal debe leer el valor 143 del campo result. Si el campo finished no se hubiera declarado como volatile, estaría permitido que la operación de almacenamiento en result fuera visible para el subproceso principal después del almacenamiento en finished, y por tanto también lo sería que el subproceso principal leyera el valor 0 en el campo result. Declarar finished como un campo volatile impide tales incoherencias.

### Inicialización de campos

El valor inicial de un campo, tanto si se trata de un campo estático como de un campo de instancia, es el valor predeterminado (§5.2) del tipo del campo. No es posible observar el valor de un campo antes de que ocurra su inicialización predeterminada y, por lo tanto, un campo nunca es “desinicializado”. El ejemplo

using System;

class Test  
{  
 static bool b;  
 int i;

static void Main() {  
 Test t = new Test();  
 Console.WriteLine("b = {0}, i = {1}", b, t.i);  
 }  
}

produce el resultado

b = False, i = 0

porque b e i se inicializan automáticamente a valores predeterminados.

### Inicializadores de variables

Las declaraciones de campo pueden incluir los inicializadores de variables (variable-initializer). En campos estáticos, los inicializadores de variables corresponden a instrucciones de asignación que se ejecutan durante la inicialización de la clase. En campos de instancia, los inicializadores de variables corresponden a instrucciones de asignación que se ejecutan cuando se crea una instancia de la clase.

El ejemplo

using System;

class Test  
{  
 static double x = Math.Sqrt(2.0);  
 int i = 100;  
 string s = "Hello";

static void Main() {  
 Test a = new Test();  
 Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);  
 }  
}

produce el resultado

x = 1.4142135623731, i = 100, s = Hello

porque la asignación de x se produce cuando se ejecutan los inicializadores de campos estáticos y las asignaciones a i y s se producen cuando se ejecutan los inicializadores de campos de instancia.

La inicialización al valor predeterminado descrita en §10.5.4 ocurre para todos los campos, incluyendo campos que tienen inicializadores de variables. Por lo tanto, cuando se inicializa una clase, primero se inicializan todos sus campos estáticos a sus valores predeterminados y, a continuación, se ejecutan los inicializadores de campos estáticos en orden textual. Igualmente, cuando se crea una instancia de una clase, primero se inicializan todos sus campos de instancia a sus valores predeterminados y, a continuación, se ejecutan los inicializadores de campos de instancia en orden textual.

Es posible observar campos estáticos con inicializadores de variables en su estado de valor predeterminado. Sin embargo, está totalmente desaconsejado por razones de estilo. El ejemplo

using System;

class Test  
{  
 static int a = b + 1;  
 static int b = a + 1;

static void Main() {  
 Console.WriteLine("a = {0}, b = {1}", a, b);  
 }  
}

expone este comportamiento. A pesar de las definiciones circulares de a y b, el programa es correcto. Da como resultado

a = 1, b = 2

porque los campos estáticos a y b se inicializan a 0 (el valor predeterminado para int) antes de que se ejecuten sus inicializadores. Cuando se ejecuta el inicializador para a, el valor de b es cero y a se inicializa a 1. Cuando se ejecuta el inicializador para b, el valor de a ya es 1 y b se inicializa a 2.

#### Inicialización de campos estáticos

Los inicializadores de variable de un campo estático de una clase corresponden a una secuencia de asignaciones que se ejecutan en el orden textual en que aparecen en la declaración de la clase. Si en la clase existe un constructor estático (§10.12), \r \h la ejecución de los inicializadores de campo estáticos tiene lugar inmediatamente antes de la ejecución del constructor estático. En caso contrario, los inicializadores de campo estático se ejecutan en un momento que depende de la inicialización y que es anterior a la primera utilización de un campo estático de esta clase. El ejemplo

using System;

class Test   
{   
 static void Main() {  
 Console.WriteLine("{0} {1}", B.Y, A.X);  
 }

public static int F(string s) {  
 Console.WriteLine(s);  
 return 1;  
 }  
}

class A  
{  
 public static int X = Test.F("Init A");  
}

class B  
{  
 public static int Y = Test.F("Init B");  
}

podría producir el resultado:

Init A  
Init B  
1 1

o el siguiente:

Init B  
Init A  
1 1

porque la ejecución de los inicializadores de X e Y puede producirse en cualquier orden; solo es imprescindible que se produzca antes de las referencias a esos campos. No obstante, en el ejemplo

using System;

class Test  
{  
 static void Main() {  
 Console.WriteLine("{0} {1}", B.Y, A.X);  
 }

public static int F(string s) {  
 Console.WriteLine(s);  
 return 1;  
 }  
}

class A  
{  
 static A() {}

public static int X = Test.F("Init A");  
}

class B  
{  
 static B() {}

public static int Y = Test.F("Init B");  
}

el resultado debe ser:

Init B  
Init A  
1 1

porque las reglas de ejecución de los constructores estáticos (como se define en §10.12) dictan que el constructor estático de B (y por lo tanto, los inicializadores de campo estático de B) debe ejecutarse antes que el constructor estático de A y sus inicializadores de campo.

#### Inicialización de campos de instancia

Los inicializadores de variable de un campo de instancia de una clase corresponden a una secuencia de asignaciones que se ejecutan inmediatamente al entrar en cualquiera de los constructores de instancia (§10.11.1) de la clase. Los inicializadores de variable se ejecutan en el orden textual en que aparecen en la declaración de clase. El proceso de creación e inicialización de una instancia de clase se describe con más detalle en §10.11.

Un inicializador de variable para un campo de instancia no puede hacer referencia a la instancia que está creando. Por lo tanto, supone un error en tiempo de compilación que en un inicializador de variable se haga referencia a this o a un miembro de instancia mediante un nombre simple (simple-name). En el ejemplo

class A  
{  
 int x = 1;  
 int y = x + 1; // Error, reference to instance member of this  
}

el inicializador de variable para y produce un error en tiempo de compilación porque hace referencia a un miembro de la instancia que se está creando.

## Métodos

Un método es un miembro que implementa un cálculo o una acción que puede realizar un objeto o una clase. Los métodos se declaran mediante declaraciones de métodos (method-declaration):

method-declaration:  
method-header method-body

method-header:  
attributesopt method-modifiersopt partialopt return-type member-name type-parameter-listopt  
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt

method-modifiers:  
method-modifier  
method-modifiers method-modifier

method-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern  
async

return-type:  
type  
void

member-name:  
identifier  
interface-type . identifier

method-body:  
block  
;

Una declaración de método (method-declaration) puede incluir un conjunto de atributos (attributes) (§17) y una combinación válida de los cuatro modificadores de acceso (§10.3.5), los modificadores new (§10.3.4), static (§10.6.2), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6) y extern (§10.6.7).

Una declaración tiene una combinación válida de modificadores si se cumplen las siguientes condiciones:

* La declaración incluye una combinación válida de modificadores de acceso (§10.3.5).
* La declaración no incluye el mismo modificador varias veces.
* La declaración incluye como máximo uno de los modificadores siguientes: static, virtual y override.
* La declaración incluye como máximo uno de los modificadores siguientes: new y override.
* Si la declaración incluye el modificador abstract, no podrá incluir ninguno de los siguientes modificadores: static, virtual, sealed o extern.
* Si la declaración incluye el modificador private, no podrá incluir ninguno de los siguientes modificadores: virtual, override o abstract.
* Si la declaración incluye el modificador sealed, también incluirá el modificador override.
* Si la declaración incluye el modificador partial, no podrá incluir ninguno de los siguientes modificadores: new, public, protected, internal, private, virtual, sealed, override, abstract o extern.

Un método que tiene el modificador asincrónico es una función asincrónica y sigue las reglas descritas en §10.14.

El tipo de valor devuelto (return-type) de una declaración de método especifica el tipo del valor calculado y devuelto por el método. Si el método no devuelve un valor, el tipo del valor devuelto (return-type) es void. Si la declaración incluye el modificador partial, el tipo de valor devuelto también debe ser void.

El nombre de miembro (member-name) especifica el nombre del método. Salvo que el método sea una implementación de miembro de interfaz explícita (§13.4.1), el nombre de miembro (member-name) es simplemente un identificador (identifier). Para una implementación de miembro de interfaz explícita, el nombre de miembro (member-name) se compone de un tipo de interfaz (interface-type) seguido de “.” y un identificador (identifier).

La lista de parámetros de tipo (type-parameter-list) opcional especifica los parámetros de tipo del método (§10.1.3). Si se especifica una lista de parámetros de tipo (type-parameter-list), el método es un método genérico. Si el método tiene un modificador extern, no se puede especificar una lista de parámetros de tipo (type-parameter-list).

La lista de parámetros formales (formal-parameter-list) opcional especifica los parámetros del método (§10.6.1).

Las cláusulas de restricciones de parámetros de tipo (type-parameter-constraints-clauses) opcionales especifican las restricciones en parámetros de tipo individuales (§10.1.5) y solo se pueden especificar si se proporciona también una lista de parámetros de tipo (type-parameter-list), y el método no tiene un modificador override.

El tipo de valor devuelto (return-type) y cada uno de los tipos a los que se hace referencia en la lista de parámetros formales (formal-parameter-list) de un método deben tener como mínimo el mismo nivel de accesibilidad que el método (§3.5.4).

Para los métodos abstract y extern, el cuerpo del método (method-body) consiste simplemente en un punto y coma. Para los métodos partial el cuerpo del método (method-body) puede consistir en un punto y coma o en un bloque (block). Para el resto de métodos, el cuerpo del método (method-body) consiste en un bloque (block) que especifica las instrucciones a ejecutar cuando se invoca el método.

Si el cuerpo de método (method-body) consta de un punto y coma, la declaración no puede incluir el modificador async.

El nombre, la lista de parámetros de tipo y la lista de parámetros formales de un método definen la signatura (§3.6) del método. En concreto, la firma de un método se compone del nombre del método, el número de parámetros de tipo y de modificadores, y los tipos de sus parámetros formales. Con este fin, todo parámetro de tipo del método que se produce en el tipo de un parámetro formal se identifica no por su nombre, sino por su posición ordinal en la lista de argumentos de tipo del método. El tipo de valor devuelto no es parte de la firma de un método, ni los nombres de los parámetros de tipo o de los parámetros formales.

El nombre de un método debe ser diferente de los nombres del resto de miembros que no sean métodos declarados en la misma clase. Además, la signatura de un método debe ser distinta de las signaturas de todos los demás métodos declarados en la misma clase, y dos métodos declarados en la misma clase no pueden tener signaturas que solo se diferencien por ref y out.

Los parámetros de tipo (type-parameter) del método se encuentran dentro del ámbito de la declaración del método (method-declaration) y se pueden utilizar para formar tipos, en todo ese ámbito, en tipos de valores devueltos (return-type), cuerpos de método (method-body) y cláusulas de restricciones de parámetros de tipo (type-parameter-constraints-clauses), pero no en atributos (attributes).

Todos los parámetros formales y parámetros de tipo deben tener nombres diferentes.

### Parámetros de métodos

Los parámetros de un método, si existen, se declaran mediante la lista de parámetros formales (formal-parameter-list) del método.

formal-parameter-list:  
fixed-parameters  
fixed-parameters , parameter-array  
parameter-array

fixed-parameters:  
fixed-parameter  
fixed-parameters , fixed-parameter

fixed-parameter:  
attributesopt parameter-modifieropt type identifier default-argumentopt

default-argument:  
= expression

parameter-modifier:  
ref  
out  
this

parameter-array:  
attributesopt params array-type identifier

La lista de parámetros formales consiste en uno o varios parámetros separados por comas de los cuales sólo el último puede ser una matriz de parámetros (parameter-array).

Un parámetro de tipo fijado (fixed-parameter) consiste en un conjunto opcional de atributos (attributes) (§17), un modificador ref, out o this opcional, un tipo (type), un identificador (identifier) y un argumento predeterminado (default-argument) opcional. Cada parámetro de tipo fijado (fixed-parameter) declara un parámetro con el tipo y nombre especificados. El modificador this designa el método como un método de extensión y solo se permite en el primer parámetro de un método estático. Los métodos de extensión se describen con más detalle en la sección §10.6.9.

Un parámetro fijo (fixed-parameter) con un argumento predeterminado (default-argument) se conoce como parámetro opcional, mientras que un parámetro fijo sin argumento predeterminado es un parámetro necesario. Un parámetro necesario no puede aparecer después de un parámetro opcional en una lista de parámetros formales (formal-parameter-list).

Un parámetro ref o out no puede tener un argumento predeterminado (default-argument). La expresión de un argumento predeterminado (default-argument) tiene que ser una de las siguientes:

* una expresión constante (constant-expression)
* una expresión con la forma new S() donde S es un tipo de valor
* una expresión con la forma default(S) donde S es un tipo de valor

La expresión debe poder convertirse implícitamente en el tipo del parámetro mediante una conversión de identidad o una conversión que acepte valores NULL.

Si los parámetros opcionales aparecen al implementar una declaración de método opcional (§10.2.7), una implementación de miembro de interfaz explícito (§13.4.1) o una declaración de indizador de parámetro único (§10.9), el compilador debería emitir una advertencia, porque estos miembros no se pueden invocar nunca de un modo que permita la omisión de argumentos.

Una matriz de parámetros (parameter-array) consiste en un conjunto opcional de atributos (attributes) (§17), un modificador params, un tipo de matriz (array-type) y un identificador (identifier). Una matriz de parámetros declara un único parámetro con el tipo de matriz y nombre especificados. El tipo de matriz (array-type) de una matriz de parámetros debe ser unidimensional (§12.1). Cuando se llama a un método, la matriz de parámetros permite especificar un único argumento del tipo de matriz dado, o cero o varios argumentos del tipo de elemento de matriz. Las matrices de parámetros se describen más detalladamente en §10.6.1.4.

Una matriz de parámetros (parameter-array) puede aparecer después de un parámetro opcional, pero no puede tener un valor predeterminado: la omisión de los argumentos de una matriz de parámetros daría como resultado la creación de una matriz vacía.

En el siguiente ejemplo se muestran distintos tipos de parámetros:

public void M(  
 ref int i,  
 decimal d,  
 bool b = false,  
 bool? n = false,  
 string s = "Hello",  
 object o = null,  
 T t = default(T),  
 params int[] a  
) { }

En la lista de parámetros formales (formal-parameter-list) de M, i es un parámetro ref necesario, d es un parámetro de valor necesario, b, s, o y t son opcionales y a es una matriz de parámetros.

Una declaración de método crea un espacio de declaración independiente para los parámetros, los parámetros de tipo y las variables locales. La lista de parámetros de tipo y la lista de parámetros formales del método, así como las declaraciones de variables locales en el bloque (block) del método introducen los nombres en ese espacio de declaración. Que dos miembros de un espacio de declaración del método tengan el mismo nombre supone un error. Que el espacio de declaración del método y el espacio de declaración de la variable local de un espacio de declaración anidado contengan elementos con el mismo nombre supone un error.

La invocación de un método (§7.6.5.1) crea una copia, específica de esa invocación, de los parámetros formales y de las variables locales del método, y la lista de argumentos de la invocación asigna valores o referencias a variables a los parámetros formales recién creados. Dentro del bloque (block) de un método, se puede hacer referencia a los parámetros formales mediante sus identificadores en expresiones de nombre simple (simple-name) (§7.6.2).

Existen cuatro tipos de parámetros formales:

* Parámetros de valor, que se declaran sin modificadores.
* Parámetros de referencia, que se declaran con el modificador ref.
* Parámetros de salida, que se declaran con el modificador out.
* Matrices de parámetros, que se declaran con el modificador params.

Como se describe en §3.6, los modificadores ref y out forman parte de la signatura de un método, pero no el modificador params.

#### Parámetros de valor

Un parámetro declarado sin modificadores es un parámetro de valor. Un parámetro de valor corresponde a una variable local que obtiene su valor inicial del correspondiente argumento proporcionado en la invocación del método.

Cuando un parámetro formal es un parámetro de valor, el argumento correspondiente de la invocación del método debe ser una expresión que sea convertible implícitamente (§6.1) al tipo del parámetro formal.

Un método puede asignar nuevos valores a un parámetro de valor. Tales asignaciones sólo afectan a la ubicación de almacenamiento local representada por el parámetro de valor, no tienen ningún efecto sobre el argumento real definido en la invocación del método.

#### Parámetros de referencia

Un parámetro declarado con un modificador ref es un parámetro de referencia. A diferencia de un parámetro de valor, un parámetro de referencia no crea una nueva ubicación de almacenamiento. En lugar de ello, un parámetro de referencia representa la misma ubicación de almacenamiento que la variable especificada como argumento en la invocación del método.

Cuando un parámetro formal es un parámetro de referencia, el argumento correspondiente de la invocación del método se debe componer de la palabra clave ref seguida de una referencia de variable (variable-reference) (§5.3.3) del mismo tipo del parámetro formal. Una variable debe estar asignada de manera definitiva antes de que se pueda pasar como parámetro de referencia.

Dentro de un método, un parámetro de referencia siempre se considera asignado de manera definitiva.

Un método declarado como iterador (§10.14) no puede tener parámetros de referencia.

El ejemplo

using System;

class Test  
{  
 static void Swap(ref int x, ref int y) {  
 int temp = x;  
 x = y;  
 y = temp;  
 }

static void Main() {  
 int i = 1, j = 2;  
 Swap(ref i, ref j);  
 Console.WriteLine("i = {0}, j = {1}", i, j);  
 }  
}

produce el resultado

i = 2, j = 1

Para la invocación de Swap en Main, x representa a i e y representa a j. Por lo tanto, la invocación tiene el efecto de intercambiar los valores de i y de j.

En un método que toma parámetros de referencia, varios nombres pueden representar la misma ubicación de almacenamiento. En el ejemplo

class A  
{  
 string s;

void F(ref string a, ref string b) {  
 s = "One";  
 a = "Two";  
 b = "Three";  
 }

void G() {  
 F(ref s, ref s);  
 }  
}

la invocación de F en G pasa una referencia a s para a y b. De esta forma, en esa invocación, los nombres s, a y b hacen referencia todos ellos a la misma ubicación de almacenamiento, y las tres asignaciones modifican el campo de instancia s.

#### Parámetros de salida

Un parámetro de salida es un parámetro que se declara con un modificador out. Al igual que un parámetro de referencia, un parámetro de salida no crea una nueva ubicación de almacenamiento. En lugar de ello, un parámetro de salida representa la misma ubicación de almacenamiento que la variable especificada como argumento en la invocación del método.

Cuando un parámetro formal es un parámetro de salida, el argumento correspondiente de la invocación del método se debe componer de la palabra clave out seguida de una referencia de variable (variable-reference) (§5.3.3) del mismo tipo del parámetro formal. Una variable no necesita estar asignada de manera definitiva antes de ser pasada como parámetro de salida, pero si se hace un seguimiento de una invocación donde se pasó una variable como parámetro de salida, la variable se considera asignada de manera definitiva.

Dentro de un método, de manera similar a una variable local, un parámetro de salida se considera inicialmente no asignado y debe ser asignado de manera definitiva antes de utilizar su valor.

Cada parámetro de salida de un método debe estar asignado de manera definitiva antes de que el método devuelva la llamada.

Un método declarado como método parcial (§10.2.7) o un iterador (§10.14) no puede tener parámetros de salida.

Los parámetros de salida se utilizan normalmente en métodos que devuelven varios valores. Por ejemplo:

using System;

class Test  
{  
 static void SplitPath(string path, out string dir, out string name) {  
 int i = path.Length;  
 while (i > 0) {  
 char ch = path[i – 1];  
 if (ch == '\\' || ch == '/' || ch == ':') break;  
 i--;  
 }  
 dir = path.Substring(0, i);  
 name = path.Substring(i);  
 }

static void Main() {  
 string dir, name;  
 SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);  
 Console.WriteLine(dir);  
 Console.WriteLine(name);  
 }  
}

El ejemplo produce el resultado:

c:\Windows\System\  
hello.txt

Observe que las variables dir y name pueden estar sin asignar antes de pasar a SplitPath, y que se consideran asignadas de manera definitiva siguiendo la llamada.

#### Matrices de parámetros

Un parámetro declarado con un modificador params es una matriz de parámetros. Si una lista de parámetros formales incluye una matriz de parámetros, ésta debe ser el último parámetro de la lista y debe ser de tipo unidimensional. Por ejemplo, los tipos string[] y string[][] se pueden utilizar como tipo de una matriz de parámetros, pero el tipo string[,] no. No se puede combinar el modificador params con los modificadores ref y out.

Una matriz de parámetros permite especificar argumentos en cualquiera de las dos formas de una invocación de método:

* El argumento especificado para una matriz de parámetros puede ser una única expresión que sea convertible implícitamente (§6.1) al tipo de la matriz de parámetros. En este caso, la matriz de parámetros actúa exactamente como un parámetro de valor.
* Alternativamente, la invocación puede especificar cero o más argumentos para la matriz de parámetros, donde cada argumento es una expresión de un tipo implícitamente convertible (§6.1) al tipo de elemento de la matriz de parámetros. En este caso, la invocación crea una instancia del tipo de la matriz de parámetros con una longitud correspondiente al número de argumentos, inicializa los elementos de la instancia de matriz con los valores de los argumentos especificados y utiliza la instancia de matriz recién creada como argumento real.

Excepto en lo que se refiere a permitir un número variable de argumentos en una invocación, una matriz de parámetros equivale exactamente a un parámetro de valor (§10.6.1.1) del mismo tipo.

El ejemplo

using System;

class Test  
{  
 static void F(params int[] args) {  
 Console.Write("Array contains {0} elements:", args.Length);  
 foreach (int i in args)   
 Console.Write(" {0}", i);  
 Console.WriteLine();  
 }

static void Main() {  
 int[] arr = {1, 2, 3};  
 F(arr);  
 F(10, 20, 30, 40);  
 F();  
 }  
}

produce el resultado

Array contains 3 elements: 1 2 3  
Array contains 4 elements: 10 20 30 40  
Array contains 0 elements:

La primera invocación de F simplemente pasa la matriz a como un parámetro de valor. La segunda invocación de F crea automáticamente una matriz int[] de cuatro elementos con los valores de elemento especificados y pasa esa instancia de matriz como parámetro de valor. Del mismo modo, la tercera invocación de F crea una matriz int[] de cero elementos y pasa esa instancia como parámetro de valor. La segunda y tercera invocación equivalen exactamente a escribir:

F(new int[] {10, 20, 30, 40});  
F(new int[] {});

Cuando se realiza la resolución de sobrecarga, un método con una matriz de parámetros puede ser aplicable en su forma normal o en su forma expandida (§7.5.3.1). La forma expandida de un método está disponible solo si no es aplicable la forma normal del método y únicamente cuando no esté declarado ya en el mismo tipo un método aplicable con la misma signatura que la forma expandida.

El ejemplo

using System;

class Test  
{  
 static void F(params object[] a) {  
 Console.WriteLine("F(object[])");  
 }

static void F() {  
 Console.WriteLine("F()");  
 }

static void F(object a0, object a1) {  
 Console.WriteLine("F(object,object)");  
 }

static void Main() {  
 F();  
 F(1);  
 F(1, 2);  
 F(1, 2, 3);  
 F(1, 2, 3, 4);  
 }  
}

produce el resultado

F();  
F(object[]);  
F(object,object);  
F(object[]);  
F(object[]);

En el ejemplo, dos de las posibles formas expandidas del método con una matriz de parámetros ya están incluidas en la clase como métodos regulares. Por lo tanto, estas formas expandidas no son consideradas cuando se realiza la resolución de sobrecarga, y las invocaciones del primer y tercer método seleccionan los métodos regulares. Cuando una clase declara un método con una matriz de parámetros, se suelen incluir también algunas de las formas expandidas como métodos regulares. Así es posible evitar la asignación de una instancia de matriz que se produce cuando se invoca una forma expandida de un método con una matriz de parámetros.

Cuando el tipo de una matriz de parámetros es object[], puede producirse una ambigüedad entre la forma normal del método y la forma expandida para un único parámetro object. La razón de la ambigüedad es que un tipo object[] es implícitamente convertible al tipo object. La ambigüedad no representa un problema, ya que se puede resolver insertando una conversión si es necesario.

El ejemplo

using System;

class Test  
{  
 static void F(params object[] args) {  
 foreach (object o in args) {  
 Console.Write(o.GetType().FullName);  
 Console.Write(" ");  
 }  
 Console.WriteLine();  
 }

static void Main() {  
 object[] a = {1, "Hello", 123.456};  
 object o = a;  
 F(a);  
 F((object)a);  
 F(o);  
 F((object[])o);  
 }  
}

produce el resultado

System.Int32 System.String System.Double  
System.Object[]  
System.Object[]  
System.Int32 System.String System.Double

En la primera y última invocación de F, la forma normal de F es aplicable porque existe una conversión implícita del tipo del argumento al tipo del parámetro (ambos son de tipo object[]). Por lo tanto, la resolución de sobrecarga selecciona la forma normal de F y el argumento se pasa como un parámetro de valor regular. En las invocaciones segunda y tercera, la forma normal de F no es aplicable porque no existe una conversión implícita del tipo del argumento al tipo del parámetro (el tipo object no se puede convertir implícitamente al tipo object[]). Sin embargo, la forma expandida de F es aplicable, y por ello la selecciona la resolución de sobrecarga. Como resultado, la invocación crea una matriz object[] de un elemento, y dicho elemento de la matriz se inicializa con el valor del argumento especificado (que a su vez es una referencia a un object[]).

### Métodos estáticos y de instancia

Cuando una declaración de método incluye un modificador static, se dice que el método es estático. Si no existe un modificador static, se dice que es un método de instancia.

Un método estático no opera en una instancia específica, y se produce un error en tiempo de compilación al hacer referencia a this en un método estático.

Un método de instancia opera en una determinada instancia de una clase y es posible tener acceso a dicha instancia con this (§7.6.7).

Cuando se hace referencia a un método en un acceso a miembro (member-access) (§7.6.4) con el formato E.M, si M es un método estático, E debe denotar un tipo que contenga M, y si M es un método de instancia, E debe denotar una instancia de un tipo que contenga M.

Las diferencias entre miembros estáticos y de instancia se tratan con más detalle en §10.3.7.

### Métodos virtuales

Cuando una declaración de método de instancia incluye un modificador virtual, se dice que ese método es un método virtual. Si no existe un modificador virtual, se dice que el método es un método no virtual.

La implementación de un método no virtual es invariable. La implementación es la misma tanto si se invoca un método en una instancia de la clase en la que se declaró o en una instancia de una clase derivada. En cambio, la implementación de un método virtual se puede sustituir por clases derivadas. El proceso de sustitución de la implementación de un método virtual heredado es conocido como invalidación del método (§10.6.4).

En la invocación de un método virtual, el tipo en tiempo de ejecución de la instancia para la que tiene lugar la invocación determina la implementación del método real a invocar. Cuando se invoca a un método no virtual, el factor determinante es el tipo en tiempo de compilación de la instancia. Concretamente, cuando se invoca un método denominado N con una lista de argumentos A en una instancia con un tipo C en tiempo de compilación y un tipo R en tiempo de ejecución (donde R es C o una clase derivada de C), la invocación se procesa de la forma siguiente:

* En primer lugar, la resolución de sobrecarga se aplica a C, N y A para seleccionar un método específico M del conjunto de métodos declarados en y heredados por C. Este proceso se describe en §7.6.5.1.
* A continuación, si M es un método no virtual, se invoca M.
* Si no, M es un método virtual, y se invoca la implementación más derivada de M con respecto a R.

Para cada método virtual que se ha declarado en una clase o heredado por ella, existe una implementación más derivada del método con respecto a esa clase. La implementación más derivada de un método virtual M con respecto a una clase R está determinada de la siguiente manera:

* Si R contiene la declaración virtual introductora de M, esta es la implementación más derivada de M.
* En caso contrario, si R contiene una invalidación de override de M, esta es la implementación más derivada de M.
* En caso contrario, la implementación más derivada de M con respecto a R es la misma que la implementación más derivada de M con respecto a la clase base directa de R.

En el siguiente ejemplo se ilustran las diferencias entre los métodos virtuales y los métodos no virtuales:

using System;

class A  
{  
 public void F() { Console.WriteLine("A.F"); }

public virtual void G() { Console.WriteLine("A.G"); }  
}

class B: A  
{  
 new public void F() { Console.WriteLine("B.F"); }

public override void G() { Console.WriteLine("B.G"); }  
}

class Test  
{  
 static void Main() {  
 B b = new B();  
 A a = b;  
 a.F();  
 b.F();  
 a.G();  
 b.G();  
 }  
}

En el ejemplo, A introduce un método no virtual F y un método virtual G. La clase B introduce un nuevo método no virtual F, con lo cual oculta el método F heredado y además reemplaza el método G heredado. El ejemplo produce el resultado:

A.F  
B.F  
B.G  
B.G

Observe que la instrucción a.G() llama a B.G, no a A.G. Esto se debe a que el tipo en tiempo de ejecución de la instancia (que es B), y no el tipo en tiempo de compilación (que es A), determina la implementación del método real que se invoca.

Debido a que se permite que los métodos oculten métodos heredados, es posible que una clase contenga varios métodos virtuales con la misma signatura. Esto no presenta un problema de ambigüedad puesto que todos los métodos, salvo el más derivado, están ocultos. En el ejemplo

using System;

class A  
{  
 public virtual void F() { Console.WriteLine("A.F"); }  
}

class B: A  
{  
 public override void F() { Console.WriteLine("B.F"); }  
}

class C: B  
{  
 new public virtual void F() { Console.WriteLine("C.F"); }  
}

class D: C  
{  
 public override void F() { Console.WriteLine("D.F"); }  
}

class Test  
{  
 static void Main() {  
 D d = new D();  
 A a = d;  
 B b = d;  
 C c = d;  
 a.F();  
 b.F();  
 c.F();  
 d.F();  
 }  
}

las clases C y D contienen dos métodos virtuales con la misma signatura: el que introduce A y el que introduce C. El método introducido por C oculta el método que se hereda de A. De este modo, la declaración de invalidación en D reemplaza el método introducido por C, y no es posible que D reemplace el método introducido por A. El ejemplo produce el resultado:

B.F  
B.F  
D.F  
D.F

Observe que es posible invocar el método virtual oculto mediante el acceso a una instancia de D a través de un tipo menos derivado en el que el método no está oculto.

### Métodos de reemplazo

Cuando una declaración de método de instancia incluye un modificador override, se dice que el método es un método de invalidación. Un método de invalidación invalida un método virtual heredado con la misma firma. Mientras que una declaración de método virtual introduce un método nuevo, una declaración de método de invalidación especializa un método virtual heredado existente proporcionando una nueva implementación de ese método.

El método invalidado por una declaración override se conoce como método base invalidado. Para un método de invalidación M declarado en una clase C, el método base invalidado se determina examinando cada clase base de C, comenzando con la clase base directa de C y continuando con cada una de las sucesivas clases base directas hasta que, en un tipo de clase base dado se encuentre al menos un método accesible que tenga la misma signatura que M después de la sustitución de los argumentos de tipo. Para encontrar el método base invalidado, un método se considera accesible si es public, protected, protected internal o internal y está declarado en el mismo programa que C.

Se produce un error en tiempo de compilación a menos que se cumplan todas las condiciones siguientes en una declaración de invalidación:

* Se puede encontrar un método base invalidado como se ha descrito anteriormente.
* Existe exactamente ese método base invalidado. Esta restricción tiene efecto sólo si el tipo de la clase base es un tipo construido donde la sustitución de argumentos de tipo hace que las firmas de los dos métodos sean idénticas.
* El método base invalidado es un método virtual, abstracto o de invalidación. En otras palabras, el método base invalidado no puede ser estático ni no virtual.
* El método base invalidado no es un método sellado.
* La declaración de invalidación y el método base invalidado tienen el mismo tipo de valor devuelto.
* La declaración de invalidación y el método base invalidado tienen la misma accesibilidad declarada. Es decir, una declaración de invalidación no puede cambiar la accesibilidad del método virtual. Sin embargo, si el método base invalidado es protected (protegido) o internal (interno) y está declarado en un ensamblado diferente del ensamblado que contiene el método de invalidación, se debe proteger la accesibilidad declarada del método de invalidación.
* La declaración de invalidación no especifica cláusulas de restricciones de parámetros de tipo. En lugar de eso, se heredan las restricciones del método base de invalidación. Observe que las restricciones que son parámetros de tipo en el método invalidado pueden reemplazarse por argumentos de tipo en la restricción heredada. Esto puede llevar a restricciones que no son válidas cuando se especifican explícitamente, como los tipos de valor o los tipos sealed.

En el siguiente ejemplo se muestra el funcionamiento de las reglas de invalidación para clases genéricas:

abstract class C<T>  
{  
 public virtual T F() {...}

public virtual C<T> G() {...}

public virtual void H(C<T> x) {...}  
}

class D: C<string>  
{  
 public override string F() {...} // Ok

public override C<string> G() {...} // Ok

public override void H(C<T> x) {...} // Error, should be C<string>  
}

class E<T,U>: C<U>  
{  
 public override U F() {...} // Ok

public override C<U> G() {...} // Ok

public override void H(C<T> x) {...} // Error, should be C<U>  
}

Una declaración de invalidación puede obtener acceso al método base invalidado mediante un acceso base (base-access) (§7.6.8). En el ejemplo

class A  
{  
 int x;

public virtual void PrintFields() {  
 Console.WriteLine("x = {0}", x);  
 }  
}

class B: A  
{  
 int y;

public override void PrintFields() {  
 base.PrintFields();  
 Console.WriteLine("y = {0}", y);  
 }  
}

la invocación de base.PrintFields() en B llama al método PrintFields declarado en A. Un acceso base (base-access) deshabilita el mecanismo de llamada virtual y simplemente trata el método base como un método no virtual. Si la invocación de B se hubiera escrito ((A)this).PrintFields(), llamaría de forma recursiva al método PrintFields declarado en B, no el declarado en A ya que PrintFields es virtual y el tipo en tiempo de ejecución de ((A)this) es B.

Solo cuando un método incluye un modificador override puede invalidar otro método. En los demás casos, un método con la misma firma que un método heredado simplemente oculta el método heredado. En el ejemplo

class A  
{  
 public virtual void F() {}  
}

class B: A  
{  
 public virtual void F() {} // Warning, hiding inherited F()  
}

el método F de B no incluye un modificador override y, por tanto, no reemplaza el método F en A. En realidad, el método F de B oculta el método en A y se emite una advertencia porque la declaración no incluye un modificador new.

En el ejemplo

class A  
{  
 public virtual void F() {}  
}

class B: A  
{  
 new private void F() {} // Hides A.F within body of B  
}

class C: B  
{  
 public override void F() {} // Ok, overrides A.F  
}

el método F en B oculta el método F virtual heredado de A. Como el nuevo método F de B tiene acceso privado, su ámbito sólo incluye el cuerpo de la clase de B y no se extiende hacia C. Por tanto, se permite la declaración de F en C para reemplazar el método F que ha heredado de A.

### Métodos sellados

Cuando una declaración de método de instancia incluye un modificador sealed, se dice que el método es un método sellado (sealed). Si una declaración de método de instancia incluye el modificador sealed, también debe incluir el modificador override. El uso del modificador sealed impide que una clase derivada siga invalidando el método.

En el ejemplo

using System;

class A  
{  
 public virtual void F() {  
 Console.WriteLine("A.F");  
 }

public virtual void G() {  
 Console.WriteLine("A.G");  
 }  
}

class B: A  
{  
 sealed override public void F() {  
 Console.WriteLine("B.F");  
 }

override public void G() {  
 Console.WriteLine("B.G");  
 }   
}

class C: B  
{  
 override public void G() {  
 Console.WriteLine("C.G");  
 }   
}

la clase B proporciona dos métodos de invalidación: un método F que tiene el modificador sealed y un método G que no lo tiene. El uso en B del modificador (modifier) sealed impide que C siga invalidando F.

### Métodos abstractos

Cuando una declaración de método de instancia incluye un modificador abstract, se dice que el método es un método abstracto. Aunque un método abstracto es también implícitamente un método virtual, no puede tener el modificador virtual.

Una declaración de método abstracto introduce un nuevo método virtual pero no proporciona una implementación del método. En cambio, es necesario que las clases derivadas no abstractas proporcionen su propia implementación mediante la invalidación del método. Debido a que un método abstracto no proporciona una implementación real, el cuerpo del método (method-body) de un método abstracto consiste simplemente en un punto y coma.

Las declaraciones de métodos abstractos sólo se permiten en clases abstractas (§10.1.1.1).

En el ejemplo

public abstract class Shape  
{  
 public abstract void Paint(Graphics g, Rectangle r);  
}

public class Ellipse: Shape  
{  
 public override void Paint(Graphics g, Rectangle r) {  
 g.DrawEllipse(r);  
 }  
}

public class Box: Shape  
{  
 public override void Paint(Graphics g, Rectangle r) {  
 g.DrawRect(r);  
 }  
}

la clase Shape define la noción abstracta de un objeto de forma geométrica que puede dibujarse a sí mismo. El método Paint es abstracto porque no existe una implementación predeterminada significativa. Las clases Ellipse y Box son implementaciones Shape concretas. Ya que estas clases no son abstractas, son necesarias para invalidar el método Paint y proporcionar una implementación real.

Supone un error en tiempo de compilación que un acceso base (base-access) (§7.6.8) haga referencia a un método abstracto. En el ejemplo

abstract class A  
{  
 public abstract void F();  
}

class B: A  
{  
 public override void F() {  
 base.F(); // Error, base.F is abstract  
 }  
}

se produce un error en tiempo de compilación en la llamada base.F() debido a que hace referencia a un método abstracto.

Una declaración de método abstracto puede invalidar un método virtual. Esto permite que una clase abstracta fuerce una nueva implementación del método en las clases derivadas, e impide la disponibilidad de la implementación original del método. En el ejemplo

using System;

class A  
{  
 public virtual void F() {  
 Console.WriteLine("A.F");  
 }  
}

abstract class B: A  
{  
 public abstract override void F();  
}

class C: B  
{  
 public override void F() {  
 Console.WriteLine("C.F");  
 }  
}

la clase A declara un método virtual, la clase B invalida este método con un método abstracto y la clase C invalida el método abstracto para proporcionar su propia implementación.

### Métodos externos

Cuando una declaración de método incluye un modificador extern, se dice que el método es un método externo. Los métodos externos se implementan externamente, utilizando generalmente un lenguaje distinto de C#. Debido a que la declaración de un método externo no proporciona una implementación real, el cuerpo del método (method-body) de un método externo consiste simplemente en un punto y coma. Un método externo puede no ser genérico.

El modificador extern se suele utilizar junto con el atributo DllImport (§17.5.1), permitiendo implementar los métodos externos mediante Bibliotecas de vínculos dinámicos (.dll). El entorno de ejecución puede ser compatible con otros mecanismos que permitan las implementaciones de métodos externos.

Cuando un método externo incluye un atributo DllImport, la declaración del método también debe incluir el modificador static. En este ejemplo se demuestra el uso del modificador extern y del atributo DllImport:

using System.Text;  
using System.Security.Permissions;  
using System.Runtime.InteropServices;

class Path  
{  
 [DllImport("kernel32", SetLastError=true)]  
 static extern bool CreateDirectory(string name, SecurityAttribute sa);

[DllImport("kernel32", SetLastError=true)]  
 static extern bool RemoveDirectory(string name);

[DllImport("kernel32", SetLastError=true)]  
 static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

[DllImport("kernel32", SetLastError=true)]  
 static extern bool SetCurrentDirectory(string name);  
}

### Métodos parciales

Cuando una declaración de método incluye un modificador partial, se dice que el método es un método parcial. Los métodos parciales solo se pueden declarar como tipos parciales (§10.2), y están sujetos a un número de restricciones. Los métodos parciales se describen con más detalle en la sección §10.2.7.

### Métodos de extensión

Cuando el primer parámetro de un método de instancia incluye un modificador this, se dice que el método es un método de extensión. Los métodos de extensión sólo se pueden declarar en clases estáticas no anidadas y no genéricas. El primer parámetro de un método de extensión no puede tener ningún modificador que no sea this, y el tipo de parámetro no puede ser un tipo de puntero.

En el siguiente ejemplo se muestra una clase estática que declara dos métodos de extensión:

public static class Extensions  
{  
 public static int ToInt32(this string s) {  
 return Int32.Parse(s);  
 }

public static T[] Slice<T>(this T[] source, int index, int count) {  
 if (index < 0 || count < 0 || source.Length – index < count)  
 throw new ArgumentException();  
 T[] result = new T[count];  
 Array.Copy(source, index, result, 0, count);  
 return result;  
 }  
}

Un método de extensión es un método estático regular. Además, dentro del ámbito de su clase estática envolvente, es posible invocar un método de extensión mediante una sintaxis de invocación de métodos de instancia (§7.6.5.2), utilizando la expresión del receptor como primer argumento.

El siguiente programa utiliza los métodos de extensión declarados antes:

static class Program  
{  
 static void Main() {  
 string[] strings = { "1", "22", "333", "4444" };  
 foreach (string s in strings.Slice(1, 2)) {  
 Console.WriteLine(s.ToInt32());  
 }  
 }  
}

El método Slice está disponible en string[], y el método ToInt32 está disponible en string (cadena), porque se han declarado como métodos de extensión. El significado del programa es igual que el que sigue, utilizando llamadas a métodos estáticos ordinarios:

static class Program  
{  
 static void Main() {  
 string[] strings = { "1", "22", "333", "4444" };  
 foreach (string s in Extensions.Slice(strings, 1, 2)) {  
 Console.WriteLine(Extensions.ToInt32(s));  
 }  
 }  
}

### Cuerpo del método

El cuerpo del método (method-body) de una declaración de método consiste en un bloque (block) o en un punto y coma.

Las declaraciones de métodos abstractos y externos no proporcionan una implementación del método, por lo que los cuerpos de estos métodos son simplemente un punto y coma. Para el resto de métodos, el cuerpo del método es un bloque (§8.2) que contiene las instrucciones que se ejecutan cuando se invoca el método.

El tipo de resultado de un método es void si el tipo devuelto es void o si el método es asincrónico y el tipo devuelto es System.Threading.Tasks.Task. De lo contrario, el tipo de resultado de un método no asincrónico es su tipo devuelto, y el tipo de resultado de un método asincrónico con el tipo devuelto System.Threading.Tasks.Task<*T*> es *T*.

Cuando el tipo de valor devuelto por un método es void, no está permitido que las instrucciones return (§8.9.4) del cuerpo del método especifiquen una expresión. Si la ejecución del cuerpo del método de un método void se completa normalmente (es decir, el control llega al final del cuerpo del método), el método regresa al llamador actual.

Cuando el tipo de resultado de un método no es void, cada instrucción return del cuerpo del método debe especificar una expresión convertible implícitamente al tipo de resultado. El extremo del cuerpo de un método que devuelve un valor no debe ser alcanzable. En otras palabras, en los métodos que devuelven valores no está permitido que el control llegue al final del cuerpo del método.

En el ejemplo

class A  
{  
 public int F() {} // Error, return value required

public int G() {  
 return 1;  
 }

public int H(bool b) {  
 if (b) {  
 return 1;  
 }  
 else {  
 return 0;  
 }  
 }  
}

el método F que devuelve un valor produce un error en tiempo de compilación, porque el control puede alcanzar el final del cuerpo del método. Los métodos G y H son correctos porque todos los caminos de ejecución posibles terminan en una instrucción return que especifica un valor a devolver.

### Sobrecarga de métodos

Las reglas de resolución de sobrecarga de métodos se describen en §7.5.2.

## Propiedades

Una propiedad es un miembro que proporciona acceso a una característica de un objeto o una clase. Los ejemplos de propiedades incluyen la longitud de una cadena, el tamaño de una fuente, el título de una ventana, el nombre de un cliente, etc. Las propiedades son una extensión natural de los campos; ambos son miembros denominados con tipos asociados, y la sintaxis que se utiliza para el acceso a campos y propiedades es la misma. No obstante, a diferencia de los campos, las propiedades no denotan ubicaciones de almacenamiento. Las propiedades tienen descriptores de acceso que especifican las instrucciones que deben ejecutarse para leer o escribir sus valores. Las propiedades proporcionan un mecanismo para asociar acciones a la lectura y escritura de los atributos de un objeto. Además, permiten el cálculo de dichos atributos.

Las propiedades se declaran mediante declaraciones de propiedad (property-declaration):

property-declaration:  
attributesopt property-modifiersopt type member-name { accessor-declarations }

property-modifiers:  
property-modifier  
property-modifiers property-modifier

property-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

member-name:  
identifier  
interface-type . identifier

Una declaración de propiedad (property-declaration) puede incluir un conjunto de atributos (attributes) (§17) y una combinación válida de los cuatro modificadores de acceso (§10.3.5), los modificadores new (§10.3.4), static (§10.6.2), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6) y extern (§10.6.7).

Las declaraciones de propiedad están sujetas a las mismas reglas que las declaraciones de método (§10.6) en lo que respecta a las combinaciones válidas de modificadores.

El tipo (type) de una declaración de propiedad especifica el tipo de la propiedad introducida en la declaración, y el nombre de miembro (member-name) especifica el nombre de la propiedad. Salvo que la propiedad sea una implementación de miembro de interfaz explícita, el nombre de miembro (member-name) es simplemente un identificador (identifier). Para una implementación de miembro de interfaz explícita (§13.4.1), el nombre de miembro (member-name) se compone de un tipo de interfaz (interface-type) seguido de “.” y un identificador (identifier).

El tipo (type) de una propiedad debe ser al menos tan accesible como la misma propiedad (§3.5.4).

Las declaraciones de descriptores de acceso (accessor-declarations), que deben estar encerradas entre los tokens “{” y “}”, declaran los descriptores de acceso (§10.7.2) de la propiedad. Los descriptores de acceso especifican las instrucciones ejecutables asociadas a la lectura y escritura de la propiedad.

Aunque la sintaxis para tener acceso a una propiedad es idéntica a la de un campo, una propiedad no está clasificada como variable. Por tanto, no se puede transferir una propiedad como un argumento ref u out.

Cuando una declaración de propiedad incluye un modificador extern, se dice que la propiedad es una propiedad externa. Debido a que la declaración de una propiedad externa no proporciona una implementación real, cada una de sus declaraciones de descriptores de acceso (accessor-declarations) consiste en un punto y coma.

### Propiedades estáticas y de instancia

Cuando una declaración de propiedad incluye un modificador static, se dice que la propiedad es una propiedad estática. Si no existe un modificador static, se dice que la propiedad es una propiedad de instancia.

Una propiedad estática no está asociada a una instancia específica y supone un error en tiempo de compilación hacer referencia a this en los descriptores de acceso de una propiedad estática.

Una propiedad de instancia está asociada a una determinada instancia de una clase y es posible tener acceso a dicha instancia con this (§7.6.7) en los descriptores de acceso de la propiedad.

Cuando se hace referencia a una propiedad en un acceso a miembro (member-access) (§7.6.4) con el formato E.M, si M es una propiedad estática, E debe denotar un tipo que contenga M, y si M es una propiedad de instancia, E debe denotar una instancia de un tipo que contenga M.

Las diferencias entre miembros estáticos y de instancia se tratan con más detalle en §10.3.7.

### Descriptores de acceso

Las declaraciones de descriptores de acceso (accessor-declarations) de una propiedad especifican las instrucciones ejecutables asociadas a la lectura y escritura de la propiedad.

accessor-declarations:  
get-accessor-declaration set-accessor-declarationopt  
set-accessor-declaration get-accessor-declarationopt

get-accessor-declaration:  
attributesopt accessor-modifieropt  get accessor-body

set-accessor-declaration:  
attributesopt accessor-modifieropt set accessor-body

accessor-modifier:  
protected  
internal  
private  
protected internal  
internal protected

accessor-body:  
block  
;

Las declaraciones de descriptores de acceso se componen de una declaración de descriptor de acceso get (get-accessor-declaration), en una declaración de descriptor de acceso set (set-accessor-declaration) o en ambas. Cada declaración de descriptor de acceso consta del token get o set seguido de un modificador de descriptor de acceso (accessor-modifier) opcional y un cuerpo de descriptor de acceso (accessor-body).

El uso de los modificadores de descriptores de acceso (accessor-modifier) está controlado por las siguientes restricciones:

* Es posible que un modificador de descriptor de acceso (accessor-modifier) no se utilice en una interfaz ni en una implementación de miembros de interfaz explícitos.
* Para una propiedad o para un indizador que no tiene un modificador override, solo se permite un modificador de descriptor de acceso (accessor-modifier) si la propiedad o el indizador tiene un descriptor de acceso get y set, y solo se permite en uno de esos descriptores de acceso.
* Para una propiedad o un indizador que incluye un modificador override un descriptor de acceso debe coincidir con el modificador de descriptor de acceso (accessor-modifier), si lo hay, del descriptor de acceso que se está invalidando.
* El modificador de descriptor de acceso (accessor-modifier) debe declarar una accesibilidad que sea estrictamente más restrictiva que la accesibilidad declarada de la misma propiedad o indizador. Para ser preciso:
* Si la propiedad o el indizador tiene una accesibilidad declarada de public, el modificador de descriptor de acceso (accessor-modifier) puede ser protected internal, internal, protected o private.
* Si la propiedad o el indizador tiene una accesibilidad declarada de protected internal, el modificador de descriptor de acceso (accessor-modifier) puede ser internal, protected o private.
* Si la propiedad o el indizador tiene una accesibilidad declarada de internal o protected, el modificador de descriptor de acceso (accessor-modifier) debe ser private.
* Si la propiedad o el indizador tiene una accesibilidad declarada de private, no se puede utilizar el modificador de descriptor de acceso (accessor-modifier).

Para las propiedades abstract y extern, el cuerpo de descriptor de acceso (accessor-body) de cada descriptor de acceso especificado consiste simplemente en un punto y coma. Una propiedad no abstracta y no externa puede ser una propiedad automáticamente implementada, en cuyo caso se deben dar ambos descriptores de acceso, get y set, y ambos con punto y coma (§10.7.3). Para los descriptores de acceso de cualquier otra propiedad no abstracta y no externa, el cuerpo de descriptor de acceso (accessor-body) es un bloque (block) que especifica las instrucciones que deben ejecutarse cuando se invoque al correspondiente descriptor de acceso.

Un descriptor de acceso get corresponde a un método sin parámetros que devuelve un valor del tipo de la propiedad. Exceptuando cuando se trata del destino de una asignación, al hacer referencia a una propiedad en una expresión, se invoca al descriptor de acceso get de la propiedad para calcular su valor (§7.1.1). El cuerpo de un descriptor de acceso get debe cumplir las reglas de los métodos que devuelven valores, descritas en la sección §10.6.10. En concreto, todas las instrucciones return del cuerpo de un descriptor de acceso get deben especificar una expresión que sea convertible implícitamente al tipo de la propiedad. Además, el extremo de un descriptor de acceso get no debe ser alcanzable.

Un descriptor de acceso set corresponde a un método con un solo parámetro de valor del tipo de la propiedad y un tipo de valor devuelto void. El parámetro implícito de un descriptor de acceso set siempre se denomina value. Cuando se hace referencia a una propiedad como el destino de una asignación (§7.17) o como el operando de los operadores ++ o -- (§7.6.9, §7.7.5), se invoca el descriptor de acceso set con un argumento (cuyo valor es el del lado derecho de la asignación o del operando del operador ++ o --), que proporciona el nuevo valor (§7.17.1). El cuerpo de un descriptor de acceso set debe cumplir las reglas de los métodos void, descritas en la sección §10.6.10. En concreto, no está permitido que las instrucciones return del cuerpo de un descriptor de acceso set especifiquen una expresión. Debido a que un descriptor de acceso set tiene implícitamente un parámetro denominado value, supone un error en tiempo de compilación que una declaración de variable local o una declaración de constante en un descriptor de acceso set tengan ese nombre.

Basándose en la existencia o no de los descriptores de acceso get y set, las propiedades se clasifican en:

* Una propiedad que incluye los descriptores de acceso get y set se dice que es una propiedad de lectura-escritura.
* Una propiedad que solo tiene un descriptor de acceso get se dice que es una propiedad de solo lectura. Supone un error en tiempo de compilación que una propiedad de solo lectura sea el destino de una asignación.
* Una propiedad que solo tiene un descriptor de acceso set se dice que es una propiedad de solo escritura. Salvo cuando la propiedad sea el destino de una asignación, supone un error en tiempo de compilación hacer referencia en una expresión a una propiedad de sólo escritura.

En el ejemplo

public class Button: Control  
{  
 private string caption;

public string Caption {  
 get {  
 return caption;  
 }  
 set {  
 if (caption != value) {  
 caption = value;  
 Repaint();  
 }  
 }  
 }

public override void Paint(Graphics g, Rectangle r) {  
 // Painting code goes here  
 }  
}

el control Button declara una propiedad pública Caption. El descriptor de acceso get de la propiedad Caption devuelve la cadena almacenada en el campo privado caption. El descriptor de acceso set comprueba si el nuevo valor es diferente del valor actual y, en ese caso, almacena el nuevo valor y vuelve a dibujar el control. Las propiedades siguen a menudo el diseño descrito anteriormente: el descriptor de acceso get simplemente devuelve un valor almacenado en un campo privado, y el descriptor de acceso set modifica el campo privado y después realiza otras acciones adicionales necesarias para actualizar completamente el estado del objeto.

Dada la clase Button definida anteriormente, en el siguiente ejemplo se muestra el uso de la propiedad Caption:

Button okButton = new Button();  
okButton.Caption = "OK"; // Invokes set accessor  
string s = okButton.Caption; // Invokes get accessor

Aquí, el descriptor de acceso set se invoca mediante la asignación de un valor a la propiedad, y el descriptor de acceso get se invoca haciendo referencia a esta en una expresión.

Los descriptores de acceso get y set de una propiedad no son miembros distintos, y no es posible declarar los descriptores de acceso de una propiedad de forma independiente. Por ello, no es posible que los dos descriptores de acceso de una propiedad de lectura-escritura tengan distinta accesibilidad. En el ejemplo

class A  
{  
 private string name;

public string Name { // Error, duplicate member name  
 get { return name; }  
 }

public string Name { // Error, duplicate member name  
 set { name = value; }  
 }  
}

no se declara una única propiedad de sólo lectura. Se declaran dos propiedades con el mismo nombre, una de sólo lectura y otra de sólo escritura. Dado que dos miembros que se declaran en la misma clase no pueden tener el mismo nombre, el código del ejemplo generará un error de compilación.

Cuando una clase derivada declara una propiedad con el mismo nombre que una propiedad heredada, la propiedad derivada oculta la propiedad heredada tanto para lectura como para escritura. En el ejemplo

class A  
{  
 public int P {  
 set {...}  
 }  
}

class B: A  
{  
 new public int P {  
 get {...}  
 }  
}

la propiedad P de B oculta la propiedad P de A para lectura y escritura. Por lo tanto, en las instrucciones

B b = new B();  
b.P = 1; // Error, B.P is read-only  
((A)b).P = 1; // Ok, reference to A.P

la asignación a b.P causa un error en tiempo de compilación del que debe informarse, ya que la propiedad P de solo lectura en B oculta la propiedad P de solo escritura en A. Observe, sin embargo, que se puede utilizar una conversión para tener acceso a la propiedad P oculta.

A diferencia de los campos públicos, las propiedades proporcionan una separación entre el estado interno de un objeto y su interfaz pública. Considere este ejemplo:

class Label  
{  
 private int x, y;  
 private string caption;

public Label(int x, int y, string caption) {  
 this.x = x;  
 this.y = y;  
 this.caption = caption;  
 }

public int X {  
 get { return x; }  
 }

public int Y {  
 get { return y; }  
 }

public Point Location {  
 get { return new Point(x, y); }  
 }

public string Caption {  
 get { return caption; }  
 }  
}

Aquí, la clase Label utiliza dos campos int, x e y, para almacenar su ubicación. La ubicación se expone públicamente como una propiedad X e Y, y como una propiedad Location de tipo Point. Si en una futura versión de Label fuera más conveniente almacenar internamente la ubicación como un Point, se podría realizar el cambio sin que afectara a la interfaz pública de la clase:

class Label  
{  
 private Point location;  
 private string caption;

public Label(int x, int y, string caption) {  
 this.location = new Point(x, y);  
 this.caption = caption;  
 }

public int X {  
 get { return location.x; }  
 }

public int Y {  
 get { return location.y; }  
 }

public Point Location {  
 get { return location; }  
 }

public string Caption {  
 get { return caption; }  
 }  
}

Si x e y fueran campos public readonly, no sería posible realizar ese cambio en la clase Label.

La exposición del estado mediante propiedades no es menos eficiente que exponer los campos directamente. En concreto, cuando una propiedad no es virtual y contiene sólo una pequeña cantidad de código, el entorno de ejecución puede reemplazar las llamadas a los descriptores de acceso con el código real de los descriptores de acceso. Este proceso es conocido con el nombre de alineado, y permite que el acceso a las propiedades sea tan eficiente como el acceso a los campos, preservando la mayor flexibilidad de las propiedades.

Puesto que invocar un descriptor de acceso get equivale conceptualmente a leer el valor de un campo, se considera una técnica de programación poco recomendable el que los descriptores de acceso get muestren efectos secundarios. En el ejemplo

class Counter  
{  
 private int next;

public int Next {  
 get { return next++; }  
 }  
}

el valor de la propiedad Next depende del número de veces que se haya tenido acceso anteriormente a la propiedad. Por consiguiente, puesto que el acceso a la propiedad produce un efecto secundario observable, es preferible implementar la propiedad como método.

La convención “sin efectos secundarios” que se aplica a los descriptores de acceso get no significa que éstos deban escribirse siempre únicamente para que devuelvan los valores almacenados en campos. De hecho, los descriptores de acceso get a menudo calculan el valor de una propiedad mediante el acceso a varios campos o invocando métodos. No obstante, un descriptor de acceso get diseñado correctamente no lleva a cabo acciones que generen cambios observables en el estado del objeto.

Pueden utilizarse propiedades para retrasar la inicialización de un recurso hasta el momento en que se haga referencia al mismo. Por ejemplo:

using System.IO;

public class Console  
{  
 private static TextReader reader;  
 private static TextWriter writer;  
 private static TextWriter error;

public static TextReader In {  
 get {  
 if (reader == null) {  
 reader = new StreamReader(Console.OpenStandardInput());  
 }  
 return reader;  
 }  
 }

public static TextWriter Out {  
 get {  
 if (writer == null) {  
 writer = new StreamWriter(Console.OpenStandardOutput());  
 }  
 return writer;  
 }  
 }

public static TextWriter Error {  
 get {  
 if (error == null) {  
 error = new StreamWriter(Console.OpenStandardError());  
 }  
 return error;  
 }  
 }  
}

La clase Console contiene tres propiedades, In, Out y Error, que representan los dispositivos estándar de entrada, salida y error, respectivamente. Mediante la exposición de estos miembros como propiedades, la clase Console puede retrasar su inicialización hasta que se utilicen realmente. Por ejemplo, en la primera referencia a la propiedad Out del código siguiente

Console.Out.WriteLine("hello, world");

se crea el TextWriter subyacente para el dispositivo de salida. Pero si la aplicación no hace referencia a las propiedades In y Error, entonces no se crean objetos para esos dispositivos.

### Propiedades automáticamente implementadas

Cuando una propiedad se especifica como propiedad automáticamente implementada, pasa a estar disponible para la propiedad un campo de respaldo oculto automáticamente, y los descriptores de acceso se implementan para leer y escribir en dicho campo de respaldo.

El siguiente ejemplo:

public class Point {  
 public int X { get; set; } // automatically implemented  
 public int Y { get; set; } // automatically implemented  
}

equivale a la siguiente declaración:

public class Point {  
 private int x;  
 private int y;  
 public int X { get { return x; } set { x = value; } }  
 public int Y { get { return y; } set { y = value; } }  
}

Al estar inaccesible el campo de respaldo, sólo se puede leer y escribir mediante los descriptores de acceso de la propiedad, incluso dentro del tipo contenedor. Esto significa que las propiedades automáticamente implementadas de sólo escritura o de sólo lectura no tienen sentido, y no se permiten. No obstante, es posible establecer el nivel de acceso de cada descriptor de acceso de manera distinta. Por tanto, el efecto de una propiedad de sólo lectura con un campo de respaldo privado se puede imitar del siguiente modo:

public class ReadOnlyPoint {  
 public int X { get; private set; }  
 public int Y { get; private set; }  
 public ReadOnlyPoint(int x, int y) { X = x; Y = y; }  
}

Esta restricción también significa que sólo se puede lograr una asignación definitiva de tipos struct con propiedades autoimplementadas mediante un constructor estándar del struct, ya que la asignación a la propiedad en sí requiere que el struct esté asignada de manera definitiva. Esto significa que los constructores definidos por el usuario deben llamar al constructor predeterminado.

### Accesibilidad

Si un descriptor de acceso tiene un modificador de descriptor de acceso (accessor-modifier), el dominio de accesibilidad (§3.5.2) del descriptor de acceso viene determinado por la accesibilidad declarada del modificador de descriptor de acceso (accessor-modifier). Si un descriptor de acceso no tiene un modificador de descriptor de acceso (accessor-modifier), el dominio de accesibilidad del descriptor de acceso viene determinado por la accesibilidad declarada de la propiedad o del indizador.

La presencia de un modificador de descriptor de acceso (accessor-modifier) nunca afecta a la búsqueda de miembros (§7.3) ni a la resolución de sobrecargas (§7.5.3). Los modificadores de la propiedad o del indizador siempre determinan qué propiedad o indizador está enlazada independientemente del contexto del acceso.

Una vez seleccionada una propiedad o indizador concreto, los dominios de accesibilidad de los descriptores de acceso específicos implicados se utilizan para determinar si el uso es válido:

* Si el uso es un valor (§7.1.1), el descriptor de acceso get debe existir y ser accesible.
* Si el uso es como destino de una asignación simple (§7.17.1), el descriptor de acceso set debe existir y ser accesible.
* Si el uso es como el destino de la asignación compuesta (§7.17.2) o como el destino de los operadores ++ o -- (§7.5.9, §7.6.5), deben existir los descriptores de acceso get y set, y estar accesibles.

En el siguiente ejemplo, la propiedad A.Text está oculta por la propiedad B.Text, incluso en contextos donde sólo se llama al descriptor de acceso set. Por el contrario, la propiedad B.Count no es accesible para la clase M por lo que se utiliza la propiedad accesible A.Count en su lugar.

class A  
{  
 public string Text {  
 get { return "hello"; }  
 set { }  
 }

public int Count {  
 get { return 5; }  
 set { }  
 }  
}

class B: A  
{  
 private string text = "goodbye";   
 private int count = 0;

new public string Text {  
 get { return text; }  
 protected set { text = value; }  
 }

new protected int Count {   
 get { return count; }  
 set { count = value; }  
 }  
}

class M  
{  
 static void Main() {  
 B b = new B();  
 b.Count = 12; // Calls A.Count set accessor  
 int i = b.Count; // Calls A.Count get accessor  
 b.Text = "howdy"; // Error, B.Text set accessor not accessible  
 string s = b.Text; // Calls B.Text get accessor  
 }  
}

Es posible que un descriptor de acceso que se utiliza para implementar una interfaz no tenga un modificador de descriptor de acceso (accessor-modifier). Si sólo se utiliza un descriptor de acceso para implementar una interfaz, el otro descriptor de acceso se puede declarar con un modificador de descriptor de acceso (accessor-modifier):

public interface I  
{  
 string Prop { get; }  
}

public class C: I  
{  
 public Prop {  
 get { return "April"; } // Must not have a modifier here  
 internal set {...} // Ok, because I.Prop has no set accessor  
 }  
}

### Descriptores de acceso virtual, sellado, de invalidación y abstracto

Una declaración de propiedad virtual especifica que los descriptores de acceso de la propiedad son virtuales. El modificador virtual se aplica a los dos descriptores de acceso de una propiedad de lectura-escritura. No es posible que sólo uno de los descriptores de acceso de lectura-escritura sea virtual. Sin embargo, si uno de los descriptores de acceso es private, el modificador virtual se omite para dicho descriptor, ya que no tiene sentido que sea a la vez virtual y private.

Una declaración de propiedad abstract especifica que los descriptores de acceso de la propiedad son virtuales, pero no proporcionan una implementación real de los descriptores de acceso. En cambio, es necesario que las clases derivadas no abstractas proporcionen su propia implementación para los descriptores de acceso mediante la invalidación de la propiedad. Como un descriptor de acceso para una declaración de propiedad abstracta no proporciona una implementación real, su cuerpo de descriptor de acceso (accessor-body) consiste simplemente en un punto y coma.

Una declaración de propiedad que incluye los dos modificadores abstract y override especifica que la propiedad es abstracta y que invalida una propiedad base. Los descriptores de acceso de este tipo de propiedad son también abstractos.

Las declaraciones de propiedades abstractas sólo se permiten en clases abstractas (§10.1.1.1). Los descriptores de acceso de una propiedad virtual heredada se pueden invalidar en una clase derivada incluyendo una declaración de propiedad que especifique una directiva override. Esto se conoce como declaración de propiedad invalidada. Una declaración de propiedad invalidada no declara una nueva propiedad. En su lugar, simplemente especializa las implementaciones de los descriptores de acceso de una propiedad virtual existente.

Una declaración de propiedad invalidada debe especificar exactamente los mismos modificadores de accesibilidad, tipo y nombre que la propiedad heredada. Si la propiedad heredada sólo tiene un descriptor de acceso (por ejemplo, si la propiedad heredada es de sólo lectura o de sólo escritura), la propiedad invalidada debe incluir sólo ese descriptor de acceso. Si la propiedad heredada incluye ambos descriptores de acceso (por ejemplo, si la propiedad heredada es de lectura-escritura), la propiedad invalidada puede incluir un descriptor de acceso o ambos.

Una declaración de propiedad invalidada puede incluir el modificador sealed. El uso de este modificador impide que una clase derivada siga invalidando la propiedad. Los descriptores de acceso de una propiedad sellada también son sellados.

Salvo en las diferencias referentes a la sintaxis de declaración e invocación, los descriptores de acceso virtuales, sellados, de invalidación y abstractos se comportan exactamente igual que los métodos virtuales, sellados, de invalidación y abstractos. Específicamente, las reglas descritas en las secciones §10.6.3, §10.6.4, §10.6.5 y §10.6.6 se aplican como si los descriptores de acceso fueran métodos con una forma correspondiente:

* Un descriptor de acceso get corresponde a un método sin parámetros que devuelve un valor del tipo de la propiedad y los mismos modificadores que la propiedad contenedora.
* Un descriptor de acceso set corresponde a un método con un único parámetro de valor del tipo de la propiedad, un tipo de valor devuelto void y los mismos modificadores que la propiedad contenedora.

En el ejemplo

abstract class A  
{  
 int y;

public virtual int X {  
 get { return 0; }  
 }

public virtual int Y {  
 get { return y; }  
 set { y = value; }  
 }

public abstract int Z { get; set; }  
}

X es una propiedad virtual de solo lectura, Y es una propiedad virtual de lectura-escritura y Z es una propiedad abstracta de lectura-escritura. Puesto que Z es abstracta, la clase contenedora A debe declararse como abstracta.

A continuación se muestra una clase que se deriva de la clase A:

class B: A  
{  
 int z;

public override int X {  
 get { return base.X + 1; }  
 }

public override int Y {  
 set { base.Y = value < 0? 0: value; }  
 }

public override int Z {  
 get { return z; }  
 set { z = value; }  
 }  
}

Aquí, las declaraciones de X, Y y Z son declaraciones de propiedades invalidadas. Cada declaración de propiedad coincide exactamente con los modificadores de accesibilidad, tipo y nombre de la propiedad heredada correspondiente. El descriptor de acceso get de X y el descriptor de acceso set de Y usan la palabra clave base para tener acceso a los descriptores de acceso heredados. La declaración de Z invalida ambos descriptores de acceso abstractos; de este modo, no hay miembros de función abstracta pendientes en B, y esta puede ser una clase no abstracta.

Cuando una propiedad se declara como override, los descriptores de acceso deben estar accesibles al código de invalidación. Además, la accesibilidad declarada tanto de la propiedad como del indizador y los descriptores de acceso deben coincidir con los de los miembros y descriptores de acceso invalidados. Por ejemplo:

public class B  
{  
 public virtual int P {  
 protected set {...}  
 get {...}  
 }  
}

public class D: B  
{  
 public override int P {  
 protected set {...} // Must specify protected here  
 get {...} // Must not have a modifier here  
 }  
}

## Eventos

Un evento es un miembro que permite a un objeto o una clase proporcionar notificaciones. Los clientes pueden adjuntar código ejecutable a los eventos mediante controladores de eventos.

Los eventos se declaran mediante declaraciones de eventos (event-declaration):

event-declaration:  
attributesopt event-modifiersopt event type variable-declarators ;  
attributesopt event-modifiersopt event type member-name { event-accessor-declarations }

event-modifiers:  
event-modifier  
event-modifiers event-modifier

event-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

event-accessor-declarations:  
add-accessor-declaration remove-accessor-declaration  
remove-accessor-declaration add-accessor-declaration

add-accessor-declaration:  
attributesopt add block

remove-accessor-declaration:  
attributesopt remove block

Una declaración de evento (event-declaration) puede incluir un conjunto de atributos (attributes) (§17) y una combinación válida de los cuatro modificadores de acceso (§10.3.5), los modificadores new (§10.3.4), static (§10.6.2), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6) y extern (§10.6.7).

Las declaraciones de evento están sujetas a las mismas reglas que las declaraciones de método (§10.6) en lo que respecta a las combinaciones válidas de modificadores.

El tipo (type) de una declaración de evento debe ser un tipo delegado (delegate-type) (§4.2) y dicho tipo delegado debe ser al menos tan accesible como el propio evento (§3.5.4).

Una declaración de evento puede incluir una declaración de descriptor de acceso a eventos (event-accessor-declaration). Sin embargo, si no fuera así, el compilador las proporcionará automáticamente para los eventos no externos y no abstractos (§10.8.1); para los eventos externos, los descriptores de acceso se proporcionan externamente.

Una declaración de evento que omite las declaraciones de descriptor de acceso a eventos (event-accessor-declaration) define uno o más evento, uno para cada uno de los declaradores de variable (variable-declarator). Los atributos y modificadores se aplican a todos los miembros declarador por esa declaración de evento (event-declaration).

Es un error en tiempo de compilación que una declaración de evento (event-declaration) incluya el modificador abstract y declaraciones de acceso a eventos (event-accessor-declaration) entre llaves.

Cuando la declaración de un evento incluye un modificador extern, se dice que el evento es un evento externo. Debido a que una declaración de evento externo no proporciona ninguna implementación real, supone un error incluir a la vez el modificador extern y las declaraciones de descriptor de acceso a eventos (event-accessor-declaration).

Es un error en tiempo de compilación que un declarador de variable (variable-declarator) de una declaración de evento con un modificador abstract o external incluya un inicializador de variable (variable-initializer).

Un evento se puede utilizar como el operando izquierdo de los operadores += y -= (§7.17.3). Estos operadores se utilizan para adjuntar o quitar respectivamente controladores de eventos de un evento, y los modificadores de acceso del evento controlan los contextos en los que se permiten estas operaciones.

Como += y -= son las únicas operaciones permitidas en un evento fuera del tipo que declara el evento, mediante código externo se pueden agregar o quitar controladores de un evento, pero no se puede obtener ni modificar la lista subyacente de controladores de eventos de ninguna otra forma.

En una operación con el formato x += y o x -= y, cuando x es un evento y la referencia tiene lugar fuera del tipo que contiene la declaración de x, el resultado de la operación es de tipo void (en oposición al tipo de x, con el valor de x después de la asignación). Esta regla prohíbe que el código externo examine indirectamente el delegado subyacente de un evento.

En el siguiente ejemplo se muestra cómo adjuntar controladores de eventos a instancias de la clase Button:

public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control  
{  
 public event EventHandler Click;  
}

public class LoginDialog: Form  
{  
 Button OkButton;  
 Button CancelButton;

public LoginDialog() {  
 OkButton = new Button(...);  
 OkButton.Click += new EventHandler(OkButtonClick);  
 CancelButton = new Button(...);  
 CancelButton.Click += new EventHandler(CancelButtonClick);  
 }

void OkButtonClick(object sender, EventArgs e) {  
 // Handle OkButton.Click event  
 }

void CancelButtonClick(object sender, EventArgs e) {  
 // Handle CancelButton.Click event  
 }  
}

Aquí, el constructor de instancia de LoginDialog crea dos instancias de Button y adjunta controladores de evento a los eventos de Click.

### Eventos como campos

Dentro del texto del programa de la clase o struct que contiene la declaración de un evento, algunos eventos se pueden utilizar como campos. Para poder usarlo de esta forma, un evento no debe ser abstract ni extern, y no debe incluir explícitamente ninguna declaración de descriptor de acceso a eventos (event-accessor-declaration). Esta clase de evento se puede usar en cualquier contexto que permita un campo. El campo contiene un delegado (§15) que hace referencia a la lista de controladores de eventos que se han agregado al evento. Si no se ha agregado ningún controlador de evento, el campo contiene null.

En el ejemplo

public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control  
{  
 public event EventHandler Click;

protected void OnClick(EventArgs e) {  
 if (Click != null) Click(this, e);  
 }

public void Reset() {  
 Click = null;  
 }  
}

Click se usa como un campo dentro de la clase Button. Como se demuestra en el ejemplo, el campo se puede examinar, modificar y utilizar en expresiones de invocación a delegados. El método OnClick de la clase Button genera el evento Click. La noción de iniciar un evento equivale exactamente a invocar el delegado representado por el evento; por lo tanto, no hay construcciones especiales del lenguaje para producir eventos. Observe que la invocación del delegado está precedida por una comprobación que confirma que el delegado no es nulo.

Fuera de la declaración de la clase Button, el miembro Click solo se puede utilizar como operando izquierdo de los operadores += y –=, como en

b.Click += new EventHandler(…);

que anexa un delegado a la lista de invocación del evento Click, y

b.Click –= new EventHandler(…);

que quita un delegado de la lista de invocación del evento Click.

Al compilar un campo como un evento, el compilador crea automáticamente espacio de almacenamiento para alojar el delegado, y crea descriptores de acceso para el evento que agregan o quitan controladores de evento del campo del delegado. Las operaciones de agregado o eliminación son seguras para la ejecución de subprocesos y pueden hacerse (pero no es obligatorio hacerlas) mientras se mantiene el bloqueo (§8.12) en el objeto contenedor para un evento de instancia, o bien el objeto de tipo (§7.6.10.6) para un evento estático.

Por lo tanto, una declaración de evento de instancia de la forma:

class X  
{  
 public event D Ev;  
}

se compilará de manera equivalente a:

class X  
{  
 private D \_\_Ev; // field to hold the delegate

public event D Ev {  
 add {  
 /\* add the delegate in a thread safe way \*/  
 }

remove {  
 /\* remove the delegate in a thread safe way \*/  
 }  
 }  
}

En la clase X, las referencias a Ev on en el lado izquierdo de los operadores += y –= hacen que se llame a los descriptores de acceso de agregar y quitar. Todas las demás referencias a Ev, se compilan para hacer referencia al campo oculto \_\_Ev en su lugar (§7.6.4). El nombre “\_\_Ev” es arbitrario; el campo oculto puede tener cualquier nombre o incluso no tener ninguno.

### Descriptores de acceso de evento

Las declaraciones de eventos normalmente omiten las declaraciones de descriptores de acceso a eventos (event-accessor-declarations), como en el ejemplo anterior de Button. Una situación en la que puede darse es en el caso de que el costo de almacenamiento de un campo por evento no sea aceptable. En estos casos, una clase puede incluir declaraciones de descriptores de acceso a eventos (event-accessor-declarations) y utilizar un mecanismo privado para almacenar la lista de controladores de eventos.

Las declaraciones de descriptores de acceso a eventos (event-accessor-declarations) de un evento especifican las instrucciones ejecutables asociadas a la acción de agregar o quitar controladores de eventos.

Las declaraciones del descriptor de acceso consisten en una declaración de descriptor de acceso add (add-accessor-declaration) y en una declaración de descriptor de acceso remove (remove-accessor-declaration). Cada declaración de descriptor de acceso consiste en el token add o remove seguido de un bloque (block). El bloque (block) asociado a una declaración de descriptor de acceso add (add-accessor-declaration) especifica las instrucciones que se van a ejecutar cuando se agrega un controlador de evento, y el bloque (block) asociado a una declaración de descriptor de acceso remove (remove-accessor-declaration) especifica las instrucciones a ejecutar cuando se quita un controlador de evento.

Cada declaración de descriptor de acceso add (add-accessor-declaration) y cada declaración de descriptor de acceso remove (remove-accessor-declaration) corresponde a un método con un solo parámetro de valor del tipo de evento y un tipo devuelto void. El parámetro implícito de un descriptor de acceso a evento se denomina value. Cuando se utiliza un evento en una asignación de evento, se utiliza el descriptor de acceso de evento apropiado. Si el operador de asignación es específicamente +=, se utiliza el descriptor de acceso add y, si es -=, el descriptor de acceso remove. En ambos casos, el operando derecho del operador de asignación se utiliza como argumento para el descriptor de acceso del evento. El bloque de una declaración de descriptor de acceso add (add-accessor-declaration) o de una declaración de descriptor de acceso remove (remove-accessor-declaration) debe cumplir las reglas para los métodos void descritas en §10.6.10. En concreto, no está permitido que las instrucciones return del bloque especifiquen una expresión.

Debido a que un descriptor de acceso a evento tiene implícitamente un parámetro denominado value, se producirá un error en tiempo de compilación si una variable local o una constante declaradas en un descriptor de acceso a evento tienen ese nombre.

En el ejemplo

class Control: Component  
{  
 // Unique keys for events  
 static readonly object mouseDownEventKey = new object();  
 static readonly object mouseUpEventKey = new object();

// Return event handler associated with key  
 protected Delegate GetEventHandler(object key) {...}

// Add event handler associated with key  
 protected void AddEventHandler(object key, Delegate handler) {...}

// Remove event handler associated with key  
 protected void RemoveEventHandler(object key, Delegate handler) {...}

// MouseDown event  
 public event MouseEventHandler MouseDown {  
 add { AddEventHandler(mouseDownEventKey, value); }  
 remove { RemoveEventHandler(mouseDownEventKey, value); }  
 }

// MouseUp event  
 public event MouseEventHandler MouseUp {  
 add { AddEventHandler(mouseUpEventKey, value); }  
 remove { RemoveEventHandler(mouseUpEventKey, value); }  
 }

// Invoke the MouseUp event  
 protected void OnMouseUp(MouseEventArgs args) {  
 MouseEventHandler handler;   
 handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);  
 if (handler != null)  
 handler(this, args);  
 }  
}

la clase Control implementa un mecanismo de almacenamiento interno para los eventos. El método AddEventHandler asocia un valor de delegado a una clave, el método GetEventHandler devuelve el delegado asociado actualmente a una clave, y el método RemoveEventHandler quita un delegado como controlador de evento del evento especificado. Presumiblemente, el mecanismo de almacenamiento subyacente está diseñado para que no suponga un costo el hecho de asociar un valor de delegado null a una clave y, por lo tanto, los eventos no controlados no consumen almacenamiento.

### Eventos estáticos y de instancia

Cuando la declaración de un evento incluye un modificador static, se dice que el evento es un evento estático. Cuando no existe un modificador static, se dice que el evento es un evento de instancia.

Un evento estático no está asociado a una instancia específica y supone un error en tiempo de compilación hacer referencia a this en los descriptores de acceso de un evento estático.

Un evento de instancia está asociado a una determinada instancia de una clase y es posible tener acceso a la instancia con this (§7.6.7) en los descriptores de acceso del evento.

Cuando se hace referencia a un evento en un acceso a miembro (member-access) (§7.6.4) con la estructura E.M, si M es un evento estático, E debe denotar un tipo que contenga M, y si M es un evento de instancia, E debe denotar una instancia de un tipo que contenga M.

Las diferencias entre miembros estáticos y de instancia se tratan con más detalle en §10.3.7.

### Descriptores de acceso virtual, sellado, de invalidación y abstracto

Una declaración de evento virtual especifica que los descriptores de acceso del evento son virtuales. El modificador virtual se aplica a todos los descriptores de acceso de un evento.

Una declaración de evento abstract especifica que los descriptores de acceso del evento son virtuales, pero no proporciona una implementación real de los descriptores de acceso. En su lugar, es necesario que las clases derivadas no abstractas proporcionen su propia implementación para los descriptores de acceso mediante la invalidación del evento. Debido a que un descriptor de acceso para una declaración de evento abstracto no proporciona una implementación real, no puede proporcionar declaraciones de descriptores de acceso a eventos (event-accessor-declaration) entre llaves.

Una declaración de evento que incluye los modificadores abstract y override especifica que el evento es abstracto y que invalida un evento base. Los descriptores de acceso de este tipo de eventos son también abstractos.

Las declaraciones de eventos abstractos sólo se permiten en clases abstractas (§10.1.1.1).

Los descriptores de acceso de un evento virtual heredado pueden ser invalidados en una clase derivada incluyendo una declaración de evento que especifique un modificador override. Esto se conoce con el nombre de declaración de evento invalidado. Una declaración de evento invalidado no declara un evento nuevo. En lugar de eso, simplemente especializa las implementaciones de los descriptores de acceso de un evento virtual existente.

Una declaración de evento invalidado debe especificar exactamente los mismos modificadores de accesibilidad, tipo y nombre que el evento invalidado.

Una declaración de evento invalidado puede incluir el modificador sealed. El uso de este modificador impide que una clase derivada siga invalidando el evento. Los descriptores de acceso de un evento sellado también son sellados.

Supone un error en tiempo de compilación que una declaración de evento invalidado incluya el modificador new.

Salvo en las diferencias referentes a la sintaxis de declaración e invocación, los descriptores de acceso virtuales, sellados, de invalidación y abstractos se comportan exactamente igual que los métodos virtuales, sellados, de invalidación y abstractos. Específicamente, las reglas descritas en las secciones §10.6.3, §10.6.4, §10.6.5 y §10.6.6 se aplican como si los descriptores de acceso fueran métodos con una forma correspondiente. Cada descriptor de acceso corresponde a un método con un solo parámetro de valor del tipo del evento, un tipo de valor devuelto void y los mismos modificadores que el evento contenedor.

## Indizadores

Un indizador es un miembro que permite indizar un objeto de la misma manera que una matriz. Los indizadores se declaran mediante declaraciones de indizadores (indexer-declaration):

indexer-declaration:  
attributesopt indexer-modifiersopt indexer-declarator { accessor-declarations }

indexer-modifiers:  
indexer-modifier  
indexer-modifiers indexer-modifier

indexer-modifier:  
new  
public  
protected  
internal  
private   
virtual  
sealed  
override  
abstract  
extern

indexer-declarator:  
type this [ formal-parameter-list ]  
type interface-type . this [ formal-parameter-list ]

Una declaración de indizador (indexer-declaration) puede incluir un conjunto de atributos (attributes) (§17) y una combinación válida de los cuatro modificadores de acceso (§10.3.5), los modificadores new (§10.3.4), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6) y extern (§10.6.7).

Las declaraciones de indizadores están sujetas a las mismas reglas que las declaraciones de métodos (§10.6) en lo que respecta a las combinaciones válidas de modificadores, con la única excepción de que el modificador static no está permitido en una declaración de indizador.

Los modificadores virtual, override y abstract se excluyen mutuamente excepto en un caso. Los modificadores abstract y override pueden utilizarse conjuntamente de manera que un indizador abstracto pueda invalidar a uno virtual.

El tipo (type) de una declaración de indizador especifica el tipo de elemento del indizador que se incluye en la declaración. A menos que el indizador no sea una implementación explícita de miembro de interfaz, el tipo (type) va seguido de la palabra clave this. Para una implementación explícita de miembro de interfaz, el tipo (type) viene seguido de un tipo de interfaz (interface-type), de un “.” y de la palabra clave this. A diferencia de otros miembros, los indizadores no tienen nombres definidos por el usuario.

La lista de parámetros formales (formal-parameter-list) especifica los parámetros del indizador. La lista de parámetros formales de un indizador se corresponde con la de un método (§10.6.1), exceptuando que al menos se debe especificar un parámetro y que no se permiten los modificadores de parámetro ref ni out.

El tipo (type) de un indizador y cada uno de los tipos a los que se hace referencia en la lista de parámetros formales (formal-parameter-list) deben tener como mínimo el mismo nivel de accesibilidad que el propio indizador (§3.5.4).

Las declaraciones de descriptores de acceso (accessor-declarations) (§10.7.2), que deben estar encerradas entre los tokens “{” y “}”, declaran los descriptores de acceso del indizador. Los descriptores de acceso especifican las instrucciones ejecutables asociadas a los elementos de lectura y escritura del indizador.

Aunque la sintaxis para tener acceso a un elemento de indizador es la misma que para un elemento de matriz, un elemento de indizador no está clasificado como variable. Por tanto, no se puede pasar un elemento de indizador como argumento ref u out.

La lista de parámetros formales de un indizador define la signatura (§3.6) del indizador. En concreto, la firma de un indizador está formada por el número de parámetros formales y sus tipos. El tipo y el nombre de los elementos de los parámetros formales no forman parte de la firma del indizador.

La firma de un indizador debe ser diferente de las firmas de los demás indizadores declarados en la misma clase.

Los indizadores y las propiedades son similares conceptualmente, pero tienen las siguientes diferencias:

* Una propiedad se identifica por su nombre, mientras que un indizador se identifica por su firma.
* Para tener acceso a una propiedad se utiliza un nombre simple (simple-name) (§7.6.2) o un acceso de miembro (member-access) (§7.6.4), pero a un elemento del indizador se tiene acceso a través de un acceso de elemento (element-access) (§7.6.6.2).
* Una propiedad puede ser un miembro static y un indizador siempre es un miembro de instancia.
* Un descriptor de acceso get de una propiedad corresponde a un método sin parámetros, pero un descriptor de acceso get de un indizador corresponde a un método con la misma lista de parámetros formales que el indizador.
* Un descriptor de acceso set de una propiedad corresponde a un método con un solo parámetro denominado value, pero un descriptor de acceso set de un indizador corresponde a un método con la misma lista de parámetros formales que el indizador y un parámetro adicional denominado value.
* Supone un error en tiempo de compilación que un descriptor de acceso de un indizador declare una variable local con el mismo nombre que un parámetro del indizador.
* En una declaración de propiedad invalidada, se tiene acceso a la propiedad heredada utilizando la sintaxis base.P, donde P es el nombre de la propiedad. En una declaración de indizador invalidado, se tiene acceso al indizador heredado utilizando la sintaxis base[E], donde E es una lista de expresiones separadas por comas.

Al margen de estas diferencias, todas las reglas definidas en §10.7.2 y §10.7.3 se aplican tanto a los descriptores de acceso de indizadores como a los descriptores de acceso de propiedades.

Cuando la declaración de un indizador incluye un modificador extern, se dice que es un indizador externo. Debido a que la declaración de un indizador externo no proporciona una implementación real, cada una de sus declaraciones de descriptores de acceso (accessor-declarations) consiste en un punto y coma.

En el ejemplo siguiente se declara una clase BitArray que implementa un indizador para obtener acceso a los bits individuales de una matriz de bits.

using System;

class BitArray  
{  
 int[] bits;  
 int length;

public BitArray(int length) {  
 if (length < 0) throw new ArgumentException();  
 bits = new int[((length - 1) >> 5) + 1];  
 this.length = length;  
 }

public int Length {  
 get { return length; }  
 }

public bool this[int index] {  
 get {  
 if (index < 0 || index >= length) {  
 throw new IndexOutOfRangeException();  
 }  
 return (bits[index >> 5] & 1 << index) != 0;  
 }  
 set {  
 if (index < 0 || index >= length) {  
 throw new IndexOutOfRangeException();  
 }  
 if (value) {  
 bits[index >> 5] |= 1 << index;  
 }  
 else {  
 bits[index >> 5] &= ~(1 << index);  
 }  
 }  
 }  
}

Una instancia de la clase BitArray consume mucha menos memoria que el correspondiente bool[] (ya que cada valor de la instancia del primero ocupa sólo un bit en lugar del byte que consume el último), pero permite las mismas operaciones que bool[].

La siguiente clase CountPrimes utiliza una BitArray y el clásico algoritmo de “criba” para calcular el número de primos entre 1 y un número máximo dado:

class CountPrimes  
{  
 static int Count(int max) {  
 BitArray flags = new BitArray(max + 1);  
 int count = 1;  
 for (int i = 2; i <= max; i++) {  
 if (!flags[i]) {  
 for (int j = i \* 2; j <= max; j += i) flags[j] = true;  
 count++;  
 }  
 }  
 return count;  
 }

static void Main(string[] args) {  
 int max = int.Parse(args[0]);  
 int count = Count(max);  
 Console.WriteLine("Found {0} primes between 1 and {1}", count, max);  
 }  
}

Observe que la sintaxis para tener acceso a los elementos de BitArray es precisamente la misma que para bool[].

En el siguiente ejemplo se muestra una clase de una cuadrícula de 26 × 10 que tiene un indizador con dos parámetros. El primer parámetro debe ser una letra mayúscula o minúscula dentro del intervalo A–Z, y el segundo debe ser un número entero dentro del intervalo 0–9.

using System;

class Grid  
{  
 const int NumRows = 26;  
 const int NumCols = 10;

int[,] cells = new int[NumRows, NumCols];

public int this[char c, int col] {  
 get {  
 c = Char.ToUpper(c);  
 if (c < 'A' || c > 'Z') {  
 throw new ArgumentException();  
 }  
 if (col < 0 || col >= NumCols) {  
 throw new IndexOutOfRangeException();  
 }  
 return cells[c - 'A', col];  
 }

set {  
 c = Char.ToUpper(c);  
 if (c < 'A' || c > 'Z') {  
 throw new ArgumentException();  
 }  
 if (col < 0 || col >= NumCols) {  
 throw new IndexOutOfRangeException();  
 }  
 cells[c - 'A', col] = value;  
 }  
 }  
}

### Sobrecarga de indizadores

Las reglas de resolución de sobrecarga de indizadores se describen en §7.5.2.

## Operadores

Un operador es un miembro que define el significado de un operador de expresión que puede aplicarse a instancias de la clase. Los operadores se declaran mediante declaraciones de operadores (operator-declaration):

operator-declaration:  
attributesopt operator-modifiers operator-declarator operator-body

operator-modifiers:  
operator-modifier  
operator-modifiers operator-modifier

operator-modifier:  
public  
static  
extern

operator-declarator:  
unary-operator-declarator  
binary-operator-declarator  
conversion-operator-declarator

unary-operator-declarator:  
type operator overloadable-unary-operator ( type identifier )

overloadable-unary-operator: one of  
+ - ! ~ ++ -- true false

binary-operator-declarator:  
type operator overloadable-binary-operator ( type identifier , type identifier )

overloadable-binary-operator:  
+  
-  
\*  
/  
%  
&  
|  
^  
<<  
right-shift  
==  
!=  
>  
<  
>=  
<=

conversion-operator-declarator:  
implicit operator type ( type identifier )  
explicit operator type ( type identifier )

operator-body:  
block  
;

Hay tres categorías de operadores sobrecargables: operadores unarios (§10.10.1), operadores binarios (§10.10.2) y operadores de conversión (§10.10.3).

Cuando la declaración de un operador incluye un modificador extern, se dice que el operador es un operador externo. Debido a que un operador externo no proporciona una implementación real, su cuerpo de operador (operator-body) consiste en un punto y coma. Para el resto de operadores, el cuerpo de operador (operator-body) consiste en un bloque (block) que especifica las instrucciones que deben ejecutarse cuando se invoca al operador. El bloque (block) de un operador debe cumplir las reglas de los métodos que devuelven valores, descritas en §10.6.10.

Las siguientes reglas se aplican a todas las declaraciones de operadores:

* Una declaración de operador debe incluir los modificadores public y static.
* Los parámetros de un operador deben ser parámetros de valor (§5.1.4). Supone un error en tiempo de compilación que una declaración de operador especifique los parámetros ref u out.
* La signatura de un operador (§10.10.1, §10.10.2, §10.10.3) debe ser diferente de las signaturas de todos los demás operadores declarados en la misma clase.
* Todos los tipos a los que se hace referencia en una declaración de operador deben ser por lo menos tan accesibles como el propio operador (§3.5.4).
* Cuando el mismo modificador aparece varias veces en una declaración de operador, se produce un error.

Cada categoría de operador impone restricciones adicionales, que se describen en las secciones siguientes.

Al igual que otros miembros, los operadores declarados en una clase base se heredan por sus clases derivadas. Debido a que las declaraciones de operadores siempre necesitan que la clase o el struct donde se declara el operador participe en la firma del operador, no es posible que un operador declarado en una clase derivada oculte un operador declarado en una clase base. Por lo tanto, nunca es necesario el modificador new, y por ello no se permite en una declaración de operador.

Puede encontrar información adicional sobre operadores unarios y binarios en §7.3.

Puede encontrar información adicional sobre operadores de conversión en §6.4.

### Operadores unarios

Las siguientes reglas se aplican a las declaraciones de operadores unarios, donde T denota el tipo de instancia de la clase o struct que contiene la declaración del operador:

* Un operador unario +, -, ! o ~ debe adoptar un único parámetro del tipo T o T? y puede devolver cualquier tipo.
* Un operador unario ++ o -- debe obtener un solo parámetro de tipo T o T? y debe devolver el mismo tipo o un tipo derivado de él.
* Un operador unario true o false debe adoptar un único parámetro del tipo T o T?, y debe devolver cualquier tipo bool.

La signatura de un operador unario consiste en el token de operador (+, -, !, ~, ++, --, true o false) y el tipo del parámetro formal. El tipo de valor devuelto no forma parte de la firma de un operador unario, ni es el nombre del parámetro formal.

Los operadores unarios true y false requieren declaraciones par a par. Se produce un error en tiempo de compilación si una clase declara uno de estos operadores sin declarar el otro. Los operadores true y false se describen más detalladamente en las secciones §7.12.2 y §7.20.

En el siguiente ejemplo se muestra una implementación y la utilización subsiguiente del operador operator ++ para una clase de vector de tipo entero:

public class IntVector  
{  
 public IntVector(int length) {...}

public int Length {...} // read-only property

public int this[int index] {...} // read-write indexer

public static IntVector operator ++(IntVector iv) {  
 IntVector temp = new IntVector(iv.Length);  
 for (int i = 0; i < iv.Length; i++)  
 temp[i] = iv[i] + 1;  
 return temp;  
 }  
}

class Test  
{  
 static void Main() {  
 IntVector iv1 = new IntVector(4); // vector of 4 x 0  
 IntVector iv2;

iv2 = iv1++; // iv2 contains 4 x 0, iv1 contains 4 x 1  
 iv2 = ++iv1; // iv2 contains 4 x 2, iv1 contains 4 x 2  
 }  
}

Observe cómo el método del operador devuelve el valor producido al sumar 1 al operando, al igual que los operadores de incremento y decremento postfijo (§7.6.9), y los operadores de incremento y decremento prefijo (§7.7.5). A diferencia de C++, este método no necesita modificar el valor de su operando directamente. De hecho, la modificación del valor del operando infringiría la semántica estándar del operador postfijo de incremento.

### Operadores binarios

Las siguientes reglas se aplican a las declaraciones de operadores binarios, donde T denota el tipo de instancia de la clase o struct que contiene la declaración del operador:

* Un operador binario que no sea de desplazamiento debe adoptar dos parámetros (al menos uno debe tener el tipo T o T?) y puede devolver cualquier tipo.
* Un operador binario << o >> debe adoptar dos parámetros (el primero debe tener el tipo T o T? y el segundo debe tener el tipo int o int?) y puede devolver cualquier tipo.

La signatura de un operador binario consiste en el token (+, -, \*, /, %, &, |, ^, <<, >>, ==, !=, >, <, >= o <=) y los tipos de los dos parámetros formales. El tipo de valor devuelto y los nombres de los parámetros formales no forman parte de la firma de un operador binario.

Algunos operadores binarios requieren declaración de par a par. Para la declaración de cada operador del par, debe existir una declaración coincidente del otro operador del par. Las declaraciones de dos operadores R1 op*1*(P1, Q1) y R2 op*2*(P2, Q2) coinciden si existen conversiones de identidad entre los tipos devueltos R1 y R2, entre los tipos de operandos P1 y P2 y entre los tipos de operandos Q1 y Q2. Los siguientes operadores requieren declaración par a par:

* operator == y operator !=
* operator > y operator <
* operator >= y operator <=

### Operadores de conversión

Una declaración de operador de conversión introduce una conversión definida por el usuario (§6.4) que aumenta las conversiones implícitas y explícitas predefinidas.

Una declaración de operador de conversión que incluye la palabra clave implicit define una conversión implícita definida por el usuario. Las conversiones implícitas pueden ocurrir en distintas situaciones, incluyendo las invocaciones de miembros de función, las expresiones de conversión y las asignaciones. Esta categoría se explica con más detalle en la sección §6.1.

Una declaración de operador de conversión que incluye la palabra clave explicit define una conversión explícita definida por el usuario. Las conversiones explícitas pueden ocurrir en las expresiones de conversión, y se describen detalladamente en §6.2.

Un operador de conversión convierte de un tipo de origen, indicado por el tipo del parámetro del operador de conversión, en un tipo destino, indicado por el tipo de valor devuelto del operador de conversión.

Para un tipo de origen S y de destino T dados, si S o T son tipos que aceptan valores NULL, S0 y T0 hacen referencia a sus tipos subyacentes, en caso contrario S0 y T0 son iguales que S y T respectivamente. Una clase o struct tiene permitido declarar una conversión de un tipo de origen S a un tipo de destino T solamente si son verdaderos todos los puntos siguientes:

* S0 y T0 son tipos diferentes.
* S0 o T0 es el tipo de clase o estructura en el que tiene lugar la declaración del operador.
* Ni S0 ni T0 es un tipo de interfaz (interface-type).
* Excluyendo las conversiones definidas por el usuario, no existe una conversión de S a T ni de T a S.

En lo que respecta a estas reglas, cualquier parámetro de tipo asociado con S o T se considera un tipo único sin relación de herencia con otros tipos; asimismo, se omite cualquier restricción en dichos parámetros de tipo.

En el ejemplo

class C<T> {...}

class D<T>: C<T>  
{  
 public static implicit operator C<int>(D<T> value) {...} // Ok

public static implicit operator C<string>(D<T> value) {...} // Ok

public static implicit operator C<T>(D<T> value) {...} // Error  
}

las dos primeras declaraciones de operador se permiten porque, para los objetivos especificados en la sección §10.9.3, T, int y string se consideran respectivamente tipos únicos sin relación. Sin embargo, el tercer operador es un error porque C<T> es la clase base de D<T>.

De la segunda regla se deriva que un operador de conversión debe convertir a o del tipo de la clase o struct en la que se declara el operador. Por ejemplo, es posible que un tipo de clase o de struct C defina una conversión de C a int y de int a C, pero no de int a bool.

No es posible redefinir directamente una conversión predefinida. Por lo tanto, no está permitido utilizar operadores de conversión para convertir de o a object porque ya existen conversiones implícitas y explícitas entre object y el resto de tipos. Además, ni el tipo de origen ni el de destino de una conversión puede ser un tipo base del otro, porque entonces ya existiría una conversión.

Sin embargo, sí es posible declarar operadores en tipos genéricos que, para determinados argumentos de tipo, especifiquen conversiones que ya existan como conversiones predefinidas. En el ejemplo

struct Convertible<T>  
{  
 public static implicit operator Convertible<T>(T value) {...}

public static explicit operator T(Convertible<T> value) {...}  
}

cuando el tipo object se especifica como un argumento de tipo para T, el segundo operador declara una conversión que ya existe (existe una conversión implícita y, por lo tanto, también explícita desde cualquier tipo al tipo object).

En los casos donde se da una conversión predefinida entre dos tipos, se omite toda conversión definida por el usuario entre dichos tipos. Concretamente:

* Si ya existe una conversión implícita predefinida (§6.1) del tipo S al tipo T, se omitirán todas las conversiones definidas por el usuario (implícitas o explícitas) de S a T.
* Si ya existe una conversión explícita predefinida (§6.2) del tipo S al tipo T, se omitirán todas las conversiones explícitas definidas por el usuario de S a T. Además:
* Si T es un tipo de interfaz, se ignoran las conversiones implícitas definidas por el usuario de S en T.
* De lo contrario, se seguirán considerando las conversiones implícitas definidas por el usuario de S en T.

Para todos los tipos, excepto para object, los operadores declarados por el tipo Convertible<T> anteriormente no entran en conflicto con las conversiones predefinidas. Por ejemplo:

void F(int i, Convertible<int> n) {  
 i = n; // Error  
 i = (int)n; // User-defined explicit conversion  
 n = i; // User-defined implicit conversion  
 n = (Convertible<int>)i; // User-defined implicit conversion  
}

Sin embargo, para el tipo object, las conversiones predefinidas ocultan conversiones definidas por el usuario en todos los casos excepto en uno:

void F(object o, Convertible<object> n) {  
 o = n; // Pre-defined boxing conversion  
 o = (object)n; // Pre-defined boxing conversion  
 n = o; // User-defined implicit conversion  
 n = (Convertible<object>)o; // Pre-defined unboxing conversion  
}

La conversiones definidas por el usuario no pueden convertirse a o de los tipos de interfaz (interface-type). Esta restricción impide, en particular, que se produzcan transformaciones definidas por el usuario cuando se convierte a un tipo de interfaz (interface-type), y asegura que una conversión a un tipo de interfaz se ejecute correctamente sólo si el objeto que se está convirtiendo implementa realmente el tipo de interfaz.

La firma de un operador de conversión está formado por el tipo de origen y el tipo de destino (ésta es la única forma de miembro en la que el tipo de valor devuelto participa en la firma). La clasificación implicit o explicit de un operador de conversión no forma parte de la signatura del operador. Por lo tanto, una clase o un struct no puede declarar a la vez operadores de conversión implicit y explicit con los mismos tipos de origen y destino.

En general, las conversiones implícitas definidas por el usuario deben diseñarse para que nunca produzcan excepciones ni pierdan información. Si una conversión definida por el usuario puede dar lugar a excepciones (por ejemplo, debido a que el argumento de origen está fuera del intervalo) o a pérdida de información (como descartar los bits de mayor orden), dicha conversión debería definirse como explícita.

En el ejemplo

using System;

public struct Digit  
{  
 byte value;

public Digit(byte value) {  
 if (value < 0 || value > 9) throw new ArgumentException();  
 this.value = value;  
 }

public static implicit operator byte(Digit d) {  
 return d.value;  
 }

public static explicit operator Digit(byte b) {  
 return new Digit(b);  
 }  
}

la conversión de Digit a byte es implícita porque nunca produce excepciones ni pierde información, pero la conversión de byte a Digit es explícita ya que Digit solo puede representar un subconjunto de los posibles valores de un byte.

## Constructores de instancia

Un constructor de instancia es un miembro que implementa las acciones que se requieren para inicializar una instancia de una clase. Los constructores de instancia se declaran mediante declaraciones de constructor (constructor-declaration):

constructor-declaration:  
attributesopt constructor-modifiersopt constructor-declarator constructor-body

constructor-modifiers:  
constructor-modifier  
constructor-modifiers constructor-modifier

constructor-modifier:  
public  
protected  
internal  
private  
extern

constructor-declarator:  
identifier ( formal-parameter-listopt ) constructor-initializeropt

constructor-initializer:  
: base ( argument-listopt )  
: this ( argument-listopt )

constructor-body:  
block  
;

Una declaración de constructor (constructor-declaration) puede incluir un conjunto de atributos (attributes) (§17), una combinación válida de los cuatro modificadores de acceso (§10.3.5) y un modificador extern (§10.6.7). No se permite que una declaración de constructor incluya un mismo modificador varias veces.

El identificador (identifier) de un declarador de constructor (constructor-declarator) debe nombrar la clase en la que se declara el constructor de instancia. Si se especifica cualquier otro nombre, se produce un error en tiempo de compilación.

La lista de parámetros formales (formal-parameter-list) opcional de un constructor de instancia está sujeta a las mismas reglas que la lista de parámetros formales (formal-parameter-list) de un método (§10.6). La lista de parámetros formales define la signatura (§3.6) de un constructor de instancia y dirige el proceso por el que la resolución de sobrecarga (§7.5.2) selecciona un constructor de instancia en particular en una invocación.

Cada uno de los tipos a los que se hace referencia en la lista de parámetros formales (formal-parameter-list) de un constructor de instancia deben tener como mínimo el mismo nivel de accesibilidad que el propio constructor (§3.5.4).

El inicializador de constructor (constructor-initializer) opcional especifica otro constructor de instancia al que hay que invocar antes de ejecutar las instrucciones que aparecen en el cuerpo del constructor (constructor-body) del primer constructor de instancia. Esta categoría se explica con más detalle en la sección §10.11.1.

Cuando la declaración de un constructor incluye un modificador extern, se dice que el constructor es un constructor externo. Debido a que la declaración de constructor externo no proporciona una implementación real, su cuerpo del constructor (constructor-body) consiste en un punto y coma. Para el resto de constructores, el cuerpo del constructor (constructor-body) consiste en un bloque (block) que especifica las instrucciones para inicializar una nueva instancia de la clase. Esto corresponde exactamente al bloque (block) de un método de instancia con un tipo de valor devuelto void (§10.6.10).

Los constructores de instancia no se heredan. Por lo tanto, una clase sólo tiene los constructores de instancia que se declaran realmente en la clase. Si una clase no contiene ninguna declaración de constructor de instancia, se proporciona automáticamente un constructor de instancia predeterminado (§10.11.4).

Los constructores de instancia se invocan mediante expresiones de creación de objetos (object-creation-expression) (§7.6.10.1) y a través de inicializadores de constructor (constructor-initializer).

### Inicializadores de constructor

Todos los constructores de instancia (excepto aquellos para la clase object) incluyen implícitamente una invocación de otro constructor de instancia inmediatamente antes del cuerpo del constructor (constructor-body). El constructor que se invoca implícitamente está determinado por el inicializador de constructor (constructor-initializer):

* Un inicializador de constructor de instancia con el formato base(lista de argumentos (argument-list)opt) causa la invocación de un constructor de instancia desde la clase base directa. Este constructor se selecciona utilizando la lista de argumentos (argument-list) y las reglas de resolución de sobrecargas de §7.5.3. El conjunto de constructores de instancia candidatos está compuesto de todos los constructores de instancia accesibles contenidos en la clase base directa, o del constructor predeterminado (§10.11.4), si en la clase base directa no se declara ningún constructores de instancia. Si el conjunto está vacío o no es posible identificar el mejor constructor de instancia, se produce un error en tiempo de compilación.
* Un inicializador de constructor de instancia con la estructura this(lista de argumentos (argument-list)opt) causa la invocación de un constructor de instancia desde la propia clase. El constructor se selecciona utilizando la lista de argumentos (argument-list) y las reglas de resolución de sobrecargas de §7.5.3. El conjunto de constructores de instancia candidatos está formado por todos los constructores de instancia accesibles declarados en la clase. Si el conjunto está vacío o no es posible identificar el mejor constructor de instancia, se produce un error en tiempo de compilación. Si la declaración de un constructor de instancia incluye un inicializador de constructor que invoca al propio constructor, se produce un error en tiempo de compilación.

Si un constructor de instancia no tiene inicializador de constructor, se proporciona implícitamente uno con la estructura base(). Por lo tanto, una declaración de constructor de instancia de la forma

C(...) {...}

equivale exactamente a

C(...): base() {...}

El ámbito de los parámetros dados en la lista de parámetros formales (formal-parameter-list) de una declaración de constructor de instancia incluye el inicializador de constructor de dicha declaración. Por lo tanto, un inicializador de constructor puede tener acceso a los parámetros del constructor. Por ejemplo:

class A  
{  
 public A(int x, int y) {}  
}

class B: A  
{  
 public B(int x, int y): base(x + y, x - y) {}  
}

Un inicializador de constructor de instancia no puede tener acceso a la instancia que está siendo creada. Por ello, se produce un error en tiempo de compilación si se hace referencia a this en una expresión de argumento del inicializador del constructor, al igual que se produce un error en tiempo de compilación si una expresión de argumento hace referencia a cualquier miembro de instancia a través del nombre simple (simple-name).

### Inicializadores de variables de instancia

Cuando un constructor de instancia no tiene inicializador de constructor, o cuando tiene uno de la forma base(...), el constructor realiza implícitamente las inicializaciones especificadas por los inicializadores de variables (variable-initializer) de los campos de instancia declarados en su clase. Esto corresponde a una secuencia de asignaciones que se ejecutan inmediatamente al entrar en el constructor y antes de invocar implícitamente al constructor de la clase base directa. Los inicializadores de variable se ejecutan en el orden textual en que aparecen en la declaración de clase.

### Ejecución de constructores

Los inicializadores de variables se transforman en instrucciones de asignación, que se ejecutan antes de la invocación del constructor de instancia de la clase base. Tal ordenamiento garantiza que todos los campos de instancia son inicializados por sus inicializadores de variable antes de que se ejecute cualquier instrucción que tenga acceso a la instancia.

Dado el ejemplo:

using System;

class A  
{  
 public A() {  
 PrintFields();  
 }

public virtual void PrintFields() {}

}

class B: A  
{  
 int x = 1;  
 int y;

public B() {  
 y = -1;  
 }

public override void PrintFields() {  
 Console.WriteLine("x = {0}, y = {1}", x, y);  
 }  
}

cuando se utiliza new B() para crear una instancia de B, el resultado que se produce es:

x = 1, y = 0

El valor de x es 1 porque el inicializador de variable se ejecuta antes de que se invoque el constructor de instancia de la clase base. Sin embargo, el valor de y es 0 (el valor predeterminado para un tipo int) porque la asignación a y no se ejecuta hasta que el constructor de la clase base devuelve el control.

Es útil considerar los inicializadores de variables de instancia y los inicializadores de constructor como instrucciones que se insertan automáticamente antes del cuerpo del constructor (constructor-body). El ejemplo

using System;  
using System.Collections;

class A  
{  
 int x = 1, y = -1, count;

public A() {  
 count = 0;  
 }

public A(int n) {  
 count = n;  
 }  
}

class B: A  
{  
 double sqrt2 = Math.Sqrt(2.0);  
 ArrayList items = new ArrayList(100);  
 int max;

public B(): this(100) {  
 items.Add("default");  
 }

public B(int n): base(n – 1) {  
 max = n;  
 }  
}

contiene varios inicializadores de variable e inicializadores de constructor de ambas formas (base y this). El código del ejemplo se corresponde con el código siguiente, donde cada comentario indica una instrucción que se inserta automáticamente (la sintaxis utilizada para invocar el constructor insertado automáticamente no es correcta, pero sirve para ilustrar el mecanismo).

using System.Collections;

class A  
{  
 int x, y, count;

public A() {  
 x = 1; // Variable initializer  
 y = -1; // Variable initializer  
 object(); // Invoke object() constructor  
 count = 0;  
 }

public A(int n) {  
 x = 1; // Variable initializer  
 y = -1; // Variable initializer  
 object(); // Invoke object() constructor  
 count = n;  
 }  
}

class B: A  
{  
 double sqrt2;  
 ArrayList items;  
 int max;

public B(): this(100) {  
 B(100); // Invoke B(int) constructor  
 items.Add("default");  
 }

public B(int n): base(n – 1) {  
 sqrt2 = Math.Sqrt(2.0); // Variable initializer  
 items = new ArrayList(100); // Variable initializer  
 A(n – 1); // Invoke A(int) constructor  
 max = n;  
 }  
}

### Constructores predeterminados

Si una clase no contiene ninguna declaración de constructores de instancia, se proporciona automáticamente un constructor de instancia predeterminado. El constructor predeterminado simplemente invoca el constructor sin parámetros de la clase base directa. Si la clase es abstracta, la accesibilidad declarada para el constructor predeterminado es protected. Si no, la accesibilidad declarada para el constructor predeterminado es public. Por lo tanto, el constructor predeterminado es siempre de la forma

protected C(): base() {}

o

public C(): base() {}

donde C es el nombre de la clase. Si la resolución de sobrecarga no puede determinar un mejor candidato único para el inicializador de constructor de la clase base, se producirá un error en tiempo de compilación.

En el ejemplo

class Message  
{  
 object sender;  
 string text;  
}

se proporciona un constructor predeterminado debido a que la clase no contiene ninguna declaración de constructor de instancia. Así, el ejemplo es exactamente equivalente a

class Message  
{  
 object sender;  
 string text;

public Message(): base() {}  
}

### Constructores Private

Cuando una clase T solo declara constructores de instancia privados, las clases externas al texto del programa de T no pueden ni derivarse ni crear directamente instancias de T. Por lo tanto, si una clase contiene sólo miembros estáticos y no se desea crear una instancia con él, puede evitarse tal instancia agregando un constructor de instancia privado que esté vacío. Por ejemplo:

public class Trig  
{  
 private Trig() {} // Prevent instantiation

public const double PI = 3.14159265358979323846;

public static double Sin(double x) {...}  
 public static double Cos(double x) {...}  
 public static double Tan(double x) {...}  
}

La clase Trig agrupa métodos y constantes relacionados, pero su intención no es crear una instancia de la misma. Por ello declara un único constructor de instancia privado vacío. Se debe declarar al menos un constructor de instancia para suprimir la generación automática de un constructor predeterminado.

### Parámetros de constructor de instancia opcionales

La forma this(...) de un inicializador de constructor se suele utilizar conjuntamente con sobrecarga para implementar parámetros de constructor de instancia opcionales. En el ejemplo

class Text  
{  
 public Text(): this(0, 0, null) {}

public Text(int x, int y): this(x, y, null) {}

public Text(int x, int y, string s) {  
 // Actual constructor implementation  
 }  
}

los dos primeros constructores de instancia sólo proporcionan los valores predeterminados para los argumentos que faltan. Ambos usan un inicializador de constructor this(...) para invocar el tercer constructor de instancia, que es el que realmente inicializa la nueva instancia. El efecto son parámetros de constructor opcionales:

Text t1 = new Text(); // Same as Text(0, 0, null)  
Text t2 = new Text(5, 10); // Same as Text(5, 10, null)  
Text t3 = new Text(5, 20, "Hello");

## Constructores static

Un constructor estático es un miembro que implementa las acciones que se requieren para inicializar un tipo de clase cerrado. Los constructores estáticos se declaran mediante declaraciones de constructores estáticos (static-constructor-declaration):

static-constructor-declaration:  
attributesopt static-constructor-modifiers identifier ( ) static-constructor-body

static-constructor-modifiers:  
externopt static  
static externopt

static-constructor-body:  
block  
;

Una declaración de constructor estático (static-constructor-declaration) puede incluir un conjunto de atributos (attributes) (§17) y un modificador extern (§10.6.7).

El identificador (identifier) de una declaración de constructor estático (static-constructor-declaration) debe nombrar la clase en la que se declara el constructor estático. Si se especifica cualquier otro nombre, se produce un error en tiempo de compilación.

Cuando la declaración de un constructor estático incluye un modificador extern, se dice que el constructor es un constructor estático externo. Debido a que la declaración de constructor estático externo no proporciona una implementación real, su cuerpo de constructor estático (static-constructor-body) consiste en un punto y coma. Para todas las demás declaraciones de constructor estático, el cuerpo del constructor estático (static-constructor-body) se compone de un bloque (block) que especifica las instrucciones que deben ejecutarse para poder inicializar la clase. Esto corresponde exactamente al cuerpo del método (method-body) estático con un tipo de valor devuelto void (§10.6.10).

Los constructores estáticos no se heredan y no pueden ser llamados directamente.

El constructor estático de un tipo de clase cerrado se ejecuta como mucho una vez en un dominio de aplicación dado. La ejecución de un constructor estático la desencadena el primero de los siguientes eventos que se produzcan en un dominio de aplicación.

* Se crea una instancia del tipo de clase.
* Se hace referencia a cualquiera de los miembros estáticos del tipo de clase.

Si una clase contiene el método Main (§3.1) en el que comienza la ejecución, el constructor estático de esta clase se ejecuta antes de que se llame al método Main.

Para inicializar un nuevo tipo de clase cerrado, primero se crea un nuevo conjunto de campos estáticos (§10.5.1) para dicho tipo cerrado. Cada uno de los campos estáticos recupera su valor predeterminado (§5.2). A continuación, los inicializadores de campos estáticos (§10.4.5.1) se ejecutan para dichos campos estáticos. Por último, se ejecuta el constructor estático.

En el ejemplo

using System;

class Test  
{  
 static void Main() {  
 A.F();  
 B.F();  
 }  
}

class A  
{  
 static A() {  
 Console.WriteLine("Init A");  
 }  
 public static void F() {  
 Console.WriteLine("A.F");  
 }  
}

class B  
{  
 static B() {  
 Console.WriteLine("Init B");  
 }  
 public static void F() {  
 Console.WriteLine("B.F");  
 }  
}

el resultado debe ser:

Init A  
A.F  
Init B  
B.F

porque la ejecución del constructor estático de A se desencadena al llamar a A.F, y la ejecución del constructor estático de B se desencadena cuando se llama a B.F.

Es posible construir dependencias circulares que permitan a los campos estáticos con inicializadores de variables ser observados en su estado de valor predeterminado.

El ejemplo

using System;

class A  
{  
 public static int X;

static A() {  
 X = B.Y + 1;  
 }  
}

class B  
{  
 public static int Y = A.X + 1;

static B() {}

static void Main() {  
 Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);  
 }  
}

produce el resultado

X = 1, Y = 2

Para ejecutar el método Main, el sistema ejecuta en primer lugar el inicializador de B.Y, antes que el constructor estático de la clase B. El inicializador de Y hace que se ejecute el constructor estático de A, porque se hace referencia al valor de A.X. Por su parte, el constructor estático de A continúa para calcular el valor de X y, para hacerlo, obtiene el valor predeterminado de Y, que es cero. Así, A.X se inicializa a 1. A continuación se completa el proceso de ejecutar los inicializadores de campos estáticos de A y el constructor estático, volviendo al cálculo del valor inicial de Y, cuyo resultado es 2.

Puesto que el constructor estático se ejecuta exactamente una vez para cada tipo de clase construida cerrada, resulta un lugar conveniente para exigir las comprobaciones en tiempo de ejecución en el parámetro de tipo que no se pueden comprobar durante la compilación mediante restricciones (§10.1.5). Por ejemplo, el siguiente tipo utiliza un constructor estático para exigir que el argumento de tipo es un enum:

class Gen<T> where T: struct  
{  
 static Gen() {  
 if (!typeof(T).IsEnum) {  
 throw new ArgumentException("T must be an enum");  
 }  
 }  
}

## Destructores

Un destructor es un miembro que implementa las acciones necesarias para destruir una instancia de una clase. Un destructor se declara mediante una declaración de destructor (destructor-declaration):

destructor-declaration:  
attributesopt externopt ~ identifier ( ) destructor-body

destructor-body:  
block  
;

Una declaración de destructor (destructor-declaration) puede incluir un conjunto de atributos (attributes) (§17).

El identificador (identifier) de una declaración de destructor (destructor-declarator) debe nombrar la clase en la que se declara el destructor. Si se especifica cualquier otro nombre, se produce un error en tiempo de compilación.

Cuando una declaración de destructor incluye un modificador extern, se dice que es un destructor externo. Debido a que la declaración de destructor externo no proporciona una implementación real, su cuerpo de destructor (destructor-body) consiste en un punto y coma. Para el resto de destructores, el cuerpo del destructor (destructor-body) consiste en un bloque (block) que especifica las instrucciones necesarias para destruir una instancia de la clase. Un cuerpo de destructor (destructor-body) corresponde exactamente al cuerpo del método (method-body) de un método de instancia con un tipo de valor devuelto void (§10.6.10).

Los destructores no se heredan. Por lo tanto, una clase sólo tiene el destructor que se puede declarar en la propia clase.

Como un destructor no puede tener parámetros, no puede ser sobrecargado; por lo tanto, una clase sólo puede tener como máximo un destructor.

Los destructores se invocan automáticamente y no se pueden invocar explícitamente. Una instancia se convierte en candidata para destrucción cuando ya ninguna parte de código puede utilizarla. La ejecución del destructor de la instancia puede ocurrir en cualquier momento una vez que la instancia se convierta en candidata para destrucción. Cuando se destruye una instancia, se llama a los destructores de su cadena de herencia en orden, de la más derivada a la menos derivada. Un destructor puede ejecutarse en cualquier subproceso. Para leer una explicación más detallada de las reglas que controlan cómo y cuándo se ejecuta un destructor, vea la sección §3.9.

El resultado del ejemplo

using System;

class A  
{  
 ~A() {  
 Console.WriteLine("A's destructor");  
 }  
}

class B: A  
{  
 ~B() {  
 Console.WriteLine("B's destructor");  
 }  
}

class Test  
{  
 static void Main() {  
 B b = new B();  
 b = null;  
 GC.Collect();  
 GC.WaitForPendingFinalizers();  
 }  
}

es

B’s destructor  
A’s destructor

ya que en una cadena de herencia los destructores se llaman en orden, de la más derivada a la menos derivada.

Los destructores se implementan invalidando el método virtual Finalize en System.Object. Los programas de C# no permiten invalidar este método o llamarlo directamente (o a invalidarlos del mismo). Por ejemplo, el programa

class A   
{  
 override protected void Finalize() {} // error

public void F() {  
 this.Finalize(); // error  
 }  
}

contiene dos errores.

El compilador se comporta como si este método, y sus invalidaciones, no existieran. Por lo tanto, este programa:

class A   
{  
 void Finalize() {} // permitted  
}

es válido y el método mostrado oculta el método Finalize de System.Object.

Para leer una explicación del comportamiento producido cuando se inicia una excepción desde un destructor, vea la sección §16.3.

## Iteradores

Un miembro de función (§7.5) implementado mediante un bloque de iteradores (§8.2) se denomina iterador.

Un bloque de iteradores se puede utilizar como cuerpo del miembro de función siempre que el tipo de valor devuelto del miembro de función correspondiente sea una de las interfaces del enumerador (§10.14.1) o una de las interfaces enumerables (§10.14.2). Puede tener lugar como cuerpo de método (method-body), cuerpo de operador (operator-body) o cuerpo de descriptor de acceso (accessor-body), mientras que los eventos, constructores de instancia, constructores estáticos y destructores no se pueden implementar como iteradores.

Cuando un miembro de función se implementa utilizando un bloque de iteradores, se genera un error en tiempo de compilación si la lista de parámetros formales del miembro de función especifica un parámetro ref u out.

### Interfaces de enumerador

Las interfaces de enumerador son la interfaz no genérica System.Collections.IEnumerator y todas las creaciones de instancias de la interfaz genérica System.Collections.Generic.IEnumerator<T>. Para ser breves, en este capítulo estas interfaces se denominarán IEnumerator e IEnumerator<T> respectivamente.

### Interfaces enumerables

Las interfaces enumerables son la interfaz no genérica System.Collections.IEnumerable y todas las creaciones de instancias de la interfaz genérica System.Collections.Generic.IEnumerable<T>. Para ser breves, en este capítulo estas interfaces se denominarán IEnumerable e IEnumerable<T> respectivamente.

### Tipo Yield

Un iterador genera una secuencia de valores, todos del mismo tipo. Este tipo se denomina tipo yield del iterador.

* El tipo de resultado de un iterador que devuelve IEnumerator o IEnumerable es object.
* El tipo de resultado de un iterador que devuelve IEnumerator<T> o IEnumerable<T> es T.

### Objetos del enumerador

Cuando un miembro de función que devuelve un tipo de interfaz del enumerador se implementa utilizando un bloque de iteradores, al invocar el miembro de función no se ejecuta inmediatamente el código del bloque de iteradores. En su lugar, se crea y se devuelve un objeto del enumerador. Este objeto encapsula el código especificado en el bloque de iteradores y la ejecución del código dentro del bloque de iteradores cuando se invoca el método MoveNext del objeto del enumerador. Un objeto del enumerador tiene las siguientes características:

* Implementa IEnumerator e IEnumerator<T>, donde T es el tipo yield del bloque de iteradores.
* Implementa System.IDisposable.
* Se inicializa con una copia de los valores de los argumentos (si los hay) y el valor de la instancia que se pasó al miembro de función.
* Tiene cuatro estados potenciales: antes, en ejecución, suspendido y después e inicialmente se encuentra en el estado antes.

Un objeto del enumerador es generalmente una instancia de una clase de enumerador generado por compilador que encapsula el código en el bloque de iteradores e implementa las interfaces del enumerador, pero también son posibles otros métodos de implementación. Si el compilador genera una clase de enumerador, ésta se anidará directa o indirectamente en la clase que contiene el miembro de función, tendrá accesibilidad privada y un número reservado para el uso del compilador (§2.4.2).

Un objeto del enumerador puede implementar más interfaces que las especificadas anteriormente.

En las siguientes secciones se describe el comportamiento exacto de los miembros MoveNext, Current y Dispose de las implementaciones de las interfaces IEnumerable y IEnumerable<T> proporcionadas por un objeto del enumerador.

Tenga en cuenta que los objetos del enumerador no admiten el método IEnumerator.Reset. La llamada a este método hace que se inicie una excepción System.NotSupportedException.

#### El método MoveNext

El método MoveNext de un objeto del enumerador encapsula el código de un bloque de iteradores. Si se invoca el método MoveNext, se ejecuta código en el bloque de iteradores y se establece la propiedad Current del objeto del enumerador de la manera apropiada. La acción concreta realizada por MoveNext depende del estado del objeto del enumerador cuando se invoca MoveNext:

* Si el estado del objeto del enumerador es antes y se invoca MoveNext:
* Cambia el estado a en ejecución.
* Se inicializan los parámetros (incluido this) del bloque de iteradores para los valores de argumentos y de instancias guardados cuando se inicializó el objeto del enumerador.
* Se ejecuta el bloque de iteradores desde el principio hasta que se interrumpe la ejecución (de la manera descrita a continuación).
* Si el estado del objeto del enumerador es en ejecución, el resultado de la invocación de MoveNext no se especifica.
* Si el estado del objeto del enumerador es suspendido y se invoca MoveNext:
* Cambia el estado a en ejecución.
* Se restauran los valores de todas las variables y parámetros locales (incluido this) a los valores guardados durante la última vez que se suspendió el bloque de iteradores. Tenga en cuenta que el contenido de todos los objetos a los que estas variables hacen referencia puede haber cambiado desde la última llamada a MoveNext.
* Se reanuda la ejecución del bloque de iteradores inmediatamente después de la instrucción yield return que provocó la suspensión de la ejecución y continúa hasta que ésta se interrumpe (de la manera descrita a continuación).
* Si el estado del objeto del enumerador es después y se invoca MoveNext, se obtiene false.

Cuando MoveNext ejecuta el bloque de iteradores, se puede interrumpir la ejecución de cuatro maneras: mediante las instrucciones yield return y yield break, detectando el final del bloque de iteradores, generando una excepción y propagándola fuera del bloque de iteradores.

* Cuando se detecta una instrucción yield return (§8.14):
* La expresión de la instrucción se evalúa, se convierte en el tipo yield y se asigna a la propiedad Current del objeto del enumerador.
* Se suspende la ejecución del cuerpo del iterador. Los valores de todas las variables y parámetros locales (incluido this) se guardan, al igual que la ubicación de esta instrucción yield return. Si la instrucción yield return se encuentra dentro de uno o más bloques try, los bloques finally asociados no se ejecutan en este momento.
* El estado del objeto del enumerador cambia a suspendido.
* El método MoveNext devuelve true a su llamador indicando que la iteración avanzó correctamente al siguiente valor.
* Cuando se detecta una instrucción yield break (§8.14):
* Si la instrucción yield break se encuentra dentro de uno o más bloques try, los bloques finally asociados se ejecutan en este momento.
* El estado del objeto del enumerador cambia a después.
* El método MoveNext devuelve false a su llamador indicando que la iteración está completa.
* Cuando se detecta el final del cuerpo del iterador:
* El estado del objeto del enumerador cambia a después.
* El método MoveNext devuelve false a su llamador indicando que la iteración está completa.
* Cuando se genera una excepción y se propaga fuera del bloque de iteradores:
* Al propagarse, la excepción habrá ejecutado los bloques finally apropiados en el cuerpo del iterador.
* El estado del objeto del enumerador cambia a después.
* La propagación de la excepción continúa al llamador del método MoveNext.

#### La propiedad Current

Las instrucciones yield return afectan a la propiedad Current del objeto del enumerador en el bloque de iteradores.

Cuando un objeto del enumerador se encuentra en el estado suspendido, el valor de Current es el valor establecido por la llamada anterior a MoveNext. Cuando un objeto del enumerador se encuentra en los estados antes, en ejecución o después, el resultado del acceso a Current no se especifica.

Para un bloque de iteradores con un tipo yield diferente de object, el acceso a Current mediante la implementación de IEnumerable del objeto del enumerador equivale al acceso a Current mediante la implementación de IEnumerator<T> del objeto del enumerador y el resultado se convierte en object.

#### El método Dispose

El método Dispose se utiliza para limpiar la iteración ya que el estado del objeto del enumerador pasa a ser después.

* Si el estado del objeto del enumerador es antes y se invoca Dispose, el estado pasa a ser después.
* Si el estado del objeto del enumerador es en ejecución, el resultado de la invocación de Dispose no se especifica.
* Si el estado del objeto del enumerador es suspendido y se invoca Dispose:
* Cambia el estado a en ejecución.
* Todos los bloques finally se ejecutan como si la última instrucción yield return ejecutada fuera una instrucción yield break. Si esto causa que se genere una excepción y se propague fuera del cuerpo del iterador, el estado del objeto del enumerador se establecerá en después y la excepción se propagará al llamador del método Dispose.
* Cambia el estado a después.
* Si el estado del objeto del enumerador es después y se invoca Dispose, no hay cambios.

### Objetos enumerables

Cuando se utiliza un bloque de iteradores para implementar un miembro de función que devuelve un tipo de interfaz enumerable y se invoca el miembro de función, el código de dicho bloque no se ejecuta inmediatamente. En su lugar se crea y se devuelve un objeto enumerable. El método GetEnumerator del objeto enumerable devuelve un objeto del enumerador que encapsula el código especificado en el bloque de iteradores y la ejecución del código del bloque de iteradores tiene lugar cuando se invoca el método MoveNext del objeto del enumerador. Un objeto enumerable tiene las siguientes características:

* Implementa IEnumerable e IEnumerable<T>, donde T es el tipo yield del bloque de iteradores.
* Se inicializa con una copia de los valores de los argumentos (si los hay) y el valor de la instancia que se pasó al miembro de función.

Generalmente, un objeto enumerable es una instancia de una clase enumerable generada por compilador que encapsula el código en el bloque de iteradores e implementa las interfaces enumerables, pero también son posibles otros métodos de implementación. Si el compilador genera una clase enumerable, ésta se anidará directa o indirectamente en la clase que contiene el miembro de función, tendrá accesibilidad privada y un nombre reservado para el uso del compilador (§2.4.2).

Un objeto enumerable puede implementar más interfaces que las especificadas anteriormente. En concreto, un objeto enumerable también puede implementar IEnumerator e IEnumerator<T> de manera que puede servir tanto de enumerable como de enumerador. En dicho tipo de implementación, la primer vez que se invoca un método GetEnumerator del objeto enumerable, se devuelve el mismo objeto enumerable. En las siguientes invocaciones de GetEnumerator del objeto enumerable, si las hay, se devuelve una copia del objeto enumerable. Por lo tanto, cada enumerador devuelto tiene su propio estado y los cambios de un enumerador no afectarán a otro.

#### El método GetEnumerator

Un objeto enumerable proporciona una implementación de los métodos GetEnumerator de las interfaces IEnumerable e IEnumerable<T>. Los dos métodos GetEnumerator comparten una implementación común que adquiere y devuelve un objeto de enumerador disponible. El objeto del enumerador se inicializa con los valores de los argumentos y el valor de la instancia guardados al inicializar el objeto; de lo contrario, el objeto del enumerador funciona de la manera descrita en la sección §10.14.4.

### Ejemplo de implementación

En esta sección se describe una posible implementación de iteradores en términos de construcciones estándar del lenguaje C#. La implementación aquí descrita se basa en los mismos principios utilizados por el compilador de Microsoft C#, pero no es una implementación obligatoria ni tampoco la única posible.

La siguiente clase Stack<T> implementa su método GetEnumerator con un iterador. El iterador enumera los elementos de la pila en orden descendente.

using System;  
using System.Collections;  
using System.Collections.Generic;

class Stack<T>: IEnumerable<T>  
{  
 T[] items;  
 int count;

public void Push(T item) {  
 if (items == null) {  
 items = new T[4];  
 }  
 else if (items.Length == count) {  
 T[] newItems = new T[count \* 2];  
 Array.Copy(items, 0, newItems, 0, count);  
 items = newItems;  
 }  
 items[count++] = item;  
 }

public T Pop() {  
 T result = items[--count];  
 items[count] = default(T);  
 return result;  
 }

public IEnumerator<T> GetEnumerator() {  
 for (int i = count - 1; i >= 0; --i) yield return items[i];  
 }  
}

El método GetEnumerator se puede traducir en una creación de instancias de una clase de enumerador generada por compilador que encapsula el código en el bloque de iteradores como en muestra a continuación.

class Stack<T>: IEnumerable<T>  
{  
 ...

public IEnumerator<T> GetEnumerator() {  
 return new \_\_Enumerator1(this);  
 }

class \_\_Enumerator1: IEnumerator<T>, IEnumerator  
 {  
 int \_\_state;  
 T \_\_current;  
 Stack<T> \_\_this;  
 int i;

public \_\_Enumerator1(Stack<T> \_\_this) {  
 this.\_\_this = \_\_this;  
 }

public T Current {  
 get { return \_\_current; }  
 }

object IEnumerator.Current {  
 get { return \_\_current; }  
 }

public bool MoveNext() {  
 switch (\_\_state) {  
 case 1: goto \_\_state1;  
 case 2: goto \_\_state2;  
 }  
 i = \_\_this.count - 1;  
 \_\_loop:  
 if (i < 0) goto \_\_state2;  
 \_\_current = \_\_this.items[i];  
 \_\_state = 1;  
 return true;  
 \_\_state1:  
 --i;  
 goto \_\_loop;  
 \_\_state2:  
 \_\_state = 2;  
 return false;  
 }

public void Dispose() {  
 \_\_state = 2;  
 }

void IEnumerator.Reset() {  
 throw new NotSupportedException();  
 }  
 }  
}

En la traducción anterior, el código del bloque de iteradores se convierte en un equipo de estado y se coloca en el método MoveNext de la clase del enumerador. La variable local i se convierte en un campo en el objeto del enumerador para que pueda continuar existiendo cuando se vuelva a invocar MoveNext.

En el siguiente ejemplo se muestra una sencilla tabla de multiplicar de enteros del 1 al 10. El método FromTo del ejemplo devuelve un objeto enumerable y se implementa utilizando un iterador.

using System;  
using System.Collections.Generic;

class Test  
{  
 static IEnumerable<int> FromTo(int from, int to) {  
 while (from <= to) yield return from++;  
 }

static void Main() {  
 IEnumerable<int> e = FromTo(1, 10);  
 foreach (int x in e) {  
 foreach (int y in e) {  
 Console.Write("{0,3} ", x \* y);  
 }  
 Console.WriteLine();  
 }  
 }  
}

El método FromTo se puede traducir en una creación de instancias de una clase enumerable generada por compilador que encapsula el código en el bloque de iteradores como en muestra a continuación.

using System;  
using System.Threading;  
using System.Collections;  
using System.Collections.Generic;

class Test  
{  
 ...

static IEnumerable<int> FromTo(int from, int to) {  
 return new \_\_Enumerable1(from, to);  
 }

class \_\_Enumerable1:  
 IEnumerable<int>, IEnumerable,  
 IEnumerator<int>, IEnumerator  
 {  
 int \_\_state;  
 int \_\_current;  
 int \_\_from;  
 int from;  
 int to;  
 int i;

public \_\_Enumerable1(int \_\_from, int to) {  
 this.\_\_from = \_\_from;  
 this.to = to;  
 }

public IEnumerator<int> GetEnumerator() {  
 \_\_Enumerable1 result = this;  
 if (Interlocked.CompareExchange(ref \_\_state, 1, 0) != 0) {  
 result = new \_\_Enumerable1(\_\_from, to);  
 result.\_\_state = 1;  
 }  
 result.from = result.\_\_from;  
 return result;  
 }

IEnumerator IEnumerable.GetEnumerator() {  
 return (IEnumerator)GetEnumerator();  
 }

public int Current {  
 get { return \_\_current; }  
 }

object IEnumerator.Current {  
 get { return \_\_current; }  
 }

public bool MoveNext() {  
 switch (\_\_state) {  
 case 1:  
 if (from > to) goto case 2;  
 \_\_current = from++;  
 \_\_state = 1;  
 return true;  
 case 2:  
 \_\_state = 2;  
 return false;  
 default:  
 throw new InvalidOperationException();  
 }  
 }

public void Dispose() {  
 \_\_state = 2;  
 }

void IEnumerator.Reset() {  
 throw new NotSupportedException();  
 }  
 }  
}

La clase enumerable implementa las interfaces tanto enumerables como de enumerador de manera que actúa como enumerable y como enumerador. La primera vez que se invoca el método GetEnumerator, se devuelve el mismo objeto enumerable. En las siguientes invocaciones de GetEnumerator del objeto enumerable, si las hay, se devuelve una copia del objeto enumerable. Por lo tanto, cada enumerador devuelto tiene su propio estado y los cambios de un enumerador no afectarán a otro. El método Interlocked.CompareExchange se usa para garantizar la operación segura para subprocesos.

Los parámetros from y to se convierten en campos en la clase enumerable. Puesto que from se modifica en el bloque de iteradores, se introduce un campo \_\_from para mantener el valor inicial dado a from en cada enumerador.

El método MoveNext inicia una excepción InvalidOperationException si se lo llama cuando \_\_state es 0. De esta manera se evita el uso del objeto enumerable como un objeto del enumerador sin llamar primero a GetEnumerator.

En el siguiente ejemplo se muestra una clase simple de árbol. La clase Tree<T> implementa su método GetEnumerator con un iterador. El iterador enumera los elementos del árbol en orden infijo.

using System;  
using System.Collections.Generic;

class Tree<T>: IEnumerable<T>  
{  
 T value;  
 Tree<T> left;  
 Tree<T> right;

public Tree(T value, Tree<T> left, Tree<T> right) {  
 this.value = value;  
 this.left = left;  
 this.right = right;  
 }

public IEnumerator<T> GetEnumerator() {  
 if (left != null) foreach (T x in left) yield x;  
 yield value;  
 if (right != null) foreach (T x in right) yield x;  
 }  
}

class Program  
{  
 static Tree<T> MakeTree<T>(T[] items, int left, int right) {  
 if (left > right) return null;  
 int i = (left + right) / 2;  
 return new Tree<T>(items[i],   
 MakeTree(items, left, i - 1),  
 MakeTree(items, i + 1, right));  
 }

static Tree<T> MakeTree<T>(params T[] items) {  
 return MakeTree(items, 0, items.Length - 1);  
 }

// The output of the program is:  
 // 1 2 3 4 5 6 7 8 9  
 // Mon Tue Wed Thu Fri Sat Sun

static void Main() {  
 Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);  
 foreach (int i in ints) Console.Write("{0} ", i);  
 Console.WriteLine();

Tree<string> strings = MakeTree(  
 "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");  
 foreach (string s in strings) Console.Write("{0} ", s);  
 Console.WriteLine();  
 }  
}

El método GetEnumerator se puede traducir en una creación de instancias de una clase de enumerador generada por compilador que encapsula el código en el bloque de iteradores como en muestra a continuación.

class Tree<T>: IEnumerable<T>  
{  
 ...

public IEnumerator<T> GetEnumerator() {  
 return new \_\_Enumerator1(this);  
 }

class \_\_Enumerator1 : IEnumerator<T>, IEnumerator  
 {  
 Node<T> \_\_this;  
 IEnumerator<T> \_\_left, \_\_right;  
 int \_\_state;  
 T \_\_current;

public \_\_Enumerator1(Node<T> \_\_this) {  
 this.\_\_this = \_\_this;  
 }

public T Current {  
 get { return \_\_current; }  
 }

object IEnumerator.Current {  
 get { return \_\_current; }  
 }

public bool MoveNext() {  
 try {  
 switch (\_\_state) {

case 0:  
 \_\_state = -1;  
 if (\_\_this.left == null) goto \_\_yield\_value;  
 \_\_left = \_\_this.left.GetEnumerator();  
 goto case 1;

case 1:  
 \_\_state = -2;  
 if (!\_\_left.MoveNext()) goto \_\_left\_dispose;  
 \_\_current = \_\_left.Current;  
 \_\_state = 1;  
 return true;

\_\_left\_dispose:  
 \_\_state = -1;  
 \_\_left.Dispose();

\_\_yield\_value:  
 \_\_current = \_\_this.value;  
 \_\_state = 2;  
 return true;

case 2:  
 \_\_state = -1;  
 if (\_\_this.right == null) goto \_\_end;  
 \_\_right = \_\_this.right.GetEnumerator();  
 goto case 3;

case 3:  
 \_\_state = -3;  
 if (!\_\_right.MoveNext()) goto \_\_right\_dispose;  
 \_\_current = \_\_right.Current;  
 \_\_state = 3;  
 return true;

\_\_right\_dispose:  
 \_\_state = -1;  
 \_\_right.Dispose();

\_\_end:  
 \_\_state = 4;  
 break;

}  
 }  
 finally {  
 if (\_\_state < 0) Dispose();  
 }  
 return false;  
 }

public void Dispose() {  
 try {  
 switch (\_\_state) {

case 1:  
 case -2:  
 \_\_left.Dispose();  
 break;

case 3:  
 case -3:  
 \_\_right.Dispose();  
 break;

}  
 }  
 finally {  
 \_\_state = 4;  
 }  
 }

void IEnumerator.Reset() {  
 throw new NotSupportedException();  
 }  
 }  
}

Los temporales generados por compilador utilizados en las instrucciones foreach se convierten en los campos \_\_left y \_\_right del objeto del enumerador. El campo \_\_state del objeto del enumerador se actualiza cuidadosamente con el fin de llamar al método Dispose() adecuado en caso de que se genere una excepción. Tenga en cuenta que no es posible escribir el código traducido con sencillas instrucciones foreach.

## Funciones asincrónicas

Un método (§10.6) o una función anónima (§7.15) con el modificador async se denomina función asincrónica. En general, el término asincrónico (async) se usa para describir cualquier tipo que tenga el modificador async.

La especificación de cualquier parámetro ref o out produce un error en tiempo de compilación para la lista de parámetros formales de una función asincrónica.

El tipo de valor devuelto (return-type) de un método asincrónico debe ser void o un tipo de tarea. Los tipos de tarea son System.Threading.Tasks.Task y los tipos construidos a partir de System.Threading.Tasks.Task<T>. Para ser breves, en este capítulo estos tipos se denominarán Task y Task<T> respectivamente. Un método asincrónico que devuelve un tipo de tarea es un tipo que devuelve tareas.

La definición exacta de los tipos de tareas la determina la implementación, pero desde el punto de vista del lenguaje, un tipo de tarea está en uno de los estados incompleto, correcto o con error. Una tarea con error registra la excepción correspondiente. Una tipo correcto Task<*T*> registra un resultado de tipo *T*. Los tipos de tareas admiten await y, por lo tanto, las tareas pueden ser los operandos de expresiones await (§7.7.7).

Una invocación de función asincrónica tiene la capacidad de suspender la evaluación por medio de expresiones await (§7.7.7) en su cuerpo. La evaluación puede reanudarse posteriormente en el punto de la expresión await en suspensión por medio de un delegado de reanudación. El delegado de reanudación es de tipo System.Action y, cuando se lo llama, la evaluación de la invocación de función asincrónica se reanudará desde la expresión await donde se quedó. El llamador actual de una invocación de función asincrónica es el llamador original si la invocación de función nunca se ha suspendido, o el llamador más reciente del delegado de reanudación en los demás casos.

### Evaluación de una función asincrónica que devuelve tareas

La invocación de una función asincrónica que devuelve tareas hace que se genere una instancia del tipo de tareas devuelta. Esto se denomina tarea devuelta de la función asincrónica. La tarea está inicialmente en un estado incompleto.

A continuación, se evalúa el cuerpo de la función asincrónica hasta que se suspenda (al llegar a una expresión await) o se termine, momento en el que el control se devuelve al llamador, junto con la tarea devuelta.

Cuando el cuerpo de la función asincrónica termina, la tarea devuelta sale del estado incompleto:

* Si el cuerpo de función termina como resultado de llegar a una instrucción return o al final del cuerpo, cualquier valor de resultado se registra en la tarea devuelta, que se pone en estado correcto.
* Si el cuerpo de función termina como resultado de una excepción no detectada (§8.9.5), la excepción se registra en la tarea devuelta que, a su vez, se pone en estado con error.

### Evaluación de una función asincrónica que devuelve void

Si el tipo de valor devuelto de la función asincrónica es void, la evaluación se diferencia de la anterior de la siguiente forma: dado que no se devuelve ninguna tarea, la función comunica la finalización y las excepciones al contexto de sincronización del subproceso actual. La definición exacta del contexto de sincronización depende de la implementación, pero es una representación de “dónde” se ejecuta el subproceso actual. Se notifica al contexto de sincronización cuando comienza la evaluación de una función asincrónica que devuelve void, se completa correctamente o hace que se inicie una excepción no detectada.

Esto permite que el contexto realice un seguimiento de cuántas funciones asincrónicas que devuelven void están en ejecución y que decida cómo propagar las excepciones que producen.

# Structs

Los structs son similares a las clases en que pueden representar estructuras de datos que pueden contener miembros de datos y miembros de función. No obstante, a diferencia de las clases, los structs son tipos de valores y no requieren asignación del montón. Una variable de un tipo struct contiene directamente los datos de struct, mientras que una variable de un tipo de clase contiene una referencia a los datos, que se conocerá posteriormente como objeto.

Los structs son particularmente útiles para estructuras de datos pequeñas que tienen semánticas de valor. Los números complejos, los puntos de un sistema de coordenadas o los pares clave-valor de un diccionario son buenos ejemplos de structs. La clave de estas estructuras de datos es que tienen pocos miembros de datos, que no necesitan utilizar herencia o identidad referencial y que pueden ser implementadas convenientemente utilizando semánticas de valor donde la asignación copia el valor en lugar de la referencia.

Como se describe en §4.1.4, los tipos simples que proporciona C#, como int, double y bool, son en realidad tipos de struct. Como estos tipos predefinidos son structs, es posible también utilizar structs y sobrecarga de operadores para implementar nuevos tipos “primitivos” en el lenguaje C#. Al final de este capítulo (§11.4) se proporcionan dos ejemplos de estos tipos.

## Declaraciones de struct

Una declaración de struct (struct-declaration) es una declaración de tipo (type-declaration) (§9.6) que declara un nuevo struct.

struct-declaration:  
attributesopt struct-modifiersopt partialopt struct identifier type-parameter-listopt  
 struct-interfacesopt type-parameter-constraints-clausesopt struct-body ;opt

Una declaración de struct (struct-declaration) consiste en un conjunto de atributos (attributes) (§17), seguido de un conjunto opcional de modificadores de struct (struct-modifiers) (§11.1.1), seguido de un modificador partial opcional, seguido de la palabra clave struct y un identificador (identifier) que da nombre al struct, seguido de una especificación opcional de lista de parámetros de tipo (type-parameter-list) (§10.1.3), seguida de una especificación opcional de interfaces de struct (struct-interfaces) (§11.1.2), seguida de una especificación opcional de cláusulas de restricciones de parámetros de tipo (type-parameter-constraints-clauses) (§10.1.5), seguida de un cuerpo de struct (struct-body) (§11.1.4), y seguido de un punto y coma opcional.

### Modificadores de struct

Una declaración de struct (struct-declaration) puede incluir opcionalmente una secuencia de modificadores de struct:

struct-modifiers:  
struct-modifier  
struct-modifiers struct-modifier

struct-modifier:  
new  
public  
protected  
internal  
private

Cuando el mismo modificador aparece varias veces en una declaración de struct, se produce un error en tiempo de compilación.

Los modificadores de una declaración de struct tienen el mismo significado que los de las declaraciones de clases (§10.1).

### Modificador parcial

El modificador partial indica que esta declaración de struct (struct-interfaces) es una declaración de tipo parcial. Varias declaraciones parciales de struct con el mismo nombre dentro de un espacio de nombres envolvente o una declaración de tipo se combinan para formar una declaración de struct, siguiendo las reglas especificadas en la sección §10.2.

### Interfaces struct

Una declaración de struct puede incluir una especificación de interfaces de struct (struct-interfaces), en cuyo caso se dice que el struct implementa directamente los tipos de interfaz dados.

struct-interfaces:  
: interface-type-list

Las implementaciones de interfaces se explican más detalladamente en §13.4.

### Cuerpo de struct

El cuerpo de struct (struct-body) de un struct define los miembros del struct.

struct-body:  
{ struct-member-declarationsopt }

## Miembros de struct

Los miembros de un struct se componen de los miembros introducidos por sus declaraciones de miembros de struct (struct-member-declaration) y los miembros heredados del tipo System.ValueType.

struct-member-declarations:  
struct-member-declaration  
struct-member-declarations struct-member-declaration

struct-member-declaration:  
constant-declaration  
field-declaration  
method-declaration  
property-declaration  
event-declaration  
indexer-declaration  
operator-declaration  
constructor-declaration  
static-constructor-declaration  
type-declaration

Excepto por las diferencias indicadas en §11.3, las descripciones de miembros de clase proporcionadas en las secciones §10.3 a §10.14 también se aplican a los miembros de struct.

## Diferencias entre clase y struct

Los structs se diferencian de las clases de diferentes maneras:

* Los structs son tipos de valor (§11.3.1).
* Todos los tipos struct se heredan implícitamente de la clase System.ValueType (§11.3.2).
* La asignación a una variable de un tipo struct crea una copia del valor que se asigne (§11.3.3).
* El valor predeterminado de un struct es el valor producido al establecer todos los campos de tipos de valores en su valor predeterminado, y todos los campos de tipos de referencia en null (§11.3.4).
* Las operaciones boxing y unboxing se utilizan para realizar la conversión entre un tipo struct y un tipo object (§11.3.5).
* El significado de this es diferente para los structs (§7.6.7).
* Las declaraciones de campos de instancia para un struct no pueden incluir inicializadores de variables (§11.3.7).
* Un struct no puede declarar un constructor de instancia sin parámetros (§11.3.8).
* No está permitido que un struct declare un destructor (§11.3.9).

### Semánticas de valor

Los structs son tipos de valor (§4.1) y se dice que tienen semánticas de valor. Las clases, por el contrario, son tipos de referencia (§4.2) y se dice que tienen semánticas de referencia.

Una variable de un tipo struct contiene directamente los datos de struct, mientras que una variable de un tipo de clase contiene una referencia a los datos, que se conocerá posteriormente como objeto. Cuando un struct B contiene un campo de instancia de tipo A y A es un tipo de struct, supone un error en tiempo de compilación que A dependa de B o un tipo construido a partir de B. Un struct X tiene una dependencia directa con un struct Y si X contiene un campo de instancia del tipo Y. Dada esta definición, el conjunto completo de structs de las cuales depende un struct es el cierre transitivo de la relación de dependencia directa. Por ejemplo:

struct Node  
{  
 int data;

Node next; // error, Node directly depends on itself

}

es un error porque Node contiene un campo de instancia de su propio tipo. Otro ejemplo

struct A { B b; }

struct B { C c; }

struct C { A a; }

es un error porque cada uno de los tipos A, B y C dependen entre sí.

En el caso de las clases, es posible que dos variables hagan referencia al mismo objeto y, por tanto, que las operaciones en una variable afecten al objeto al que hace referencia la otra variable. En el caso de los structs, cada variable tiene su propia copia de los datos (exceptuando las variables de los parámetros ref y out), de manera que no posible que las operaciones de una afecten a la otra. Dado que los structs no son tipos de referencia, no es posible que los valores de un tipo struct sean null.

Dada la declaración

struct Point  
{  
 public int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

el fragmento de código

Point a = new Point(10, 10);  
Point b = a;  
a.x = 100;  
System.Console.WriteLine(b.x);

devuelve como resultado el valor 10. La asignación de a a b crea una copia del valor y por tanto b no se ve afectada por la asignación a a.x. En cambio, si se hubiera declarado Point como clase, el resultado sería 100 puesto que a y b harían referencia al mismo objeto.

### Herencia

Todos los tipos struct se heredan implícitamente de la clase System.ValueType, que, a su vez, se hereda de la clase object. Una declaración de struct puede especificar una lista de interfaces implementadas, pero no es posible que especifique una clase base.

Los tipos de struct nunca son abstractos y son siempre sellados implícitamente. Por lo tanto, los modificadores abstract y sealed no están permitidos en una declaración de struct.

Debido a que los structs no permiten la herencia, la accesibilidad declarada de un miembro de struct no puede ser protected ni protected internal.

Los miembros de función de un struct no pueden ser abstract ni virtual, y el modificador override se permite solo para invalidar los métodos heredados de System.ValueType.

### Asignación

La asignación a una variable de un tipo struct crea una copia del valor que se asigne. Esto difiere de la asignación a una variable de un tipo de clase, que copia la referencia pero no el objeto identificado por la referencia.

De forma similar a una asignación, cuando se pasa un struct como parámetro de valor o se devuelve como resultado de un miembro de función, se crea una copia del struct. Un struct se puede pasar por referencia a un miembro de función utilizando un parámetro ref u out.

Cuando una propiedad o un indizador de un struct es el destino de una asignación, la expresión de instancia asociada con el acceso a la propiedad o indizador debe estar clasificada como una variable. Si la expresión de instancia está clasificada como un valor, se produce un error de compilación. Esta categoría se explica con más detalle en la sección §7.17.1.

### Valores predeterminados

Como se describe en la sección §5.2, cuando se crean algunos tipos de variables se inicializan automáticamente a sus valores predeterminados. Para variables de tipos de clase y otros tipos de referencia, el valor predeterminado es null. Sin embargo, como los structs son tipos de valor que no pueden ser null, el valor predeterminado de un struct es el valor producido al establecer todos los campos de tipos de valores a sus valores predeterminados y todos los campos de tipo de referencia a null.

Haciendo referencia al struct Point declarada anteriormente, en el ejemplo

Point[] a = new Point[100];

se inicializa cada Point de la matriz al valor producido cuando se establecen los campos x e y a cero.

El valor predeterminado de un struct corresponde al valor devuelto por el constructor predeterminado del struct (§4.1.2). A diferencia de una clase, un struct no permite declarar un constructor de instancia sin parámetros. En su lugar, cada struct tiene implícitamente un constructor de instancia sin parámetros que siempre devuelve el valor que resulta de establecer todos los campos de tipos de valores a sus valores predeterminados y todos los campos de tipo de referencia a null.

Los structs deben diseñarse considerando que el estado de inicialización predeterminado sea un estado válido. En el ejemplo

using System;

struct KeyValuePair  
{  
 string key;  
 string value;

public KeyValuePair(string key, string value) {  
 if (key == null || value == null) throw new ArgumentException();  
 this.key = key;  
 this.value = value;  
 }  
}

el constructor de instancia definido por el usuario evita los valores nulos sólo donde es llamado explícitamente. En los casos en los que una variable KeyValuePair esté sujeta a una inicialización de valor predeterminado, los campos key y value serán nulos y el struct debe estar preparado para controlar este estado.

### Conversiones boxing y unboxing

Un valor de un tipo de clase se puede convertir a un tipo object o a un tipo de interfaz que es implementada por la clase simplemente tratando la referencia como otro tipo en tiempo de compilación. Igualmente, un valor del tipo object o un valor de un tipo de interfaz se puede convertir de nuevo a un tipo de clase sin cambiar la referencia (pero en este caso es necesaria una comprobación de tipo en tiempo de ejecución).

Dado que los structs no son tipos de referencia, estas operaciones se implementan de forma diferente para los tipos struct. Cuando se convierte un valor de un tipo struct a un tipo object o a un tipo de interfaz que es implementada por el struct, tiene lugar una operación boxing. De la misma forma, cuando un valor de tipo object o un valor de tipo de interfaz se vuelve a convertir a un tipo struct, tiene lugar una operación unboxing. Una diferencia fundamental con las mismas operaciones en los tipos de clase es que las conversiones boxing y unboxing copian el valor de struct en la instancia o fuera de la instancia a la que se ha aplicado la conversión boxing. Por lo tanto, después de una operación boxing o unboxing, los cambios realizados en el struct a la que se ha aplicado la conversión unboxing no se reflejan en el struct donde se ha aplicado la conversión boxing.

Cuando un tipo struct invalida un método virtual heredado de System.Object (como Equals, GetHashCode o ToString), la invocación del método virtual mediante una instancia del tipo struct no provoca la conversión boxing. Esto es cierto incluso cuando se utiliza el struct como un parámetro de tipo y la invocación tiene lugar mediante una instancia del tipo de parámetro de tipo. Por ejemplo:

using System;

struct Counter  
{  
 int value;

public override string ToString() {  
 value++;  
 return value.ToString();  
 }  
}

class Program  
{  
 static void Test<T>() where T: new() {  
 T x = new T();  
 Console.WriteLine(x.ToString());  
 Console.WriteLine(x.ToString());  
 Console.WriteLine(x.ToString());  
 }

static void Main() {  
 Test<Counter>();  
 }  
}

El resultado del programa es:

1  
2  
3

Aunque el hecho de que ToString tenga efectos secundarios denota un estilo incorrecto, en el ejemplo se demuestra que no se realizó ninguna conversión boxing para las tres invocaciones de x.ToString().

De manera similar, la conversión boxing nunca tiene lugar de manera implícita cuando se obtiene acceso a un miembro en un parámetro de tipo restringido. Por ejemplo, supongamos que una interfaz ICounter contiene un método Increment que se puede utilizar para modificar un valor. Si ICounter se utiliza como una restricción, la implementación del método Increment se llama con una referencia a la variable en la que se llamó a Increment y nunca una copia con conversión boxing aplicada:

using System;

interface ICounter  
{  
 void Increment();  
}

struct Counter: ICounter  
{  
 int value;

public override string ToString() {  
 return value.ToString();  
 }

void ICounter.Increment() {  
 value++;  
 }  
}

class Program  
{  
 static void Test<T>() where T: ICounter, new() {  
 T x = new T();  
 Console.WriteLine(x);  
 x.Increment(); // Modify x  
 Console.WriteLine(x);  
 ((ICounter)x).Increment(); // Modify boxed copy of x  
 Console.WriteLine(x);  
 }

static void Main() {  
 Test<Counter>();  
 }  
}

La primera llamada a Increment modifica el valor en la variable x. Esto no es equivalente a la segunda llamada a Increment, que modifica el valor en una copia de x a la que se aplicó una conversión boxing. Por lo tanto, el resultado del programa es:

0  
1  
1

Para obtener información detallada sobre boxing y unboxing, vea §4.3.

### Significado de this

Dentro de un constructor de instancia o de un miembro de función de una clase, this está clasificado como un valor. Por lo tanto, mientras que this se puede utilizar para hacer referencia a la instancia para la que fue invocado el miembro de función, no es posible realizar una asignación a this en un miembro de función de una clase.

En un constructor de instancia de un struct, this corresponde a un parámetro out del tipo struct, y en un miembro de función de instancia de un struct, this corresponde a un parámetro ref del tipo struct. En ambos casos, this está clasificado como una variable, y es posible modificar toda el struct para el que fue invocado el miembro de función realizando una asignación a this o pasando this como un parámetro ref u out.

### Inicializadores de campo

Como se describe en la sección §11.3.4, el valor predeterminado de un struct se compone del valor que resulta de establecer todos los campos de tipos de valores a sus valores predeterminados y de todos los campos de tipo de referencia a null. Por esta razón, un struct no permite declaraciones de campos de instancia para incluir inicializadores de variables. Esta restricción sólo se aplica a los campos de instancia. Los campos estáticos de un struct pueden incluir inicializadores de variable.

En el ejemplo

struct Point  
{  
 public int x = 1; // Error, initializer not permitted  
 public int y = 1; // Error, initializer not permitted  
}

hay un error porque la declaración del campo de instancia incluye inicializadores de variables.

### Constructores

A diferencia de una clase, un struct no permite declarar un constructor de instancia sin parámetros. En su lugar, cada struct tiene implícitamente un constructor de instancia sin parámetros que siempre devuelve el valor que resulta de establecer todos los campos de tipos de valores a sus valores predeterminados y todos los campos de tipo de referencia a null (§4.1.2). Un struct puede declarar constructores de instancia con parámetros. Por ejemplo:

struct Point  
{  
 int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

Dada la declaración anterior, las instrucciones

Point p1 = new Point();

Point p2 = new Point(0, 0);

ambos crean un Point con x e y inicializados a cero.

Un constructor de instancia de struct no puede incluir un inicializador de constructor con la forma base(...).

Si el constructor de la instancia de struct no especifica un inicializador de constructor, la variable this corresponde a un parámetro out del tipo struct, y de forma similar a un parámetro out, this debe ser asignado de manera definitiva (§5.3) a cada ubicación a la que vuelva el constructor. Si el constructor de instancia del struct especifica un inicializador de constructor, la variable this corresponde a un parámetro ref del tipo struct, y de forma similar a un parámetro ref, this se considera definitivamente asignado en la entrada al cuerpo del constructor. Observe la implementación del constructor de instancia siguiente:

struct Point  
{  
 int x, y;

public int X {  
 set { x = value; }  
 }

public int Y {  
 set { y = value; }  
 }

public Point(int x, int y) {  
 X = x; // error, this is not yet definitely assigned  
 Y = y; // error, this is not yet definitely assigned  
 }  
}

No puede llamarse a ninguna función de miembro de instancia (incluyendo los descriptores de acceso set para las propiedades X e Y) hasta que todos los campos del struct que está construyéndose se hayan asignado definitivamente. Observe, no obstante, que si Point fuera una clase en lugar de un struct, estaría permitida la implementación del constructor de instancia.

### Destructores

No está permitido que un struct declare un destructor.

### Constructores static

Los constructores estáticos (static) para structs siguen la mayoría de las mismas reglas que las clases. La ejecución de un constructor estático para un tipo struct la desencadena el primero de los siguientes eventos que se produzcan en un dominio de aplicación:

* Se hace referencia a un miembro estático del tipo struct.
* Se llama a un constructor explícitamente declarado del tipo struct.

La creación de valores predeterminados (§11.3.4) de tipo struct no desencadena el constructor estático. (Un ejemplo de esto es el valor inicial de los elementos de una matriz.)

## Ejemplos de struct

A continuación, se muestran dos ejemplos significativos de la utilización de tipos struct para crear tipos que puedan utilizarse de manera similar a los tipos predefinidos del lenguaje, pero con semánticas modificadas.

### Tipo entero de base de datos

El struct DBInt siguiente implementa un tipo entero que puede representar el conjunto completo de valores del tipo int, además de un estado adicional que indica un valor desconocido. Un tipo de estas características se utiliza normalmente en bases de datos.

using System;

public struct DBInt  
{  
 // The Null member represents an unknown DBInt value.

public static readonly DBInt Null = new DBInt();

// When the defined field is true, this DBInt represents a known value  
 // which is stored in the value field. When the defined field is false,  
 // this DBInt represents an unknown value, and the value field is 0.

int value;  
 bool defined;

// Private instance constructor. Creates a DBInt with a known value.

DBInt(int value) {  
 this.value = value;  
 this.defined = true;  
 }

// The IsNull property is true if this DBInt represents an unknown value.

public bool IsNull { get { return !defined; } }

// The Value property is the known value of this DBInt, or 0 if this  
 // DBInt represents an unknown value.

public int Value { get { return value; } }

// Implicit conversion from int to DBInt.

public static implicit operator DBInt(int x) {  
 return new DBInt(x);  
 }

// Explicit conversion from DBInt to int. Throws an exception if the  
 // given DBInt represents an unknown value.

public static explicit operator int(DBInt x) {  
 if (!x.defined) throw new InvalidOperationException();  
 return x.value;  
 }

public static DBInt operator +(DBInt x) {  
 return x;  
 }

public static DBInt operator -(DBInt x) {  
 return x.defined ? -x.value : Null;  
 }

public static DBInt operator +(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value + y.value: Null;  
 }

public static DBInt operator -(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value - y.value: Null;  
 }

public static DBInt operator \*(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value \* y.value: Null;  
 }

public static DBInt operator /(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value / y.value: Null;  
 }

public static DBInt operator %(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value % y.value: Null;  
 }

public static DBBool operator ==(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value == y.value: DBBool.Null;  
 }

public static DBBool operator !=(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value != y.value: DBBool.Null;  
 }

public static DBBool operator >(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value > y.value: DBBool.Null;  
 }

public static DBBool operator <(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value < y.value: DBBool.Null;  
 }

public static DBBool operator >=(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value >= y.value: DBBool.Null;  
 }

public static DBBool operator <=(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value <= y.value: DBBool.Null;  
 }

public override bool Equals(object obj) {  
 if (!(obj is DBInt)) return false;  
 DBInt x = (DBInt)obj;  
 return value == x.value && defined == x.defined;  
 }

public override int GetHashCode() {  
 return value;  
 }

public override string ToString() {  
 return defined? value.ToString(): “DBInt.Null”;  
 }  
}

### Tipo booleano de base de datos

El struct DBBool siguiente implementa un tipo lógico de tres valores. Los valores posibles de este tipo son DBBool.True, DBBool.False y DBBool.Null, donde el miembro Null indica un valor desconocido. Estos tipos lógicos de tres valores se utilizan con frecuencia en bases de datos.

using System;

public struct DBBool  
{  
 // The three possible DBBool values.

public static readonly DBBool Null = new DBBool(0);  
 public static readonly DBBool False = new DBBool(-1);  
 public static readonly DBBool True = new DBBool(1);

// Private field that stores –1, 0, 1 for False, Null, True.

sbyte value;

// Private instance constructor. The value parameter must be –1, 0, or 1.

DBBool(int value) {  
 this.value = (sbyte)value;  
 }

// Properties to examine the value of a DBBool. Return true if this  
 // DBBool has the given value, false otherwise.

public bool IsNull { get { return value == 0; } }

public bool IsFalse { get { return value < 0; } }

public bool IsTrue { get { return value > 0; } }

// Implicit conversion from bool to DBBool. Maps true to DBBool.True and  
 // false to DBBool.False.

public static implicit operator DBBool(bool x) {  
 return x? True: False;  
 }

// Explicit conversion from DBBool to bool. Throws an exception if the  
 // given DBBool is Null, otherwise returns true or false.

public static explicit operator bool(DBBool x) {  
 if (x.value == 0) throw new InvalidOperationException();  
 return x.value > 0;  
 }

// Equality operator. Returns Null if either operand is Null, otherwise  
 // returns True or False.

public static DBBool operator ==(DBBool x, DBBool y) {  
 if (x.value == 0 || y.value == 0) return Null;  
 return x.value == y.value? True: False;  
 }

// Inequality operator. Returns Null if either operand is Null, otherwise  
 // returns True or False.

public static DBBool operator !=(DBBool x, DBBool y) {  
 if (x.value == 0 || y.value == 0) return Null;  
 return x.value != y.value? True: False;  
 }

// Logical negation operator. Returns True if the operand is False, Null  
 // if the operand is Null, or False if the operand is True.

public static DBBool operator !(DBBool x) {  
 return new DBBool(-x.value);  
 }

// Logical AND operator. Returns False if either operand is False,  
 // otherwise Null if either operand is Null, otherwise True.

public static DBBool operator &(DBBool x, DBBool y) {  
 return new DBBool(x.value < y.value? x.value: y.value);  
 }

// Logical OR operator. Returns True if either operand is True, otherwise  
 // Null if either operand is Null, otherwise False.

public static DBBool operator |(DBBool x, DBBool y) {  
 return new DBBool(x.value > y.value? x.value: y.value);  
 }

// Definitely true operator. Returns true if the operand is True, false  
 // otherwise.

public static bool operator true(DBBool x) {  
 return x.value > 0;  
 }

// Definitely false operator. Returns true if the operand is False, false  
 // otherwise.

public static bool operator false(DBBool x) {  
 return x.value < 0;  
 }

public override bool Equals(object obj) {  
 if (!(obj is DBBool)) return false;  
 return value == ((DBBool)obj).value;  
 }

public override int GetHashCode() {  
 return value;  
 }

public override string ToString() {  
 if (value > 0) return "DBBool.True";  
 if (value < 0) return "DBBool.False";  
 return "DBBool.Null";  
 }  
}

# Matrices

Una matriz es una estructura de datos que contiene una serie de variables a las que se obtiene acceso a través de índices calculados. Las variables contenidas en una matriz, también conocidas como elementos de la matriz, son todas del mismo tipo, denominado tipo de elemento de la matriz.

Una matriz tiene un rango que determina el número de índices asociados a cada elemento de la matriz. El rango de una matriz también se conoce como número de dimensiones de la matriz. Una matriz con un rango de uno se denomina matriz unidimensional. Una matriz con un rango mayor que uno se denomina matriz multidimensional. Las matrices multidimensionales de un determinado tamaño se suelen denominar matrices bidimensionales, tridimensionales, etc.

Cada dimensión de una matriz tiene una longitud asociada, que es un número entero mayor o igual a cero. Las longitudes de dimensión no forman parte del tipo de matriz, sino que se establecen cuando se crea una instancia del tipo de matriz en tiempo de ejecución. La longitud de una dimensión determina el intervalo válido de índices para esa dimensión: para una dimensión de longitud N, los índices pueden tomar valores de 0 a N – 1 inclusive. El número total de elementos de una matriz es el producto de las longitudes de cada dimensión de la matriz. Si alguna de las dimensiones de una matriz tiene una longitud cero, se dice que la matriz está vacía.

Los elementos de una matriz pueden ser de cualquier tipo, incluido un tipo de matriz.

## Tipos de matriz

Un tipo de matriz se escribe como un tipo que no es de matriz (non-array-type) seguido de uno o más especificadores de rango (rank-specifier):

array-type:  
non-array-type rank-specifiers

non-array-type:  
type

rank-specifiers:  
rank-specifier  
rank-specifiers rank-specifier

rank-specifier:  
[ dim-separatorsopt ]

dim-separators:  
,  
dim-separators ,

Un tipo que no es de matriz (non-array-type) es cualquier tipo que no es en sí mismo un tipo de matriz (array-type).

El rango de un tipo de matriz viene dado por el especificador de rango (rank-specifier) situado en el extremo izquierdo del tipo de matriz (array-type): así, un especificador de rango indica que la matriz posee un rango de uno más el número de tokens "," del especificador de rango.

El tipo de elemento de un tipo de matriz es el tipo resultante de eliminar el especificador de rango (rank-specifier) del extremo izquierdo:

* Un tipo de matriz con la estructura T[R] es una matriz de rango R y un tipo de elemento T que no sea de matriz.
* Un tipo de matriz con el formato T[R][R1]...[RN] es una matriz de rango R y un tipo de elemento T[R1]...[RN].

Lo que sucede es que los especificadores de rango (rank-specifier) se leen de izquierda a derecha, antes del tipo de elemento final que no es de matriz. El tipo int[][,,][,] es una matriz unidimensional que contiene matrices tridimensionales de matrices bidimensionales de tipo int.

En tiempo de ejecución, el valor de un tipo de matriz puede ser null o una referencia a una instancia de dicho tipo.

### Tipo System.Array

El tipo System.Array es el tipo base abstracto de todos los tipos de matrices. Se produce una conversión de referencia implícita (§6.1.6) entre cualquier tipo de matriz y un tipo System.Array, así como una conversión de referencia explícita (§6.2.4) de System.Array a cualquier tipo de matriz. Observe que System.Array no es en sí mismo un tipo de matriz (array-type). Más bien es un tipo de clase (class-type) del que se derivan todos los tipos de matriz (array-type).

En tiempo de ejecución, un valor de tipo System.Array puede ser null o una referencia a una instancia de cualquier tipo de matriz.

### Matrices y la interfaz IList genérica

Una matriz unidimensional T[] implementa la interfaz System.Collections.Generic.IList<T> (IList<T> para mayor brevedad) y sus interfaces base. Por lo tanto, hay una conversión implícita de T[] a IList<T> y sus interfaces base. Además, si hay una conversión implícita de referencia de S a T, entonces S[] implementa IList<T> y hay una conversión implícita de referencia de S[] a IList<T> y sus interfaces base (§6.1.6). Si hay una conversión explícita de referencia de S a T entonces hay una conversión explícita de referencia de S[] a IList<T> y sus interfaces base (§6.2.4). Por ejemplo:

using System.Collections.Generic;

class Test  
{  
 static void Main() {  
 string[] sa = new string[5];  
 object[] oa1 = new object[5];  
 object[] oa2 = sa;

IList<string> lst1 = sa; // Ok  
 IList<string> lst2 = oa1; // Error, cast needed  
 IList<object> lst3 = sa; // Ok  
 IList<object> lst4 = oa1; // Ok

IList<string> lst5 = (IList<string>)oa1; // Exception  
 IList<string> lst6 = (IList<string>)oa2; // Ok  
 }  
}

La asignación lst2 = oa1 genera un error en tiempo de compilación, puesto que la conversión de object[] a IList<string> es una conversión explícita, no implícita. La conversión (IList<string>)oa1 iniciará una excepción en tiempo de ejecución puesto que oa1 hace referencia a object[] y no a string[]. No obstante, la conversión (IList<string>)oa2 no iniciará una excepción en tiempo de ejecución puesto que oa2 hace referencia a string[].

Si hay una conversión implícita o explícita de referencia de S[] a IList<T>, también hay una conversión explícita de referencia de IList<T> y sus interfaces base a S[] (§6.2.4).

Cuando un tipo de matriz S[] implementa IList<T>, algunos de los miembros de la interfaz implementada pueden iniciar excepciones. El comportamiento concreto de la implementación de la interfaz se escapa al ámbito de esta especificación.

## Creación de matrices

Las instancias de matriz se crean mediante expresiones de creación de matrices (array-creation-expression) (§7.6.10.4) o mediante declaraciones de campo o variable local que incluyen un inicializador de matriz (array-initializer) (§12.6).

Cuando se crea una instancia de matriz, se establecen el rango y longitud de cada dimensión y a continuación permanecen constante por toda la duración de la instancia. Es decir, no es posible cambiar el rango de una instancia de matriz existente ni cambiar el tamaño de sus dimensiones.

Una instancia de matriz es siempre de tipo de matriz. El tipo System.Array es un tipo abstracto del que no se pueden crear instancias.

Los elementos de matrices creadas mediante expresiones de creación de matrices (array-creation-expression) se inicializan siempre con sus valores predeterminados (§5.2).

## Acceso a los elementos de matriz

El acceso a los elementos de una matriz se obtiene mediante expresiones de acceso a elementos (element-access) (§7.6.6.1) con el formato A[I1, I2, ..., IN], donde A es una expresión de un tipo de matriz y cada IX es una expresión de tipo int, uint, long, ulong o de un tipo que puede convertirse de manera implícita a uno o varios de estos tipos. El resultado de un acceso a elemento de matriz es una variable, en concreto el elemento de matriz seleccionado por los índices.

Los elementos de una matriz se pueden enumerar mediante una instrucción foreach (§8.8.4).

## Miembros de matriz

Cada tipo de matriz hereda los miembros declarados por el tipo System.Array.

## Covarianza de matrices

Por cada dos tipos de referencia (reference-type) A y B, si existe una conversión de referencia implícita (§6.1.6) o explícita (§6.2.4) entre A y B, también tiene lugar la misma conversión de referencia entre el tipo de matriz A[R] y el tipo de matriz B[R], donde R es cualquier especificador de rango (rank-specifier), si bien es el mismo para ambos tipos de matriz. Esta relación se conoce como covarianza matricial. La covarianza matricial significa, en concreto, que un valor de un tipo de matriz A[R] puede ser en realidad una referencia a una instancia de un tipo de matriz B[R], siempre que exista una conversión de referencia implícita entre B y A.

Debido a la covarianza de matrices, las asignaciones a elementos de matrices de tipo de referencia incluyen una comprobación en tiempo de ejecución que garantiza que el valor que se asigna al elemento de matriz es realmente de un tipo permitido (§7.17.1). Por ejemplo:

class Test  
{  
 static void Fill(object[] array, int index, int count, object value) {  
 for (int i = index; i < index + count; i++) array[i] = value;  
 }

static void Main() {  
 string[] strings = new string[100];  
 Fill(strings, 0, 100, "Undefined");  
 Fill(strings, 0, 10, null);  
 Fill(strings, 90, 10, 0);  
 }  
}

La asignación a array[i] en el método Fill incluye implícitamente una comprobación en tiempo de ejecución, que garantiza que el objeto al que hace referencia value es null o bien una instancia compatible con el tipo de elemento real de array. En Main, las dos primeras invocaciones de Fill se ejecutan correctamente, pero la tercera produce una excepción System.ArrayTypeMismatchException al ejecutarse la primera asignación a array[i]. La excepción se produce porque no se puede almacenar un valor int boxed en una matriz string.

La covarianza de matrices no se extiende específicamente a las matrices de tipo de valor (value-type). Por ejemplo, no existe ninguna conversión que permita que int[] se trate como un object[].

## Inicializadores de matrices

Los inicializadores de matrices se pueden especificar en declaraciones de campo (§10.5), declaraciones de variable local (§8.5.1) y expresiones de creación de matrices (§7.6.10.4):

array-initializer:  
{ variable-initializer-listopt }  
{ variable-initializer-list , }

variable-initializer-list:  
variable-initializer  
variable-initializer-list , variable-initializer

variable-initializer:  
expression  
array-initializer

Un inicializador de matrices se compone de una secuencia de inicializadores de variable, que figuran entre los tokens “{” y “}” y separados por “,”. Cada inicializador de variable es una expresión o, en el caso de una matriz multidimensional, un inicializador de matriz anidado.

El contexto en que se utiliza un inicializador de matriz determina el tipo de la matriz que se inicializa. En una expresión de creación de matrices, el tipo de matriz precede inmediatamente al inicializador, o se infiere de las expresiones del inicializador de matrices. En una declaración de campo o de variable, el tipo de matriz es el tipo del campo o variable que se está declarando. Cuando se utiliza un inicializador de matriz en una declaración de campo o de variable, como:

int[] a = {0, 2, 4, 6, 8};

equivale a la siguiente expresión de creación de matriz:

int[] a = new int[] {0, 2, 4, 6, 8};

Para una matriz unidimensional, el inicializador de matriz debe estar compuesto de una secuencia de expresiones que sean compatibles en materia de asignación con el tipo de elemento de la matriz. Las expresiones inicializan los elementos de matriz por orden creciente, comenzando a partir del elemento en el índice cero. El número de expresiones del inicializador de matriz determina la longitud de la instancia de matriz que se está creando. Por ejemplo, el inicializador de matriz anterior crea una instancia int[] de longitud 5 y a continuación inicializa la instancia con los siguientes valores:

a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;

Para una matriz multidimensional, el inicializador de matriz debe tener tantos niveles de anidamiento como dimensiones tenga la matriz. El nivel de anidamiento superior se corresponde con la dimensión situada en el extremo izquierdo y el nivel de anidamiento inferior se corresponde con la dimensión situada en el extremo derecho. La longitud de cada una de las dimensiones de la matriz está determinada por el número de elementos del nivel de anidamiento correspondiente en el inicializador de la matriz. Por cada inicializador de matriz anidado, el número de elementos debe ser igual al de los demás inicializadores de matriz que están situados en el mismo nivel. El ejemplo:

int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};

crea una matriz bidimensional con una longitud de cinco para la dimensión del extremo izquierdo y una longitud de dos para la dimensión del extremo derecho:

int[,] b = new int[5, 2];

y a continuación se inicializa la instancia de la matriz con los valores siguientes:

b[0, 0] = 0; b[0, 1] = 1;  
b[1, 0] = 2; b[1, 1] = 3;  
b[2, 0] = 4; b[2, 1] = 5;  
b[3, 0] = 6; b[3, 1] = 7;  
b[4, 0] = 8; b[4, 1] = 9;

Si a una dimensión diferente a la situada más a la derecha se le asigna una longitud cero, se asume que las dimensiones posteriores también tienen una longitud cero. El ejemplo:

int[,] c = {};

crea una matriz bidimensional con una longitud de cero tanto para la dimensión situada más a la izquierda como para la dimensión situada más a la derecha:

int[,] c = new int[0, 0];

Cuando una expresión de creación de matriz incluye tanto longitudes de dimensión explícitas como un inicializador de matriz, las longitudes deben ser expresiones constantes, y el número de elementos en cada nivel de anidamiento debe coincidir con la longitud de dimensión correspondiente. A continuación se describen algunos ejemplos:

int i = 3;  
int[] x = new int[3] {0, 1, 2}; // OK  
int[] y = new int[i] {0, 1, 2}; // Error, i not a constant  
int[] z = new int[3] {0, 1, 2, 3}; // Error, length/initializer mismatch

Aquí, el inicializador de y es un error de compilación porque la expresión de longitud de dimensión no es una constante, y el inicializador de z es un error de compilación porque la longitud y el número de elementos del inicializador no coinciden.

# Interfaces

Una interfaz define un contrato. Una clase o struct que implementa una interfaz debe adherirse a su contrato. Una interfaz puede heredar de varias interfaces base, y una clase o struct puede implementar varias interfaces.

Las interfaces pueden contener métodos, propiedades, eventos e indizadores. La propia interfaz no proporciona implementaciones para los miembros que define. La interfaz sólo especifica los miembros que las clases o structs deben proporcionar, y que implementan la interfaz.

## Declaraciones de interfaz

Una declaración de interfaz (interface-declaration) es una declaración de tipo (type-declaration) (§9.6) que declara un nuevo tipo de interfaz.

interface-declaration:  
attributesopt interface-modifiersopt partialopt interface   
 identifier variant-type-parameter-listopt interface-baseopt  
 type-parameter-constraints-clausesopt interface-body ;opt

Una declaración de interfaz (interface-declaration) consiste en un conjunto de atributos (attributes) (§17) opcionales, seguido de un conjunto opcional de modificadores de interfaz (interface-modifiers) (§13.1.1), seguido de un modificador partial opcional, seguido de la palabra clave interface y un identificador (identifier) que da nombre a la interfaz, seguido de una especificación opcional de lista de parámetros de tipo variante (variant*-type-parameter-list*) (§13.1.3), seguido de una especificación opcional de base de interfaz (interface-base) (§13.1.4), seguida de una especificación opcional de cláusulas de restricciones de parámetros de tipo (type-parameter-constraints-clauses) (§10.1.5), seguida de un cuerpo de interfaz (interface-body) (§13.1.5), y seguido de un punto y coma opcional.

### Modificadores de interfaz

Una declaración de interfaz (interface-declaration) puede incluir opcionalmente una secuencia de modificadores de interfaz:

interface-modifiers:  
interface-modifier  
interface-modifiers interface-modifier

interface-modifier:  
new  
public  
protected  
internal  
private

Cuando el mismo modificador aparece varias veces en una declaración de interfaz, se produce un error en tiempo de compilación.

El modificador new solo se permite en interfaces definidas dentro de una clase. Especifica que la interfaz oculta un miembro heredado con el mismo nombre, como se describe en §10.3.4.

Los modificadores public, protected, internal y private controlan la accesibilidad de la interfaz. Dependiendo del contexto en que se produzca la declaración de interfaz, puede que sólo se permitan algunos de estos modificadores (§3.5.1).

### Modificador parcial

El modificador partial indica que esta declaración de interfaz (interface-declaration) es una declaración de tipo parcial. Varias declaraciones de interfaz parcial con el mismo nombre dentro de un espacio de nombres envolvente o una declaración de tipo se combinan para formar una declaración de interfaz, siguiendo las reglas especificadas en la sección §10.2.

### Listas de parámetros de tipo variante

Las listas de parámetros de tipo variante solo aparecen en los tipos de interfaz y los tipos de delegado. La diferencia con las listas de parámetros de tipo (type-parameter-list) comunes es la anotación de varianza (variance-annotation) opcional de cada parámetro de tipo.

variant-type-parameter-list:  
< variant-type-parameters >

variant-type-parameters:  
attributesopt variance-annotationopt  type-parameter  
variant-type-parameters , attributesopt variance-annotationopt type-parameter

variance-annotation:  
in  
out

Si la anotación de varianza es out, el parámetro de tipo se dice que es covariante. Si la anotación de varianza es in, el parámetro de tipo se dice que es contravariante. Si no hay anotación de varianza, el parámetro de tipo se dice que es invariante.

En el ejemplo

interface C<out X, in Y, Z>   
{  
 X M(Y y);

Z P { get; set; }  
}

X es covariante, Y es contravariante y Z es invariante.

#### Seguridad de la varianza

La aparición de anotaciones de varianza en la lista de parámetros de tipo de un tipo restringe los lugares en los que los tipos pueden aparecer dentro de la declaración de tipo.

Un tipo T no es seguro para el resultado (output-unsafe) si se cumple alguna de las siguientes condiciones:

* T es un parámetro de tipo contravariante.
* T es un tipo de matriz con un tipo de elemento no seguro para resultado.
* T es un tipo de interfaz o de delegado S<A1,… AK> construido a partir de un tipo genérico S<X1, .. XK> donde por lo menos para un Ai se cumple una de las condiciones siguientes:
* Xi es covariante o invariante y Ai no es seguro para resultado.
* Xi es contravariante o invariante y Ai es seguro para la entrada.

Un tipo T no es seguro para entrada si se cumple alguna de las siguientes condiciones:

* T es un parámetro de tipo covariante
* T es un tipo de matriz con un tipo de elemento no seguro para la entrada.
* T es un tipo de interfaz o de delegado S<A1,… AK> construido a partir de un tipo genérico S<X1, .. XK> donde por lo menos para un Ai se cumple una de las condiciones siguientes:
* Xi es covariante o invariante y Ai no es seguro para la entrada.
* Xi es contravariante o invariante y Ai no es seguro para resultado.

De forma intuitiva, un tipo no seguro para resultado se prohíbe en una posición de resultado y un tipo no seguro para entrada se prohíbe en una posición de entrada.

Un tipo es seguro para el resultado si no es no seguro para el resultado y es seguro para la entrada si no es no seguro para la entrada.

#### Conversión de varianza

El objetivo de las anotaciones de varianza es ofrecer conversiones más indulgentes (aunque con seguridad de tipos) en tipos de interfaces y de delegados. Con este fin, las definiciones de conversión explícitas (§6.1) e implícitas (§6.2) emplean el concepto de convertibilidad de varianza, que se define del modo siguiente:

Si el tipo T<A1, …, An> es de varianza convertible en un tipo T<B1, …, Bn> si T es un tipo de interfaz o de delegado declarado con los parámetros de tipo variante T<X1, …, Xn>, y por cada parámetro de tipo variante Xi es cierta alguna de estas condiciones:

* Xi es covariante y existe una referencia implícita o una conversión de identidad de Ai en Bi
* Xi es contravariante y existe una referencia implícita o una conversión de identidad de Bi en Ai
* Xi es invariante y existe una conversión de identidad de Ai en Bi

### Interfaces base

Una interfaz puede heredar de cero o más tipos de interfaz, que se denominan interfaces base explícitas de la interfaz. Cuando una interfaz tiene una o más interfaces base explícitas, el identificador de interfaz, en la declaración, viene seguido de un carácter de dos puntos y una lista de identificadores de tipos de interfaz base separados por comas.

interface-base:  
: interface-type-list

En un tipo de interfaz construido, las interfaces base explícitas se forman tomando las declaraciones de interfaz base explícita de la declaración de tipo genérico y sustituyendo, para cada parámetro de tipo (type-parameter) de la declaración de interfaz base, el argumento de tipo (type-argument) correspondiente del tipo construido.

Las interfaces base explícitas de una interfaz deben ser por lo menos tan accesibles como la propia interfaz (§3.5.4). Por ejemplo, supone un error en tiempo de compilación especificar una interfaz private o internal en la interfaz base (interface-base) de una interfaz public.

Igualmente, se produce un error en tiempo de compilación cuando una interfaz hereda directa o indirectamente de sí misma.

Las interfaces base de una interfaz son las interfaces base explícitas y sus interfaces base. En otras palabras, el conjunto de interfaces base se compone de la terminación transitiva completa de las interfaces base explícitas, sus interfaces base explícitas, etc. Una interfaz hereda todos los miembros de sus interfaces base. En el ejemplo

interface IControl  
{  
 void Paint();  
}

interface ITextBox: IControl  
{  
 void SetText(string text);  
}

interface IListBox: IControl  
{  
 void SetItems(string[] items);  
}

interface IComboBox: ITextBox, IListBox {}

las interfaces base de IComboBox son IControl, ITextBox e IListBox.

Es decir, la interfaz IComboBox anterior hereda los miembros SetText y SetItems además de Paint.

Toda interfaz base de una interfaz debe ser segura para el resultado (§13.1.3.1). Una clase o struct que implementa una interfaz también implementa implícitamente todas las interfaces base de dicha interfaz.

### Cuerpo de interfaz

El cuerpo de una interfaz (interface-body) define los miembros de la interfaz.

interface-body:  
{ interface-member-declarationsopt }

## Miembros de interfaz

Los miembros de una interfaz son los miembros heredados de las interfaces base y los miembros declarados por la propia interfaz.

interface-member-declarations:  
interface-member-declaration  
interface-member-declarations interface-member-declaration

interface-member-declaration:  
interface-method-declaration  
interface-property-declaration  
interface-event-declaration  
interface-indexer-declaration

Una declaración de interfaz puede declarar cero o varios miembros. Los miembros de una interfaz pueden ser métodos, propiedades, eventos o indizadores. Una interfaz no puede contener constantes, campos, operadores, constructores de instancia, destructores ni tipos, ni tampoco puede una interfaz contener miembros estáticos de ningún tipo.

Todos los miembros de interfaz tienen implícitamente acceso público. Supone un error en tiempo de compilación que las declaraciones de miembros de interfaz contengan modificadores. En particular, los miembros de interfaces no se pueden declarar con los modificadores abstract, public, protected, internal, private, virtual, override o static.

En el ejemplo

public delegate void StringListEvent(IStringList sender);

public interface IStringList  
{  
 void Add(string s);

int Count { get; }

event StringListEvent Changed;

string this[int index] { get; set; }  
}

se declara una interfaz que contiene cada uno de los tipos posibles de miembro: un método, una propiedad, un evento y un indizador.

Una declaración de interfaz (interface-declaration) crea un nuevo espacio de declaración (§3.3), y las declaraciones de miembro de interfaz (interface-member-declarations) contenidas inmediatamente en la declaración de interfaz agregan nuevos miembros al espacio de declaración. Las siguientes reglas se aplican a las declaraciones de miembros de interfaz (interface-member-declaration):

* El nombre de un método debe ser diferente del de cualquier otra propiedad o evento declarados en la misma interfaz. Además, la signatura (§3.6) de un método debe ser distinta de las signaturas de todos los demás métodos declarados en la misma interfaz, y dos métodos declarados en la misma interfaz no pueden tener signaturas que solo se diferencien por ref y out.
* El nombre de una propiedad o evento debe ser diferente de los nombres de todos los demás miembros declarados en la misma interfaz.
* La firma de un indizador debe ser diferente de las firmas de los demás indizadores declarados en la misma interfaz.

Los miembros heredados de una interfaz no forman parte específicamente del espacio de declaración de la interfaz. De esta forma, una interfaz puede declarar un miembro con el mismo nombre o firma que un miembro heredado. Cuando esto ocurre, se dice que el miembro de interfaz derivado oculta el miembro de interfaz base. Ocultar un miembro heredado no se considera un error, pero hace que el compilador emita una advertencia. Para evitar la advertencia, la declaración del miembro de interfaz derivado debe incluir el modificador new para indicar que el miembro derivado está destinado a ocultar el miembro base. Este tema se explica con más detalle en la sección §3.7.1.2.

Si se incluye un modificador new en una declaración que no oculta un miembro heredado, se emite una advertencia a tal efecto. Esta advertencia se evita quitando el modificador new.

Tenga en cuenta que los miembros de la clase object no son, estrictamente hablando, miembros de ninguna interfaz (§13.2). Sin embargo, los miembros de la clase object están disponibles a través de la búsqueda de miembros en cualquier tipo de interfaz (§7.4).

### Métodos de interfaz

Los métodos de interfaz se declaran mediante declaraciones de métodos de interfaz (interface-method-declaration):

interface-method-declaration:  
attributesopt newopt return-type identifier type-parameter-list  
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

Los atributos (attributes), el tipo de valor devuelto (return-type), el identificador (identifier) y la lista de parámetros formales (formal-parameter-list) de una declaración de método de interfaz pueden tener el mismo significado que los de una declaración de método perteneciente a una clase (§10.6). No se permite que una declaración de método de interfaz especifique un cuerpo del método, y la declaración por tanto siempre termina con un punto y coma.

Cada tipo de parámetro formal de un método de interfaz debe ser seguro para la entrada (§13.1.3.1) y el tipo devuelto debe ser void o seguro para el resultado. Además, todas las restricciones de los parámetros de tipo, de los tipos de interfaz y de los tipos de clase de cualquier parámetro de tipo del método deben ser seguras para la entrada.

Estas reglas aseguran que el uso de covariantes o contravariantes mantiene la seguridad de tipos. Por ejemplo,

interface I<out T> { void M<U>() where U : T; }

no es válido porque el uso de T como restricción de parámetro de tipo en U no es seguro para la entrada.

Si esta restricción no estuviera en su sitio, sería posible vulnerar la seguridad de tipos del modo siguiente:

class B {}  
class D : B {}  
class E : B {}  
class C : I<D> { public void M<U>() {…} }  
…  
I<B> b = new C();  
b.M<E>();

Esto es en realidad una llamada a C.M<E>. Pero la llamada exige que E derive de D, de forma que aquí se vulneraría la seguridad de tipos.

### Propiedades de interfaz

Las propiedades de interfaz se declaran mediante declaraciones de propiedad de interfaz (interface-property-declaration):

interface-property-declaration:  
attributesopt newopt type identifier { interface-accessors }

interface-accessors:  
attributesopt get ;  
attributesopt set ;  
attributesopt get ; attributesopt set ;  
attributesopt set ; attributesopt get ;

Los atributos (attributes), el tipo (type) y el identificador (identifier) de una declaración de propiedad de interfaz pueden tener el mismo significado que los de una declaración de propiedad perteneciente a una clase (§10.7).

Los descriptores de acceso correspondientes a una declaración de propiedad de interfaz son los descriptores de acceso de una declaración de propiedad de clase (§10.7.2), salvo que el cuerpo del descriptor de acceso siempre debe ser un punto y coma. Así, el propósito de los descriptores de acceso es indicar si la propiedad es de lectura y escritura, de sólo lectura o de sólo escritura.

El tipo de una propiedad de interfaz debe ser seguro para el resultado si hay un descriptor de acceso get y debe ser seguro para la entrada si hay un descriptor de acceso set.

### Eventos de interfaz

Los eventos de interfaz se declaran mediante declaraciones de evento de interfaz (interface-event-declarations):

interface-event-declaration:  
attributesopt newopt event type identifier ;

Los atributos (attributes), el tipo (type) y el identificador (identifier) de una declaración de evento de interfaz tienen el mismo significado que los de una declaración de evento perteneciente a una clase (§10.8).

El tipo de un evento de interfaz debe ser seguro para el resultado.

### Indizadores de interfaz

Los indizadores de interfaz se declaran mediante declaraciones de indizador de interfaz (interface-indexer-declaration):

interface-indexer-declaration:  
attributesopt newopt type this [ formal-parameter-list ] { interface-accessors }

Los atributos (attributes), el tipo (type) y la lista de parámetros formales (formal-parameter-list) de una declaración de indizador de interfaz tienen el mismo significado que los de una declaración de indizador perteneciente a una clase (§10.9).

Los descriptores de acceso correspondientes a una declaración de indizador de interfaz son los descriptores de acceso de una declaración de indizador de clase (§10.9), salvo que el cuerpo del descriptor de acceso siempre debe ser un punto y coma. Así, el propósito de los descriptores de acceso es indicar si el indizador es de lectura y escritura, de sólo lectura o de sólo escritura.

Todos los tipos de parámetros formales de un indizador de interfaz deben ser seguros para la entrada. Además, cualquier tipo de parámetro formal out o ref también debe ser seguro para el resultado. Observe que incluso los parámetros out deben ser seguros para la entrada debido a una limitación en la plataforma de ejecución subyacente.

El tipo de un indizador de interfaz debe ser seguro para el resultado si hay un descriptor de acceso get y debe ser seguro para la entrada si hay un descriptor de acceso set.

### Acceso a miembros de interfaz

El acceso a los miembros de interfaz tiene lugar mediante las expresiones de acceso a miembros (§7.6.4) y expresiones de acceso a indizadores (§7.6.6.2) de la forma I.M e I[A], donde I es un tipo de interfaz, M es un método, propiedad o evento de dicho tipo de interfaz y A es una lista de argumentos del indizador.

Para interfaces que son estrictamente de herencia simple (todas las interfaces de la cadena de herencia tienen exactamente cero o una sola interfaz base directa), los efectos de las reglas de búsqueda de miembros (§7.4), llamadas a métodos (§7.6.5.1) y acceso a indizadores (§7.6.6.2) son exactamente iguales que las aplicables a clases y structs: más miembros derivados ocultan menos miembros derivados con el mismo nombre o signatura. Sin embargo, para interfaces de herencia múltiple, pueden existir ambigüedades cuando dos o más interfaces base no relacionadas entre sí declaran miembros con el mismo nombre o firma. Esta sección muestra varios ejemplos de tales situaciones. En todos los casos, pueden utilizarse las conversiones implícitas para resolver las ambigüedades.

En el ejemplo

interface IList  
{  
 int Count { get; set; }  
}

interface ICounter  
{  
 void Count(int i);  
}

interface IListCounter: IList, ICounter {}

class C  
{  
 void Test(IListCounter x) {  
 x.Count(1); // Error  
 x.Count = 1; // Error  
 ((IList)x).Count = 1; // Ok, invokes IList.Count.set  
 ((ICounter)x).Count(1); // Ok, invokes ICounter.Count  
 }  
}

las dos primeras instrucciones causan errores en tiempo de compilación, ya que la búsqueda de miembros (§7.4) de Count en IListCounter es ambigua. Como muestra el ejemplo, la ambigüedad se resuelve convirtiendo x en el tipo de interfaz base apropiado. Tales conversiones no tienen costo de ejecución; simplemente consisten en ver la instancia como un tipo menos derivado en tiempo de compilación.

En el ejemplo

interface IInteger  
{  
 void Add(int i);  
}

interface IDouble  
{  
 void Add(double d);  
}

interface INumber: IInteger, IDouble {}

class C  
{  
 void Test(INumber n) {  
 n.Add(1); // Invokes IInteger.Add  
 n.Add(1.0); // Only IDouble.Add is applicable  
 ((IInteger)n).Add(1); // Only IInteger.Add is a candidate  
 ((IDouble)n).Add(1); // Only IDouble.Add is a candidate  
 }  
}

la invocación n.Add(1) selecciona IInteger.Add al aplicar las reglas de resolución de sobrecargas de §7.5.3. De forma similar la invocación n.Add(1.0) selecciona IDouble.Add. Cuando se insertan conversiones explícitas, sólo hay un método candidato y por ello no hay ambigüedad.

En el ejemplo

interface IBase  
{  
 void F(int i);  
}

interface ILeft: IBase  
{  
 new void F(int i);  
}

interface IRight: IBase  
{  
 void G();  
}

interface IDerived: ILeft, IRight {}

class A  
{  
 void Test(IDerived d) {  
 d.F(1); // Invokes ILeft.F  
 ((IBase)d).F(1); // Invokes IBase.F  
 ((ILeft)d).F(1); // Invokes ILeft.F  
 ((IRight)d).F(1); // Invokes IBase.F  
 }  
}

el miembro ILeft.F oculta el miembro IBase.F. La invocación d.F(1), por lo tanto, selecciona ILeft.F, aunque IBase.F no parezca oculto en la ruta de acceso que conduce a IRight.

La regla intuitiva para la ocultación en interfaces de herencia múltiple es sencilla: si un miembros está oculto en cualquier ruta de acceso, queda oculto en todas las rutas de acceso. Debido a que la ruta de acceso desde IDerived que pasa por ILeft hasta IBase oculta IBase.F, el miembro también se oculta en la ruta de acceso de IDerived a IRight y a IBase.

## Nombres completos de miembros de interfaz

A menudo se conoce un miembro de interfaz por su nombre completo. El nombre completo de un miembro de interfaz se compone del nombre de la interfaz en que se declara el miembro, seguido de un punto y del nombre del miembro. El nombre completo de un miembro hace referencia a la interfaz donde se declara. Por ejemplo, dadas las declaraciones

interface IControl  
{  
 void Paint();  
}

interface ITextBox: IControl  
{  
 void SetText(string text);  
}

el nombre completo de Paint es IControl.Paint y el nombre completo de SetText es ITextBox.SetText.

En el ejemplo anterior, no es posible hacer referencia a Paint como ITextBox.Paint.

Cuando una interfaz forma parte de un espacio de nombres, el nombre completo de un miembro de interfaz incluye el espacio de nombres. Por ejemplo:

namespace System  
{  
 public interface ICloneable  
 {  
 object Clone();  
 }  
}

Aquí, el nombre completo del método Clone es System.ICloneable.Clone.

## Implementaciones de interfaces

Las interfaces pueden implementarse mediante clases y structs. Para indicar que una clase o struct implementa directamente una interfaz, el identificador de la interfaz se incluye en la lista de clases base de la clase o el struct. Por ejemplo:

interface ICloneable  
{  
 object Clone();  
}

interface IComparable  
{  
 int CompareTo(object other);  
}

class ListEntry: ICloneable, IComparable  
{  
 public object Clone() {...}

public int CompareTo(object other) {...}  
}

Una clase o un struct que implementa directamente una interfaz también implementa directamente todas las interfaces base de dicha interfaz. Esto es cierto aunque la clase o el struct no muestre explícitamente todas las interfaces base de la lista de clases base. Por ejemplo:

interface IControl  
{  
 void Paint();  
}

interface ITextBox: IControl  
{  
 void SetText(string text);  
}

class TextBox: ITextBox  
{  
 public void Paint() {...}

public void SetText(string text) {...}  
}

Aquí, la clase TextBox implementa IControl e ITextBox.

Cuando una clase C implementa directamente una interfaz, todas las clases derivadas de C también implementan la interfaz implícitamente. Las interfaces base especificadas en una declaración de clase pueden ser tipos de interfaz construidos (§4.4). Una interfaz base no puede ser un parámetro de tipo en sí mismo, aunque puede implicar a parámetros de tipo que se encuentran dentro del ámbito. En el siguiente código se muestra la manera en que una clase puede implementar y ampliar tipos construidos:

class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}

La interfaz base de una declaración de clase genérica debe cumplir la regla de exclusividad descrita en §13.4.2.

### Implementaciones de miembro de interfaz explícitas

En cuestión de implementación de interfaces, una clase o struct puede declarar implementaciones explícitas de miembros de interfaz. Una implementación explícita de miembro de interfaz es una declaración de método, propiedad, evento o indizador que hace referencia a un nombre completo de miembro de interfaz. Por ejemplo:

interface IList<T>  
{  
 T[] GetElements();  
}

interface IDictionary<K,V>  
{  
 V this[K key];

void Add(K key, V value);  
}

class List<T>: IList<T>, IDictionary<int,T>  
{  
 T[] IList<T>.GetElements() {...}

T IDictionary<int,T>.this[int index] {...}

void IDictionary<int,T>.Add(int index, T value) {...}  
}

Aquí IDictionary<int,T>.this e IDictionary<int,T>.Add son implementaciones de miembro de interfaz explícitas.

En algunos casos, el nombre de un miembro de interfaz puede no ser apropiado para la clase que lo implementa, en cuyo caso puede implementarse con la implementación explícita de miembros de interfaz. Es probable que una clase que implemente una abstracción de archivo, por ejemplo, implemente una función miembro Close que tenga el efecto de liberar el recurso de archivo e implemente el método Dispose de la interfaz IDisposable mediante la implementación explícita de miembros de interfaz:

interface IDisposable  
{  
 void Dispose();  
}

class MyFile: IDisposable  
{  
 void IDisposable.Dispose() {  
 Close();  
 }

public void Close() {  
 // Do what's necessary to close the file  
 System.GC.SuppressFinalize(this);  
 }  
}

No es posible obtener acceso a una implementación explícita de miembros de interfaz utilizando el nombre completo en una llamada a un método, instrucción de acceso a propiedad o acceso a indizador. Sólo se puede obtener acceso a una implementación explícita de miembros de interfaz a través de una instancia de interfaz, en cuyo caso se hace referencia a ella por el nombre del miembro.

Si en una implementación explícita de miembros de interfaz se incluyen modificadores de acceso o modificadores abstract, virtual, override o static, se producirán errores en tiempo de compilación.

Las implementaciones explícitas de miembros de interfaz tienen diferentes características de acceso respecto a otros miembros. Dado que este tipo de implementaciones nunca son accesibles por su nombre completo en una llamada a un método o una instrucción de acceso a una propiedad, son en cierto sentido privadas. Sin embargo, debido a que se puede obtener acceso a ellas a través de una instancia de propiedad, también son en otro sentido públicas.

Las implementaciones explícitas de miembros de interfaz tienen dos principales objetivos:

* Puesto que no se puede obtener acceso a ellas por medio de instancias de clase o struct, permiten excluir las implementaciones de interfaz de la interfaz pública de una clase o struct. Esto resulta de especial utilidad cuando una clase o struct implementa una interfaz interna que no es de interés para ningún consumidor de dicha clase o struct.
* Las implementaciones explícitas de miembros de interfaz permiten la eliminación de la ambigüedad en los miembros de interfaz que tienen la misma firma. Sin ellas, una clase o struct no podría contener diferentes implementaciones de los miembros de interfaz de idéntica firma y tipo de valor devuelto; tampoco podría contener ninguna implementación de los miembros de interfaz de igual firma pero distinto tipo de valor devuelto.

Para que una implementación explícita de miembros de interfaz sea válida, la clase o struct debe dar nombre a una interfaz en su clase base que contenga un miembro cuyo nombre completo, tipo y tipos de parámetro coincidan exactamente con los de la implementación explícita de miembros de interfaz. Así, en la clase siguiente

class Shape: ICloneable  
{  
 object ICloneable.Clone() {...}

int IComparable.CompareTo(object other) {...} // invalid  
}

la declaración de IComparable.CompareTo produce un error en tiempo de compilación, porque IComparable no aparece en la lista de clases base de Shape y no es una interfaz base de ICloneable. Igualmente, en las declaraciones

class Shape: ICloneable  
{  
 object ICloneable.Clone() {...}  
}

class Ellipse: Shape  
{  
 object ICloneable.Clone() {...} // invalid  
}

la declaración de ICloneable.Clone en Ellipse supone un error en tiempo de compilación, ya que ICloneable no aparece expresamente citada en la lista de clases base de Ellipse.

El nombre completo de un miembro de interfaz debe hacer referencia a la interfaz donde se declara. Así, en las declaraciones

interface IControl  
{  
 void Paint();  
}

interface ITextBox: IControl  
{  
 void SetText(string text);  
}

class TextBox: ITextBox  
{  
 void IControl.Paint() {...}

void ITextBox.SetText(string text) {...}  
}

la implementación explícita de miembro de interfaz de Paint debe escribirse como IControl.Paint.

### Exclusividad de interfaces implementadas

Las interfaces implementadas por una declaración de tipo genérico deben ser únicas para todos los tipos construidos posibles. Sin esta regla, no se podría determinar el método correcto para llamar a determinados tipos construidos. Por ejemplo, supongamos que se permite escribir una declaración de clase genérica de la siguiente manera:

interface I<T>  
{  
 void F();  
}

class X<U,V>: I<U>, I<V> // Error: I<U> and I<V> conflict  
{  
 void I<U>.F() {...}  
 void I<V>.F() {...}  
}

Si se permitiese, no se podría determinar qué código ejecutar en el siguiente caso:

I<int> x = new X<int,int>();  
x.F();

Para determinar si la lista de interfaces de una declaración de tipo genérico es válida, se llevan a cabo los siguientes pasos:

* Supongamos que L es la lista de interfaces directamente especificada en una declaración de clase, struct o interfaz genérica C.
* Agregue a L cualquier interfaz base de las interfaces que ya se encuentran en L.
* Quite todos los duplicados de L.
* Si después de sustituir los argumentos de tipo en C, los tipos construidos creados desde L provocan que dos interfaces de L sean idénticas, la declaración de C no es válida. Las declaraciones de restricción no se tienen en cuenta a la hora de determinar todos los tipos construidos posibles.

En la declaración de clase X anterior, la lista de interfaces L está formada por I<U> e I<V>. La declaración no es válida porque cualquier tipo construido con U y V del mismo tipo causaría que estas dos interfaces sean tipos idénticos.

Es posible unificar las interfaces especificadas en niveles de herencia diferentes:

interface I<T>  
{  
 void F();  
}

class Base<U>: I<U>  
{  
 void I<U>.F() {…}  
}

class Derived<U,V>: Base<U>, I<V> // Ok  
{  
 void I<V>.F() {…}  
}

Este código es válido aunque Derived<U,V> implemente tanto I<U> como I<V>. El código

I<int> x = new Derived<int,int>();  
x.F();

invoca al método de Derived, ya que Derived<int,int> reimplementa efectivamente I<int> (§13.4.6).

### Implementación de métodos genéricos

Cuando un método genérico implementa de manera implícita un método de interfaz, las restricciones dadas para cada parámetro de tipo del método deben ser equivalentes en ambas declaraciones (después de reemplazar los parámetros de tipo de interfaz por los argumentos de tipo apropiados), donde los parámetros de tipo del método se identifican por sus posiciones ordinales de izquierda a derecha.

Cuando un método genérico implementa explícitamente un método de interfaz, sin embargo, no se permite ninguna restricción en el método implementador. En lugar de eso, se heredan las restricciones del método de interfaz

interface I<A,B,C>  
{  
 void F<T>(T t) where T: A;  
 void G<T>(T t) where T: B;  
 void H<T>(T t) where T: C;  
}

class C: I<object,C,string>  
{  
 public void F<T>(T t) {...} // Ok  
 public void G<T>(T t) where T: C {...} // Ok  
 public void H<T>(T t) where T: string {...} // Error  
}

El método C.F<T> implementa implícitamente I<object,C,string>.F<T>. En este caso, C.F<T> no debe ni puede especificar la restricción T: object ya que object es una restricción implícita en todos los parámetros de tipo. El método C.G<T> implementa de manera implícita I<object,C,string>.G<T> porque las restricciones coinciden con las de la interfaz después de sustituir los parámetros de tipo de interfaz por los argumentos de tipo correspondientes. La restricción para el método C.H<T> es un error porque los tipos sealed (string en este caso) no se pueden utilizar como restricciones. También sería un error omitir la restricción ya que las implementaciones implícitas del método de interfaz deben coincidir. Por lo tanto, es imposible implementar I<object,C,string>.H<T> de manera implícita. Este método de interfaz sólo se puede implementar con una implementación explícita del miembro de interfaz:

class C: I<object,C,string>  
{  
 ...

public void H<U>(U u) where U: class {...}

void I<object,C,string>.H<T>(T t) {  
 string s = t; // Ok  
 H<T>(t);  
 }  
}

En este ejemplo, la implementación explícita del miembro de interfaz invoca un método público que tiene restricciones estrictamente más débiles. Tenga en cuenta que la asignación de t a s es válida puesto que T hereda una restricción de T: string, aunque esta restricción no es expresable en el código fuente.

### Asignación de interfaces

Una clase o struct debe proporcionar implementaciones de todos los miembros de las interfaces enumeradas en su lista de clases base. El proceso de localizar implementaciones de miembros de interfaz en una clase o struct de implementación se conoce como asignación de interfaces.

La asignación de interfaz para una clase o struct C localiza una implementación para cada uno de los miembros de todas las interfaces especificadas en la lista de clases base de C. La implementación de un miembro de interfaz I.M concreto, donde I es la interfaz donde se declara el miembro M, se determina al examinar todas las clases o structs S, comenzando desde C y repitiendo el paso para cada clase base sucesiva de C, hasta encontrar una coincidencia:

* Si S contiene una declaración de una implementación explícita de miembros de interfaz que coincide con I y M, este miembro es la implementación de I.M.
* Por otro lado, si S contiene una declaración de un miembro público no estático que coincide con M, este miembro es la implementación de I.M. Si hay más de un miembro que coincide, no se especifica qué miembro es la implementación de I.M. Esta situación solo ocurre si S es un tipo construido donde los dos miembros, tal como se declara en el tipo genérico, tienen signaturas diferentes, pero los argumentos de tipo hacen que sus signaturas sean idénticas.

Se produce un error en tiempo de compilación si no es posible localizar implementaciones de todos los miembros de cada interfaz especificada en la lista de clases base de C. Observe que los miembros de una interfaz incluyen aquellos miembros heredados de sus interfaces base.

A efectos de la asignación de interfaz, un miembro de clase A coincide con un miembro de interfaz B cuando:

* A y B son métodos, y el nombre, tipo y listas de parámetros formales de A y B son idénticos.
* A y B son propiedades, el nombre y tipo de A y B son idénticos y A posee los mismos descriptores de acceso que B (se permite que A tenga descriptores de acceso adicionales si no es una implementación explícita de miembros de interfaz).
* A y B son eventos, y el nombre y tipo de A y B son idénticos.
* A y B son indizadores, el tipo y listas de parámetros formales de A y B son idénticos y A posee los mismos descriptores de acceso que B (se permite que A tenga descriptores de acceso adicionales si no es una implementación explícita de miembros de interfaz).

Algunas de las principales implicaciones del algoritmo de asignación de interfaz son:

* Las implementaciones explícitas de miembros de interfaz tienen prioridad sobre los demás miembros de una misma clase o struct a la hora de determinar el miembro de clase o struct que implementa a un miembro de interfaz.
* Ni los miembros no públicos ni los no estáticos participan en la asignación de interfaz.

En el ejemplo

interface ICloneable  
{  
 object Clone();  
}

class C: ICloneable  
{  
 object ICloneable.Clone() {...}

public object Clone() {...}  
}

el miembro ICloneable.Clone de C se convierte en la implementación de Clone en ICloneable, porque las implementaciones explícitas de miembros de interfaz tienen prioridad sobre los demás miembros.

Si una clase o struct implementa dos o más interfaces que contienen un miembro con el mismo nombre, tipo y tipos de parámetros, es posible asignar cada uno de los miembros de interfaz a un único miembro de clase o struct. Por ejemplo:

interface IControl  
{  
 void Paint();  
}

interface IForm  
{  
 void Paint();  
}

class Page: IControl, IForm  
{  
 public void Paint() {...}  
}

Aquí, los métodos Paint de IControl e IForm se asignan al método Paint en Page. Naturalmente, también es posible trabajar con diferentes implementaciones explícitas de miembros de interfaz para los dos métodos.

Si una clase o struct implementa una interfaz que contiene miembros ocultos, necesariamente deberán implementarse algunos miembros por medio de la implementación explícita de miembros de interfaz. Por ejemplo:

interface IBase  
{  
 int P { get; }  
}

interface IDerived: IBase  
{  
 new int P();  
}

Una implementación de esta interfaz requeriría al menos una implementación explícita de miembros de interfaz, y tomaría una de las siguientes formas:

class C: IDerived  
{  
 int IBase.P { get {...} }

int IDerived.P() {...}  
}

class C: IDerived  
{  
 public int P { get {...} }

int IDerived.P() {...}  
}

class C: IDerived  
{  
 int IBase.P { get {...} }

public int P() {...}  
}

Cuando una clase implementa varias interfaces con la misma interfaz base, sólo puede haber una implementación de la interfaz base. En el ejemplo

interface IControl  
{  
 void Paint();  
}

interface ITextBox: IControl  
{  
 void SetText(string text);  
}

interface IListBox: IControl  
{  
 void SetItems(string[] items);  
}

class ComboBox: IControl, ITextBox, IListBox  
{  
 void IControl.Paint() {...}

void ITextBox.SetText(string text) {...}

void IListBox.SetItems(string[] items) {...}  
}

no es posible que haya implementaciones independientes para el IControl que figura en la lista de clases base, el IControl heredado por ITextBox y el IControl heredado por IListBox. De hecho, no existe una identidad separada para estas interfaces. En su lugar, las implementaciones de ITextBox e IListBox comparten la misma implementación de IControl, y se considera que ComboBox simplemente implementa tres interfaces: IControl, ITextBox e IListBox.

Los miembros de una clase base participan en la asignación de interfaz. En el ejemplo

interface Interface1  
{  
 void F();  
}

class Class1  
{  
 public void F() {}

public void G() {}  
}

class Class2: Class1, Interface1  
{  
 new public void G() {}  
}

el método F de Class1 se utiliza en la implementación de Class2 en Interface1.

### Herencia de implementación de interfaces

Una clase hereda todas las implementaciones de interfaz proporcionadas por sus clases base.

Sin reimplementar explícitamente una interfaz, una clase derivada no puede alterar de ningún modo las asignaciones de interfaz que hereda de sus clases base. Por ejemplo, en las declaraciones

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 public void Paint() {...}  
}

class TextBox: Control  
{  
 new public void Paint() {...}  
}

el método Paint de TextBox oculta el método Paint de Control, pero no altera la asignación de Control.Paint a IControl.Paint, y las llamadas a Paint a través de instancias de clase y de interfaz tendrán los efectos siguientes

Control c = new Control();  
TextBox t = new TextBox();  
IControl ic = c;  
IControl it = t;  
c.Paint(); // invokes Control.Paint();  
t.Paint(); // invokes TextBox.Paint();  
ic.Paint(); // invokes Control.Paint();  
it.Paint(); // invokes Control.Paint();

Sin embargo, cuando se asigna un método de interfaz a un método virtual de una clase, es posible que las clases derivadas invaliden el método virtual y alteren la implementación de la interfaz. Por ejemplo, al rescribir las declaraciones anteriores de esta manera

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 public virtual void Paint() {...}  
}

class TextBox: Control  
{  
 public override void Paint() {...}  
}

se observarán los siguientes efectos

Control c = new Control();  
TextBox t = new TextBox();  
IControl ic = c;  
IControl it = t;  
c.Paint(); // invokes Control.Paint();  
t.Paint(); // invokes TextBox.Paint();  
ic.Paint(); // invokes Control.Paint();  
it.Paint(); // invokes TextBox.Paint();

Dado que las implementaciones explícitas de miembros de interfaz no pueden declararse como virtuales, no es posible invalidarlas. Sin embargo, es perfectamente válido que una implementación explícita de miembros de interfaz llame a otro método, y se puede declarar dicho método como virtual para permitir que las clases derivadas lo invaliden. Por ejemplo:

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 void IControl.Paint() { PaintControl(); }

protected virtual void PaintControl() {...}  
}

class TextBox: Control  
{  
 protected override void PaintControl() {...}  
}

Aquí, las clases derivadas de Control pueden especializar la implementación de IControl.Paint al invalidar el método PaintControl.

### Reimplementación de interfaces

Una clase que hereda una implementación de interfaz puede reimplementar la interfaz incluyéndola en la lista de clases base.

Una reimplementación de una interfaz sigue exactamente las mismas reglas de asignación de interfaz que la implementación inicial de una interfaz. Así, la asignación de interfaz heredada no tiene efecto alguno en la asignación de interfaz especificada para la reimplementación de la interfaz. Por ejemplo, en las declaraciones

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 void IControl.Paint() {...}  
}

class MyControl: Control, IControl  
{  
 public void Paint() {}  
}

el hecho de que Control asigne IControl.Paint en Control.IControl.Paint no afecta a la reimplementación en MyControl, que asigna IControl.Paint en MyControl.Paint.

Las declaraciones heredadas de miembros públicos y las implementaciones explícitas heredadas de miembros de interfaz participan en el proceso de asignación de interfaz para las interfaces implementadas. Por ejemplo:

interface IMethods  
{  
 void F();  
 void G();  
 void H();  
 void I();  
}

class Base: IMethods  
{  
 void IMethods.F() {}  
 void IMethods.G() {}  
 public void H() {}  
 public void I() {}  
}

class Derived: Base, IMethods  
{  
 public void F() {}  
 void IMethods.H() {}  
}

Aquí, la implementación de IMethods en Derived asigna los métodos de interfaz en Derived.F, Base.IMethods.G, Derived.IMethods.H y Base.I.

Cuando una clase implementa una interfaz, también implementa implícitamente todas las interfaces base de dicha interfaz. De igual forma, una reimplementación de una interfaz también es implícitamente una reimplementación de todas sus interfaces base. Por ejemplo:

interface IBase  
{  
 void F();  
}

interface IDerived: IBase  
{  
 void G();  
}

class C: IDerived  
{  
 void IBase.F() {...}

void IDerived.G() {...}  
}

class D: C, IDerived  
{  
 public void F() {...}

public void G() {...}  
}

Aquí, la reimplementación de IDerived también reimplementa IBase, y asigna IBase.F en D.F.

### Interfaces y clases abstractas

Al igual que una clase no abstracta, una clase abstracta debe proporcionar implementaciones de todos los miembros de las interfaces que figuran en su lista de clases base. Sin embargo, una clase abstracta puede asignar métodos de interfaz a métodos abstractos. Por ejemplo:

interface IMethods  
{  
 void F();  
 void G();  
}

abstract class C: IMethods  
{  
 public abstract void F();  
 public abstract void G();  
}

Aquí, la implementación de IMethods asigna F y G a métodos abstractos, que pueden invalidarse en clases no abstractas que se deriven de C.

Debe observarse que las implementaciones explícitas de miembros de interfaz no pueden ser abstractas, pero sí pueden llamar a métodos abstractos. Por ejemplo:

interface IMethods  
{  
 void F();  
 void G();  
}

abstract class C: IMethods  
{  
 void IMethods.F() { FF(); }

void IMethods.G() { GG(); }

protected abstract void FF();

protected abstract void GG();  
}

Aquí, las clases no abstractas que se derivan de C deberían invalidar FF y GG, proporcionando de esa forma la propia implementación de IMethods.

# Enumeraciones

Un tipo enum es un tipo de valor distintivo (§4.1) que declara un conjunto de constantes con nombre.

El ejemplo

enum Color  
{  
 Red,  
 Green,  
 Blue  
}

declara un tipo enum denominado Color con miembros Red, Green y Blue.

## Declaraciones de enumeración

Una declaración de enumeración declara un nuevo tipo enum. Una declaración de enumeración comienza con la palabra clave enum y define su nombre, su tipo de acceso, su tipo subyacente y sus miembros.

enum-declaration:  
attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt

enum-base:  
: integral-type

enum-body:  
{ enum-member-declarationsopt }  
{ enum-member-declarations , }

Cada tipo enum tiene un tipo entero correspondiente denominado tipo subyacente del tipo enum. Este tipo subyacente debe poder representar todos los valores de enumerador definidos en la enumeración. Una declaración de enumeración puede declarar explícitamente un tipo subyacente de byte, sbyte, short, ushort, int, uint, long o ulong. Observe que no se puede utilizar char como tipo subyacente. Una declaración de enumeración que no declara explícitamente un tipo subyacente tiene int como tipo subyacente.

El ejemplo

enum Color: long  
{  
 Red,  
 Green,  
 Blue  
}

declara una enumeración con el tipo subyacente long. Un programador podría utilizar un tipo subyacente long, como en el ejemplo, para habilitar el uso de valores que estén en el intervalo de long, pero no en el de int, o para preservar esta opción para el futuro.

## Modificadores de enumeración

Una declaración de enumeración (enum-declaration) puede incluir opcionalmente una secuencia de modificadores de enumeración:

enum-modifiers:  
enum-modifier  
enum-modifiers enum-modifier

enum-modifier:  
new  
public  
protected  
internal  
private

Cuando el mismo modificador aparece varias veces en una declaración de enumeración, se produce un error en tiempo de compilación.

Los modificadores de una declaración de enumeración tienen el mismo significado que los de las declaraciones de clases (§10.1.1). Observe, sin embargo, que los modificadores abstract y sealed no están permitidos en una declaración de enumeración. Las enumeraciones no pueden ser abstractas y no permiten derivación.

## Miembros de enumeración

El cuerpo de una declaración de tipo enum define cero o varios miembros de enumeración, que son las constantes con nombre del tipo enum. No puede haber dos miembros de enumeración con el mismo nombre.

enum-member-declarations:  
enum-member-declaration  
enum-member-declarations , enum-member-declaration

enum-member-declaration:  
attributesopt identifier  
attributesopt identifier = constant-expression

Cada miembro de enumeración tiene un valor asociado constante. El tipo de este valor es el tipo subyacente de la enumeración contenedora. El valor constante de cada miembro de enumeración debe estar comprendido en el intervalo del tipo subyacente de la enumeración. El ejemplo

enum Color: uint  
{  
 Red = -1,  
 Green = -2,  
 Blue = -3  
}

produce un error en tiempo de compilación, porque los valores constantes -1, -2 y –3 no están en el intervalo del tipo entero subyacente uint.

Varios miembros de enumeración pueden compartir el mismo valor asociado. En el ejemplo

enum Color   
{  
 Red,  
 Green,  
 Blue,

Max = Blue  
}

se muestra una enumeración en la que dos de sus miembros (Blue y Max) tienen el mismo valor asociado.

El valor asociado de un miembro de enumeración se asigna bien implícita o explícitamente. Si la declaración del miembro de enumeración tiene un inicializador de expresión constante (constant-expression), el valor de dicha expresión, convertido de manera implícita al tipo subyacente de la enumeración, es el valor asociado del miembro de enumeración. Si la declaración del miembro de enumeración no tiene inicializador, su valor asociado se establece de forma implícita, de la siguiente forma:

* Si el miembro de enumeración es el primero de los declarados en el tipo enum, su valor asociado es cero.
* En cualquier otro caso, el valor asociado del miembro de enumeración se obtiene al aumentar en uno el valor asociado del miembro de enumeración precedente en el código. Este valor aumentado debe estar incluido en el intervalo de valores que puede representar el tipo subyacente, en caso contrario, se produce un error de compilación.

En el ejemplo

using System;

enum Color  
{  
 Red,  
 Green = 10,  
 Blue  
}

class Test  
{  
 static void Main() {  
 Console.WriteLine(StringFromColor(Color.Red));  
 Console.WriteLine(StringFromColor(Color.Green));  
 Console.WriteLine(StringFromColor(Color.Blue));  
 }

static string StringFromColor(Color c) {  
 switch (c) {  
 case Color.Red:   
 return String.Format("Red = {0}", (int) c);

case Color.Green:  
 return String.Format("Green = {0}", (int) c);

case Color.Blue:  
 return String.Format("Blue = {0}", (int) c);

default:  
 return "Invalid color";  
 }  
 }  
}

se imprimen los nombres de los miembros de enumeración y sus valores asociados. El resultado es:

Red = 0  
Green = 10  
Blue = 11

por las siguientes razones:

* al miembro de enumeración Red se le asigna automáticamente el valor cero (puesto que no tiene inicializador y es el primer miembro de enumeración);
* al miembro de enumeración Green se le asigna explícitamente el valor 10;
* y al miembro de enumeración Blue se le asigna automáticamente el valor del miembro que le precede en el código más uno.

El valor asociado de un miembro de enumeración no puede, directa ni indirectamente, usar el valor de su propio miembro de enumeración asociado. Aparte de esta restricción de circularidad, los inicializadores de miembros de enumeración pueden hacer referencia con libertad a otros inicializadores de miembros de enumeración, sin importar su posición en el código. Dentro de un inicializador de miembro de enumeración, los valores de los demás miembros de enumeración siempre se tratan como si tuvieran el tipo de su tipo subyacente, de manera que no son necesarias las conversiones de tipos al hacer referencia a otros miembros de enumeración.

En el ejemplo

enum Circular  
{  
 A = B,  
 B  
}

se produce un error en tiempo de compilación, porque las declaraciones de A y B son circulares. Es decir, A depende de B explícitamente y B depende de A implícitamente.

Los miembros de enumeración se denominan y se establece su ámbito de manera exactamente análoga a los campos pertenecientes a clases. El ámbito de un miembro de enumeración es el cuerpo de su tipo enum contenedor. Dentro de ese ámbito, se puede hacer referencia a los miembros de enumeración por su nombre simple. Desde cualquier otra ubicación del código, debe calificarse el nombre de un miembro de enumeración con el nombre de su tipo enum. Los miembros de enumeración no tienen ninguna accesibilidad declarada: un miembro de enumeración es accesible si el tipo enum que lo contiene es accesible.

## Tipo System.Enum

El tipo System.Enum es la clase base abstracta de todos los tipos enum (es distinta y diferente del tipo subyacente del tipo enum), y los miembros heredados de System.Enum están disponibles en cualquier tipo enum. Existe una conversión boxing (§4.3.1) de cada tipo enum a System.Enum y también existe una conversión unboxing (§4.3.2) de System.Enum a cualquier tipo enum.

Observe que System.Enum no es en sí mismo un tipo enum (enum-type). Más bien es un tipo de clase (class-type) del que se derivan todos los tipos enum (enum-type). El tipo System.Enum hereda del tipo System.ValueType (§4.1.1), que, a su vez, hereda del tipo object. En tiempo de ejecución, un valor de tipo System.Enum puede ser null o una referencia a un valor con conversión boxing de cualquier tipo enum.

## Valores y operaciones de enumeración

Cada tipo enum define un tipo distintivo; es necesaria una conversión explícita de enumeración (§6.2.2) para convertir un tipo enum a un tipo entero, o para convertir dos tipos enum. El conjunto de valores que puede tomar un tipo enum no está limitado por sus miembros de enumeración. En concreto, cualquier valor del tipo subyacente de una enumeración puede convertirse en el tipo enum, y es un valor aceptado y distintivo de dicho tipo enum.

Los miembros de enumeración tienen el tipo de su tipo enum contenedor (salvo dentro de otros inicializadores de miembros de enumeración: vea la sección §14.3). El valor de un miembro de enumeración declarado en el tipo enum E con un valor v asociado es (E)v.

Los operadores siguientes se pueden utilizar en los valores de tipos enum: ==, !=, <, >, <=, >= (§7.10.5), + binario (§7.8.4), ‑ binario (§7.8.5), ^, &, | (§7.11.2), ~ (§7.7.4), ++ y -- (§7.6.9 y §7.7.5).

Cada tipo enum se deriva automáticamente de la clase System.Enum (que, a su vez, se deriva de System.ValueType y object) \t "See also Enum" \t "See also Enum" \t "See also Enum" \t "See Enum" \t "See also Enum" \t "See Enum" . Así, los métodos heredados y propiedades de esta clase se pueden utilizar en los valores de un tipo de enumeración.

# Delegados

Los delegados habilitan escenarios de programación que otros lenguajes, como C++, Pascal y Modula, han resuelto mediante punteros a función. A diferencia de los punteros a función de C++, los delegados están completamente orientados a objetos y, a diferencia de los punteros a funciones miembro de C++, los delegados encapsulan a la vez una instancia de objeto y un método.

Una declaración de delegado define una clase que se deriva de la clase System.Delegate. Una instancia de delegado encapsula una lista de llamadas, que consiste en una lista de uno o varios métodos, a cada uno de los que se hace referencia como una entidad invocable. Para métodos de instancia, una entidad invocable consta de una instancia y un método de la instancia. Para los métodos estáticos, una entidad invocable está formada solamente por un método. Invocar a una instancia delegada con un conjunto adecuado de argumentos, hace que se invoque a cada una de las entidades del delegado invocables, con el conjunto de argumentos dado.

Una propiedad interesante y útil de una instancia de delegado es que no necesita conocer las clases de los métodos a los que encapsula; lo único que importa es que los métodos sean compatibles (§15.1) con el tipo del delegado. Esto hace que los delegados sean perfectos para una invocación “anónima”.

## Declaraciones de delegados

Una declaración de delegado (delegate-declaration) es una declaración de tipo (type-declaration) (§9.6) que declara un nuevo tipo delegado.

delegate-declaration:  
attributesopt delegate-modifiersopt delegate return-type   
 identifier variant-type-parameter-listopt   
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

delegate-modifiers:  
delegate-modifier  
delegate-modifiers delegate-modifier

delegate-modifier:  
new  
public  
protected  
internal  
private

Cuando el mismo modificador aparece varias veces en una declaración de delegado, se produce un error en tiempo de compilación.

El modificador new solo está permitido en delegados declarados dentro de otro tipo, en cuyo caso, éste especifica que tal delegado oculta un miembro heredado con el mismo nombre, como se describe en la sección §10.3.4.

Los modificadores public, protected, internal y private controlan la accesibilidad del tipo delegado. Dependiendo del contexto en el que ocurra la declaración de delegado, algunos de estos modificadores pueden no estar permitidos (§3.5.1).

El nombre del tipo delegado es el identificador (identifier).

La lista opcional de parámetros formales (formal-parameter-list) especifica los parámetros del delegado, y el tipo de valor devuelto (return-type) corresponde al delegado.

La lista opcional de parámetros de tipo variante (variant-type-parameter-list) (§13.1.3) especifica los parámetros de tipo que corresponden al propio delegado.

El tipo de valor devuelto de una firma del método debe ser void o seguro para resultado (§13.1.3.1).

Todos los tipos de parámetros formales de un tipo de delegado deben ser seguros para la entrada. Además, cualquier tipo de parámetro out o ref también debe ser seguro para el resultado. Observe que incluso los parámetros out deben ser seguros para la entrada debido a una limitación en la plataforma de ejecución subyacente.

Los tipos delegados en C# son equivalentes en su denominación, no estructuralmente. En concreto, dos tipos delegados diferentes con la misma lista de parámetros y el mismo tipo de valor devuelto se consideran diferentes. Sin embargo, las instancias de dos tipos delegados distintos pero estructuralmente equivalentes se pueden considerar iguales (§7.9.8).

Por ejemplo:

delegate int D1(int i, double d);

class A  
{  
 public static int M1(int a, double b) {...}  
}

class B  
{  
 delegate int D2(int c, double d);

public static int M1(int f, double g) {...}

public static void M2(int k, double l) {...}

public static int M3(int g) {...}

public static void M4(int g) {...}  
}

Los métodos A.M1 y B.M1 son compatibles con los tipos delegados D1 y D2, ya que poseen el mismo tipo de valor devuelto y la misma lista de parámetros; sin embargo, son tipos diferentes, por lo cual no son intercambiables entre sí. Los métodos B.M2, B.M3 y B.M4 son incompatibles con los tipos delegados D1 y D2, ya que poseen diferentes tipos de valores devueltos o diferentes listas de parámetros.

Al igual que otras declaraciones de tipo genérico, se deben proporcionar argumentos de tipo para crear un tipo de delegado construido. Los tipos de parámetro y el tipo de valor devuelto de un tipo de delegado construido se crean sustituyendo, para cada parámetro de tipo de la declaración de delegado, el argumento de tipo correspondiente del tipo de delegado construido. El tipo de valor devuelto resultante y los tipos de parámetros se utilizan para determinar los métodos compatibles con un tipo de delegado construido. Por ejemplo:

delegate bool Predicate<T>(T value);

class X  
{  
 static bool F(int i) {...}

static bool G(string s) {...}  
}

El método X.F es compatible con el tipo delegado Predicate<int> y el método X.G es compatible con el tipo delegado Predicate<string>.

La única forma de declarar un tipo delegado es a través de una declaración de delegado (delegate-declaration). Un tipo de delegado es un tipo de clase derivada de System.Delegate. Los tipos de delegados son sealed de manera implícita, por lo que no está permitido derivar ningún tipo de un tipo de delegado. Tampoco se permite derivar un tipo de clase no delegado a partir de System.Delegate. Observe que System.Delegate no es en sí mismo un tipo de delegado; es un tipo de clase del cual se derivan todos los tipos de delegados.

C# proporciona una sintaxis especial para la creación y llamada de instancias de delegado. Salvo para la creación de instancias, cualquier operación que pueda aplicarse a una clase o instancia de clase también puede aplicarse a una clase o instancia de delegado, respectivamente. En particular, es posible obtener acceso a los miembros del tipo System.Delegate a través de la sintaxis habitual de acceso a miembros.

El conjunto de métodos encapsulados por una instancia de delegado se denomina lista de llamadas. Cuando se crea una instancia de delegado (§15.2) a partir de un único método, encapsula el método y su lista de llamadas contiene una sola entrada. Sin embargo, cuando se combinan dos instancias de delegado no null, sus listas de llamadas se concatenan, por orden de operando izquierdo a operando derecho, para formar una nueva lista de llamadas, que contiene dos o más entradas.

Los delegados se combinan utilizando los operadores binarios + (§7.8.4) y += (§7.17.2). Se puede quitar un delegado de una combinación de delegados utilizando los operadores binarios - (§7.8.5) y -= (§7.17.2). Los delegados se pueden comparar para comprobar si hay igualdad (§7.10.8).

El ejemplo siguiente muestra la creación de instancias de un número de delegados y sus correspondientes listas de llamadas:

delegate void D(int x);

class C  
{  
 public static void M1(int i) {...}

public static void M2(int i) {...}

}

class Test  
{  
 static void Main() {  
 D cd1 = new D(C.M1); // M1  
 D cd2 = new D(C.M2); // M2  
 D cd3 = cd1 + cd2; // M1 + M2  
 D cd4 = cd3 + cd1; // M1 + M2 + M1  
 D cd5 = cd4 + cd3; // M1 + M2 + M1 + M1 + M2  
 }

}

Al crear instancias de cd1 y cd2, cada una de ellas encapsula un método. Cuando se crea una instancia de cd3, tiene una lista de llamadas de dos métodos, M1 y M2, por ese orden. La lista de llamadas de cd4 contiene M1, M2 y M1, por ese orden. Por último, la lista de llamadas de cd5 contiene M1, M2, M1, M1 y M2, por ese orden. Para ver más ejemplos de combinación (y supresión) de delegados, vea la sección §15.4.

## Compatibilidad de delegados

Un método o delegado M es compatible con un tipo delegado D si se cumplen las siguientes condiciones:

* D y M tienen el mismo número de parámetros y cada parámetro de D tiene los mismos modificadores ref u out que el parámetro correspondiente en M.
* Para cada parámetro de valor (un parámetro sin un modificador ref u out), existe una conversión de identidad (§6.1.1) o una conversión de referencia implícita (§6.1.6) desde el tipo de parámetro D al tipo de parámetro correspondiente en M.
* Para cada parámetro ref u out, el tipo de parámetro en D es el mismo que el tipo de parámetro en M.
* Existe una conversión de identidad o de referencia implícita desde el tipo de valor devuelto de M al tipo de valor devuelto de D.

## Creación de instancias de delegados

Una instancia de un delegado se crea mediante una expresión de creación de delegados (delegate-creation-expression) (§7.6.10.5) o una conversión a un tipo delegado. La instancia de delegado recién creada hace referencia a alguno de los siguientes:

* El método estático a que se hace referencia en la expresión de creación de delegados (delegate-creation-expression), o bien
* El objeto de destino (que no puede ser null) y el método de instancia a que se hace referencia en la expresión de creación de delegados (delegate-creation-expression), o bien
* Otro delegado.

Por ejemplo:

delegate void D(int x);

class C  
{  
 public static void M1(int i) {...}  
 public void M2(int i) {...}  
}

class Test  
{  
 static void Main() {   
 D cd1 = new D(C.M1); // static method  
 C t = new C();  
 D cd2 = new D(t.M2); // instance method  
 D cd3 = new D(cd2); // another delegate  
 }  
}

Una vez creadas las instancias de delegado, éstas siempre hacen referencia al mismo objeto de destino y método. Recuerde que, cuando se combinan dos delegados, o se quita uno de ellos en el otro, el resultado es un nuevo delegado con su propia lista de llamadas; las listas de llamadas de los delegados combinados o eliminados permanecen sin cambios.

## Invocación de delegados

C# proporciona una sintaxis especial para invocar un delegado. Cuando se invoca una instancia de delegado no null cuya lista de llamadas contiene un sola entrada, llama a ese método con los mismos argumentos que recibió, y devuelve el mismo valor que el método a que se hace referencia. (Vea la sección §7.6.5.3 para obtener información detallada sobre la invocación de delegados.) Si se produce una excepción durante la llamada a un delegado y no se captura dentro del método invocado, la búsqueda de una cláusula de excepción catch continúa en el método que llamó al delegado, como si ese método hubiera llamado directamente al método al que hacía referencia el delegado.

La invocación de una instancia de delegado cuya lista de llamadas contiene varias entradas continúa llamando a cada uno de los métodos de la lista, de forma sincrónica, por orden. A cada uno de los métodos así llamados se le pasa el mismo conjunto de argumentos pasados a la instancia de delegado. Si tal invocación de delegado incluye parámetros de referencia (§10.6.1.2), cada invocación de método incluirá una referencia a la misma variable; los cambios en dicha variable que realice un método de la lista de llamadas serán visibles para todos los métodos situados a continuación en la lista. Si la invocación de delegado incluye parámetros de salida o un valor devuelto, su valor final vendrá dado por la invocación del último delegado de la lista.

Si se produce una excepción durante la llamada a dicho delegado y no se captura dentro del método invocado, la búsqueda de una cláusula de excepción catch continúa en el método que llamó al delegado, y no se invoca ninguno de los métodos situados a continuación en la lista de llamadas.

El intento de llamar a una instancia de delegado cuyo valor es NULL, tiene como resultado una excepción de tipo System.NullReferenceException.

En el siguiente ejemplo se muestra cómo crear instancias, combinar, quitar e invocar delegados:

using System;

delegate void D(int x);

class C  
{  
 public static void M1(int i) {  
 Console.WriteLine("C.M1: " + i);  
 }

public static void M2(int i) {  
 Console.WriteLine("C.M2: " + i);  
 }

public void M3(int i) {  
 Console.WriteLine("C.M3: " + i);  
 }  
}

class Test  
{  
 static void Main() {   
 D cd1 = new D(C.M1);  
 cd1(-1); // call M1

D cd2 = new D(C.M2);  
 cd2(-2); // call M2

D cd3 = cd1 + cd2;  
 cd3(10); // call M1 then M2

cd3 += cd1;  
 cd3(20); // call M1, M2, then M1

C c = new C();  
 D cd4 = new D(c.M3);  
 cd3 += cd4;  
 cd3(30); // call M1, M2, M1, then M3

cd3 -= cd1; // remove last M1  
 cd3(40); // call M1, M2, then M3

cd3 -= cd4;  
 cd3(50); // call M1 then M2

cd3 -= cd2;  
 cd3(60); // call M1

cd3 -= cd2; // impossible removal is benign  
 cd3(60); // call M1

cd3 -= cd1; // invocation list is empty so cd3 is null

// cd3(70); // System.NullReferenceException thrown

cd3 -= cd1; // impossible removal is benign  
 }  
}

Tal como se muestra en la instrucción cd3 += cd1;, un delegado puede aparecer varias veces en una lista de llamadas. En este caso, se le llama una vez por cada aparición. En una lista de llamadas como esta, cuando se quita el delegado, la última aparición en la lista es la que se elimina realmente.

Inmediatamente antes de la ejecución de la instrucción final, cd3 -= cd1; el delegado cd3 hace referencia a una lista de llamadas vacía. Intentar quitar un delegado de una lista vacía (o quitar un delegado no existente de una lista que no está vacía) no es ningún error.

El resultado producido es el siguiente:

C.M1: -1  
C.M2: -2  
C.M1: 10  
C.M2: 10  
C.M1: 20  
C.M2: 20  
C.M1: 20  
C.M1: 30  
C.M2: 30  
C.M1: 30  
C.M3: 30  
C.M1: 40  
C.M2: 40  
C.M3: 40  
C.M1: 50  
C.M2: 50  
C.M1: 60  
C.M1: 60

# Excepciones

Las excepciones de C# proporcionan una forma estructurada, uniforme y con seguridad de tipos de controlar situaciones de error del sistema y de las aplicaciones. El mecanismo de excepciones de C# es bastante similar al de C++, pero con unas cuantas diferencias importantes:

* En C#, todas las excepciones deben estar representadas por una instancia de un tipo de clase derivado de System.Exception. En C++, cualquier valor de cualquier tipo puede utilizarse para representar una excepción.
* En C#, puede utilizarse un bloque finally (§8.10) para crear código de terminación que se ejecute en condiciones normales y excepcionales. Ese tipo de código es difícil de escribir en C++ sin duplicar código.
* En C#, las excepciones del nivel de sistema, como desbordamiento, división por cero o desreferenciaciones nulas, tienen clases de excepción bien definidas y están al mismo nivel que los escenarios de error de aplicaciones.

## Causas de excepciones

Una excepción se puede iniciar de dos formas diferentes.

* Una instrucción throw (§8.9.5) inicia una excepción de manera inmediata e incondicional. El control nunca llega a la instrucción que sigue inmediatamente a throw.
* Algunas situaciones excepcionales que pueden surgir durante el procesamiento de instrucciones y expresiones de C# provocan una excepción en ciertas circunstancias cuando no es posible completar normalmente una operación. Por ejemplo, una operación de división de enteros (§7.8.2) inicia una excepción System.DivideByZeroException si el denominador es cero. Vea la sección §16.4 para obtener una lista de las diferentes excepciones que pueden producirse de esta forma.

## Clase System.Exception

La clase System.Exception es el tipo base de todas las excepciones. Esta clase tiene unas cuantas propiedades a resaltar que comparten todas las excepciones:

* Message es una propiedad de solo lectura de tipo string que contiene una descripción legible por el hombre de la causa que provocó la excepción.
* InnerException es una propiedad de solo lectura de tipo Exception. Si su valor no es NULL, se refiere a la excepción que causó la excepción actual, es decir, que la excepción actual se inició en un bloque catch que controlaba la excepción InnerException. En cualquier otro caso, su valor es null, lo que indica que esta excepción no fue causada por otra excepción. El número de objetos de excepción encadenados de esta forma puede ser arbitrario.

El valor de estas propiedades puede especificarse en las llamadas al constructor de instancia de System.Exception.

## Cómo controlar excepciones

Las excepciones se controlan mediante una instrucción try (§8.10).

Cuando se produce una excepción, el sistema busca la cláusula catch más próxima que controla la excepción, según lo determina el tipo de la excepción en tiempo de ejecución. En primer lugar, se busca una instrucción try contenedora en el método actual, y se consideran las cláusulas catch asociadas de la instrucción try según su ordenación. Si ese primer paso no se produce, se busca el método que llamó al método actual en una instrucción try que contenga el punto de la llamada al método actual. Esta búsqueda continúa hasta encontrar una cláusula catch que puede controlar la excepción actual, denominando una clase de excepción que sea de la misma clase o clase base del tipo en tiempo de ejecución de la excepción iniciada. Una cláusula catch que no denomina ninguna clase de excepción puede controlar cualquier excepción.

Una vez hallada una cláusula catch coincidente, el sistema se prepara para transferir el control a la primera instrucción de la cláusula catch. Antes de comenzar la ejecución de la cláusula catch, el sistema ejecuta, por orden, todas las cláusulas finally asociadas a instrucciones try cuyo nivel de anidamiento sea mayor que el de la que capturó la excepción.

De no encontrarse ninguna cláusula catch coincidente, se produce una de estas dos circunstancias:

* Si la búsqueda de una cláusula catch coincidente llega hasta un constructor estático (§10.12) o un inicializador de campo estático, se inicia una excepción System.TypeInitializationException en el punto donde se desencadenó la llamada al constructor estático. La excepción interna de System.TypeInitializationException contiene la excepción iniciada en un principio.
* Si la búsqueda de cláusulas catch coincidentes llega hasta el código que inició el subproceso, termina la ejecución del mismo. El impacto de esta terminación se define según la implementación.

Las excepciones que se producen durante la ejecución de un destructor merecen especial atención. Si se produce una excepción durante la ejecución de un destructor y la excepción no es capturada, se termina la ejecución de dicho destructor y se llama al destructor de la clase base (si existe). Si no hay clase base (como ocurre con el tipo object) o no hay un destructor de clase base, se descarta la excepción.

## Clases de excepción comunes

A continuación se muestran las excepciones iniciadas por algunas operaciones de C#.

|  |  |
| --- | --- |
| System.ArithmeticException | Una clase base para las excepciones que se producen durante las operaciones aritméticas, como System.DivideByZeroException y System.OverflowException. |
| System.ArrayTypeMismatchException | Se inicia cuando una operación de almacenamiento en una matriz da error porque el tipo real del elemento almacenado es incompatible con el tipo de la matriz. |
| System.DivideByZeroException | Se inicia cuando se produce un intento de dividir un valor integral por cero. |
| System.IndexOutOfRangeException | Se inicia cuando se produce un intento de indizar una matriz por medio de un índice menor que cero o situado fuera de los límites de la matriz. |
| System.InvalidCastException | Se inicia cuando una conversión explícita de un tipo o interfaz base a un tipo derivado da un error en tiempo de ejecución. |
| System.NullReferenceException | Se inicia cuando se utiliza una referencia null de manera que hace obligatorio el objeto al que se hace referencia. |
| System.OutOfMemoryException | Se inicia cuando hay problemas al intentar asignar memoria a través de new. |
| System.OverflowException | Se inicia cuando una operación aritmética en un contexto checked produce un desbordamiento. |
| System.StackOverflowException | Se inicia cuando se agota la pila de excepciones debido a la existencia de demasiadas llamadas de método pendientes; suele indicar un nivel de recursividad muy profundo o ilimitado. |
| System.TypeInitializationException | Se inicia cuando un constructor estático inicia una excepción sin que haya cláusulas catch para capturarla. |

# Atributos

Gran parte del lenguaje C# permite al programador especificar información declarativa acerca de las entidades que se definen en el programa. Por ejemplo, la accesibilidad de un método en una clase se especifica al decorarlo con los modificadores de método (method-modifiers) public, protected, internal y private.

C# permite a los programadores inventar nuevas clases de información declarativa, denominadas atributos. Los programadores pueden entonces asociar atributos a varias entidades de programa y recuperar la información de atributos en un entorno en tiempo de ejecución. Por ejemplo, un marco de trabajo podría definir un atributo HelpAttribute que se pueda colocar en ciertos elementos del programa, como clases y métodos, para proporcionar una asignación o correspondencia entre dichos elementos y su documentación.

Los atributos se definen mediante la declaración de clases de atributos (§17.1), que pueden contener parámetros posicionales y con nombre (§17.1.2). Los atributos se asocian a entidades de un programa de C# mediante las especificaciones de atributos (§17.2), y pueden recuperarse en tiempo de ejecución como instancias de atributo (§17.3).

## Clases de atributos

Una clase que se deriva de la clase abstracta System.Attribute, ya sea directa o indirectamente es una clase de atributos. La declaración de una clase de atributos define un nuevo tipo de atributo que se puede insertar en una declaración. Por convención, las clases de atributos se denominan con el sufijo Attribute. Los usos de un atributo pueden incluir u omitir este sufijo.

### Uso de los atributos

El atributo AttributeUsage (§17.4.1) se utiliza para describir cómo se puede utilizar una clase de atributos.

AttributeUsage posee un parámetro posicional (§17.1.2) que habilita una clase de atributos para especificar los tipos de declaraciones en los que se puede usar. El ejemplo

using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]  
public class SimpleAttribute: Attribute   
{  
 ...  
}

define una clase de atributos denominada SimpleAttribute que solo se puede colocar en declaraciones de clase (class-declaration) y en declaraciones de interfaz (interface-declaration). El ejemplo

[Simple] class Class1 {...}

[Simple] interface Interface1 {...}

muestra varios usos del atributo Simple. Si bien el atributo se define con el nombre SimpleAttribute, cuando se utiliza puede omitirse el sufijo Attribute, dando como resultado el nombre corto Simple. Así, el ejemplo anterior es semánticamente equivalente al que se muestra a continuación:

[SimpleAttribute] class Class1 {...}

[SimpleAttribute] interface Interface1 {...}

AttributeUsage tiene un parámetro con nombre (§17.1.2) denominado AllowMultiple, que indica si el atributo puede especificarse más de una vez para una entidad dada. Si AllowMultiple para una clase de atributos es true, se trata de una clase de atributo de uso múltiple y se puede especificar más de una vez en una entidad. Si AllowMultiple para una clase de atributos es false o no está especificado, se trata de una clase de atributos de uso único y se puede especificar como mucho una vez en una entidad.

El ejemplo

using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]  
public class AuthorAttribute: Attribute  
{  
 private string name;

public AuthorAttribute(string name) {  
 this.name = name;  
 }

public string Name {  
 get { return name; }  
 }  
}

define una clase de atributos de uso múltiple de nombre AuthorAttribute. El ejemplo

[Author("Brian Kernighan"), Author("Dennis Ritchie")]   
class Class1  
{  
 ...  
}

muestra una declaración de clase con dos usos del atributo Author.

AttributeUsage tiene otro parámetro con nombre denominado Inherited, que especifica si el atributo, cuando se especifica en una clase base, también es heredado por las clases que se derivan de esta clase base. Si Inherited para una clase de atributos es true, se hereda dicho atributo. Si Inherited para una clase de atributos es false, dicho atributo no se hereda. Si no se especifica, su valor predeterminado es true.

Una clase de atributos X sin atributo AttributeUsage asociado, como en

using System;

class X: Attribute {...}

equivale a lo siguiente:

using System;

[AttributeUsage(  
 AttributeTargets.All,  
 AllowMultiple = false,  
 Inherited = true)  
]  
class X: Attribute {...}

### Parámetros posicionales y con nombre

Las clases de atributos pueden tener parámetros posicionales y parámetros con nombre. Cada constructor de instancia público de una clase de atributos define una secuencia válida de parámetros posicionales para esa clase de atributo. Cada campo o propiedad de lectura y escritura pública y no estática de una clase de atributos define un parámetro con nombre para la clase de atributo.

El ejemplo

using System;

[AttributeUsage(AttributeTargets.Class)]  
public class HelpAttribute: Attribute  
{  
 public HelpAttribute(string url) { // Positional parameter  
 ...  
 }

public string Topic { // Named parameter  
 get {...}  
 set {...}  
 }

public string Url {  
 get {...}  
 }  
}

define una clase de atributos denominada HelpAttribute que posee un parámetro posicional, url, y un parámetro con nombre, Topic. Si bien es no estática y pública, la propiedad Url no define un parámetro con nombre porque no es de lectura y escritura.

Esta clase de atributos podría utilizarse de la manera siguiente:

[Help("http://www.mycompany.com/.../Class1.htm")]  
class Class1  
{  
 ...  
}

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]  
class Class2  
{  
 ...  
}

### Tipos de parámetros de atributos

Los tipos de los parámetros posicionales y con nombre de una clase de atributos se limitan a los tipos de parámetros de atributos, que son:

* Uno de los tipos siguientes: bool, byte, char, double, float, int, long, sbyte, short, string, uint, ulong, ushort.
* El tipo object.
* El tipo System.Type.
* Un tipo enum, con la condición de que este y los tipos en los que esté anidado (si los hubiera) tengan acceso público (§17.2).
* Matrices unidimensionales de los tipos anteriores.

Un argumento de constructor o un campo público que no tenga uno de estos tipos no se puede usar como parámetro con nombre o posicional en una especificación de atributo.

## Especificación de atributos

La especificación de atributos es la aplicación de un atributo previamente definido a una declaración. Un atributo es una parte de información declarativa adicional que se especifica en una declaración. Los atributos pueden especificarse en un ámbito global (para especificar atributos en el ensamblado o módulo que los contienen) y para declaraciones de tipo (type-declaration) (§9.6), declaraciones de miembros de clase (class-member-declaration) (§10.1.5), declaraciones de miembros de interfaz (interface-member-declaration) (§13.2), declaraciones de miembros de struct (struct-member-declaration) (§11.2), declaraciones de miembros de enumeración (enum-member-declaration) (§14.3), declaraciones de descriptores de acceso (accessor-declaration) (§10.7.2), declaraciones de descriptores de acceso a eventos (event-accessor-declarations) (§10.8.1) y listas de parámetros formales (formal-parameter-lists) (§10.6.1).

Los atributos se especifican en secciones de atributos. Una sección de atributos se compone de un par de corchetes, que encierran una lista separada por comas de uno o varios atributos. Ni el orden de los atributos en la lista, ni el de las secciones adjuntas a la misma entidad de programa son significativos. Por ejemplo, las especificaciones de atributos [A][B], [B][A], [A, B] y [B, A] son equivalentes.

global-attributes:  
global-attribute-sections

global-attribute-sections:  
global-attribute-section  
global-attribute-sections global-attribute-section

global-attribute-section:  
[ global-attribute-target-specifier attribute-list ]  
[ global-attribute-target-specifier attribute-list , ]

global-attribute-target-specifier:  
global-attribute-target :

global-attribute-target:  
assembly  
module

attributes:  
attribute-sections

attribute-sections:  
attribute-section  
attribute-sections attribute-section

attribute-section:  
[ attribute-target-specifieropt attribute-list ]  
[ attribute-target-specifieropt attribute-list , ]

attribute-target-specifier:  
attribute-target :

attribute-target:  
field  
event  
method  
param  
property  
return  
type

attribute-list:  
attribute  
attribute-list , attribute

attribute:  
attribute-name attribute-argumentsopt

attribute-name:  
 type-name

attribute-arguments:  
( positional-argument-listopt )  
( positional-argument-list , named-argument-list )  
( named-argument-list )

positional-argument-list:  
positional-argument  
positional-argument-list , positional-argument

positional-argument:  
argument-nameopt attribute-argument-expression

named-argument-list:  
named-argument  
named-argument-list , named-argument

named-argument:  
identifier = attribute-argument-expression

attribute-argument-expression:  
expression

Un atributo se compone de un nombre de atributo (attribute-name) y una lista opcional de argumentos posicionales y con nombre. Los argumentos posicionales (si los hay) preceden a los argumentos con nombre. Un argumento posicional se compone de una expresión de argumentos de atributo (attribute-argument-expression). Un argumento con nombre consta de un nombre seguido por el signo igual, seguido de una expresión de argumentos de atributo (attribute-argument-expression) que, en conjunto, están restringidos por las mismas reglas que la asignación simple. El orden de los argumentos con nombre no es significativo.

El nombre de atributo (attribute-name) identifica una clase de atributos. Si la forma del nombre de atributo (attribute-name) es un nombre de tipo (type-name), este nombre debe hacer referencia a una clase de atributos. En caso contrario, se producirá un error en tiempo de compilación. En el ejemplo

class Class1 {}

[Class1] class Class2 {} // Error

se produce un error en tiempo de compilación, porque se intenta utilizar Class1 como clase de atributo a pesar de no serlo.

Algunos contextos permiten especificar un atributo en más de un destino. Un programa puede especificar explícitamente el destino incluyendo un especificador de destino de atributo (attribute-target-specifier). Cuando se coloca un atributo en el nivel global, es necesario un especificador de destino de atributo global (global-attribute-target-specifier). En todas las demás ubicaciones, se aplica un valor predeterminado razonable, pero es posible utilizar un especificador de destino de atributo (attribute-target-specifier) para afirmar o invalidar el valor predeterminado en ciertos casos ambiguos (o sólo para afirmar el valor predeterminado, en casos en que no haya ambigüedad). De esta forma, pueden omitirse habitualmente los especificadores de destino de atributo (attribute-target-specifier). Los contextos potencialmente ambiguos se resuelven de esta forma:

* Un atributo especificado en el ámbito global se puede aplicar tanto al ensamblado de destino como al módulo de destino. No existe un valor predeterminado para este contexto, por lo que siempre se requiere un especificador de destino de atributo (attribute-target-specifier). La presencia del especificador de destino de atributo (attribute-target-specifier) assembly de indica que el atributo se aplica al ensamblado de destino; la presencia del especificador de destino de atributo module indica que el atributo se aplica al módulo de destino.
* Un atributo especificado en una declaración de delegado se puede aplicar al delegado que se está declarando o a su valor devuelto. En ausencia del especificador de destino de atributo (attribute-target-specifier), el atributo se aplica al delegado. La presencia del especificador de destino de atributo (attribute-target-specifier) type indica que el atributo se aplica al delegado; la presencia del especificador de destino de atributo return indica que el atributo se aplica al valor devuelto.
* Un atributo especificado en una declaración de método se puede aplicar al método que se está declarando o a su valor devuelto. En ausencia del especificador de destino de atributo (attribute-target-specifier), el atributo se aplica al método. La presencia del especificador de destino de atributo (attribute-target-specifier) method indica que el atributo se aplica al método; la presencia del especificador de destino de atributo return indica que el atributo se aplica al valor devuelto.
* Un atributo especificado en una declaración de operador se puede aplicar al operador que se está declarando o a su valor devuelto. En ausencia de un especificador de destino de atributo (attribute-target-specifier), el atributo se aplica al operador. La presencia del especificador de destino de atributo (attribute-target-specifier) method indica que el atributo se aplica al operador; la presencia del especificador de destino de atributo return indica que el atributo se aplica al valor devuelto.
* Un atributo especificado en una declaración de evento que omite los descriptores de acceso de eventos se puede aplicar al evento que se está declarando, al campo asociado (si el evento no es abstracto) o a los métodos add y remove. En ausencia del especificador de destino de atributo (attribute-target-specifier), el atributo se aplica al evento. La presencia del especificador de destino de atributo (attribute-target-specifier) event indica que el atributo se aplica al evento; la presencia del especificador de destino de atributo field indica que el atributo se aplica al campo; y la presencia del especificador de destino de atributo method indica que el atributo se aplica a los métodos.
* Un atributo especificado en una declaración de descriptor de acceso get para una propiedad o una declaración de indizador se puede aplicar al método asociado o a su valor devuelto. En ausencia del especificador de destino de atributo (attribute-target-specifier), el atributo se aplica al método. La presencia del especificador de destino de atributo (attribute-target-specifier) method indica que el atributo se aplica al método; la presencia del especificador de destino de atributo return indica que el atributo se aplica al valor devuelto.
* Un atributo especificado en una declaración de descriptor de acceso set para una propiedad o una declaración de indizador se puede aplicar al método asociado o a su parámetro implícito solo. En ausencia del especificador de destino de atributo (attribute-target-specifier), el atributo se aplica al método. La presencia del especificador de destino de atributo (attribute-target-specifier) method indica que el atributo se aplica al método; la presencia del especificador de destino de atributo param indica que el atributo se aplica al parámetro; y la presencia del especificador de destino de atributo return indica que el atributo se aplica al valor devuelto.
* Un atributo especificado en una declaración de descriptor de acceso add o remove para una declaración de evento se aplica al método asociado o a su parámetro. En ausencia del especificador de destino de atributo (attribute-target-specifier), el atributo se aplica al método. La presencia del especificador de destino de atributo (attribute-target-specifier) method indica que el atributo se aplica al método; la presencia del especificador de destino de atributo param indica que el atributo se aplica al parámetro; y la presencia del especificador de destino de atributo return indica que el atributo se aplica al valor devuelto.

En otros contextos se permite la inclusión de un especificador de destino de atributo (attribute-target-specifier), pero no es necesaria. Por ejemplo, una declaración de clase puede incluir u omitir el especificador type:

[type: Author("Brian Kernighan")]  
class Class1 {}

[Author("Dennis Ritchie")]  
class Class2 {}

Supone un error especificar un especificador de destino de atributo (attribute-target-specifier) no válido. Por ejemplo, el especificador param no se puede utilizar en una declaración de clase:

[param: Author("Brian Kernighan")] // Error  
class Class1 {}

Por convención, las clases de atributos se denominan con el sufijo Attribute. Un nombre de atributo (attribute-name) con la estructura de nombre de tipo (type-name) puede incluir u omitir este sufijo. Si se encuentra una clase de atributos tanto con este sufijo como sin él, existe una ambigüedad y se produce un error de tiempo de compilación. Si el nombre de atributo (attribute-name) se escribe de manera que su identificador (identifier) situado más a la derecha es un identificador literal (§2.4.2), solo se corresponderá con un atributo sin sufijo, lo que supondrá una ambigüedad que se debe resolver. El ejemplo

using System;

[AttributeUsage(AttributeTargets.All)]  
public class X: Attribute  
{}

[AttributeUsage(AttributeTargets.All)]  
public class XAttribute: Attribute  
{}

[X] // Error: ambiguity  
class Class1 {}

[XAttribute] // Refers to XAttribute  
class Class2 {}

[@X] // Refers to X  
class Class3 {}

[@XAttribute] // Refers to XAttribute  
class Class4 {}

muestra dos clases de atributos denominadas X y XAttribute. El atributo [X] es ambiguo, puesto que podría referirse tanto a X como a XAttribute. El uso de un identificador literal permite especificar la intención exacta en estos casos poco frecuentes. El atributo [XAttribute] no es ambiguo (aunque lo sería si hubiera una clase de atributo denominada XAttributeAttribute!). Si se quita la declaración de la clase X, ambos atributos hacen referencia a la clase de atributo XAttribute, de la manera siguiente:

using System;

[AttributeUsage(AttributeTargets.All)]  
public class XAttribute: Attribute  
{}

[X] // Refers to XAttribute  
class Class1 {}

[XAttribute] // Refers to XAttribute  
class Class2 {}

[@X] // Error: no attribute named "X"  
class Class3 {}

Supone un error en tiempo de compilación utilizar un atributo de uso único más de una vez en la misma entidad. En el ejemplo

using System;

[AttributeUsage(AttributeTargets.Class)]  
public class HelpStringAttribute: Attribute  
{  
 string value;

public HelpStringAttribute(string value) {  
 this.value = value;  
 }

public string Value {  
 get {...}  
 }  
}

[HelpString("Description of Class1")]  
[HelpString("Another description of Class1")]  
public class Class1 {}

se produce un error en tiempo de compilación, porque intenta utilizar HelpString, una clase de atributo de uso único, más de una vez en la declaración de Class1.

Una expresión E es una expresión de argumentos de atributo (attribute-argument-expression) si todas las instrucciones siguientes son verdaderas:

* El tipo de E es un tipo de parámetro de atributo (§17.1.3).
* En tiempo de compilación, el valor de E se puede calcular a través de:
* Un valor constante.
* Un objeto System.Type.
* Una matriz unidimensional de expresiones de argumento de atributo (attribute-argument-expression).

Por ejemplo:

using System;

[AttributeUsage(AttributeTargets.Class)]  
public class TestAttribute: Attribute  
{  
 public int P1 {  
 get {...}  
 set {...}  
 }

public Type P2 {  
 get {...}  
 set {...}  
 }

public object P3 {  
 get {...}  
 set {...}  
 }  
}

[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]  
class MyClass {}

Una expresión typeof (typeof-expression) (§7.6.11) utilizada como una expresión de argumentos de atributo puede hacer referencia a un tipo no genérico, un tipo construido cerrado o un tipo genérico independiente, pero no puede hacer referencia a un tipo abierto. Así se garantiza que la expresión no se puede resolver en tiempo de compilación.

class A: Attribute  
{  
 public A(Type t) {...}  
}

class G<T>  
{  
 [A(typeof(T))] T t; // Error, open type in attribute  
}

class X  
{  
 [A(typeof(List<int>))] int x; // Ok, closed constructed type  
 [A(typeof(List<>))] int y; // Ok, unbound generic type  
}

## Instancias de atributo

Una instancia de atributo es una instancia que representa un atributo en tiempo de ejecución. Un atributo se define mediante una clase de atributos, argumentos posicionales y argumentos con nombre. Una instancia de atributo es una instancia de la clase de atributos que se inicializa con los argumentos posicionales y con nombre.

La recuperación de la instancia de atributo comprende el procesamiento en tiempo de compilación y ejecución, tal como se describe en las siguientes secciones:

### Compilación de un atributo

La compilación de un atributo (attribute) con clase de atributos T, una lista de argumentos posicionales (positional-argument-list) P y una lista de argumentos con nombre (named-argument-list) N se compone de los siguientes pasos:

* Para compilar una expresión de creación de objeto de la forma new T(P), se siguen los pasos del procesamiento en tiempo de compilación. Estos pasos pueden producir un error en tiempo de compilación o determinar un constructor de instancia C en T, que se puede invocar en tiempo de ejecución.
* Si C no tiene acceso público, se produce un error en tiempo de compilación.
* Para cada argumento con nombre (named-argument) Arg de N:
* Name será el identificador (identifier) del argumento con nombre (named-argument) Arg.
* Name debe identificar a un campo o propiedad público, de lectura y escritura y no estático en T. Si T no tiene tal campo o propiedad, se producirá un error en tiempo de compilación.
* Mantenga la siguiente información para la creación de instancias del atributo en tiempo de ejecución: la clase de atributos T, el constructor de instancia C de T, la lista de argumentos posicionales (positional-argument-list) P y la lista de argumentos con nombre (named-argument-list) N.

### Recuperación en tiempo de ejecución de una instancia de atributo

La compilación de un atributo (attribute) da como resultado una clase de atributos T, un constructor de instancia C en T, una lista de argumentos posicionales (positional-argument-list) P y una lista de argumentos con nombre (named-argument-list) N. Dada esta información, se puede recuperar una instancia de atributo en tiempo de ejecución siguiendo estos pasos:

* Se siguen los pasos de procesamiento en tiempo de ejecución para ejecutar una expresión de creación de objeto (object-creation-expression) de la forma new T(P), utilizando el constructor de instancia C tal como se determina en tiempo de compilación. Estos pasos dan como resultado una excepción, o producen una instancia O de T.
* Para cada argumento con nombre (named-argument) Arg de N, con el orden:
* Name será el identificador (identifier) del argumento con nombre (named-argument) Arg. Si Name no identifica a un campo o propiedad de lectura y escritura, público y no estático en O, se inicia una excepción.
* Value será el resultado de evaluar la expresión de argumentos de atributo (attribute-argument-expression) de Arg.
* Si Name identifica un campo en O, establezca este campo en Value.
* En cualquier otro caso, Name identifica a una propiedad en O. Establezca esta propiedad en Value.
* El resultado es O, una instancia de la clase de atributos T inicializada con la lista de argumentos posicionales (positional-argument-list) P y la lista de argumentos con nombre (named-argument-list) N.

## Atributos reservados

Un pequeño número de atributos afectan al lenguaje de alguna forma. Estos atributos incluyen:

* System.AttributeUsageAttribute (§17.4.1), que se usa para describir las formas en que puede utilizarse una clase de atributos.
* System.Diagnostics.ConditionalAttribute (§17.4.2), que se usa para definir métodos condicionales.
* System.ObsoleteAttribute (§17.4.3), que se usa para marcar un miembro como obsoleto.
* System.Runtime.CompilerServices.CallerLineNumberAttribute, System.Runtime.CompilerServices.CallerFilePathAttribute y System.Runtime.CompilerServices.CallerMemberNameAttribute (§17.4.4), que se usan para proporcionar información sobre el contexto de la llamada a los parámetros opcionales.

### Atributo AttributeUsage

El atributo AttributeUsage se utiliza para describir la manera en que se puede utilizar la clase de atributos.

Una clase decorada con el atributo AttributeUsage debe derivarse de System.Attribute, ya sea de forma directa o indirecta. En caso contrario, se producirá un error en tiempo de compilación.

namespace System  
{  
 [AttributeUsage(AttributeTargets.Class)]  
 public class AttributeUsageAttribute: Attribute  
 {  
 public AttributeUsageAttribute(AttributeTargets validOn) {...}

public virtual bool AllowMultiple { get {...} set {...} }

public virtual bool Inherited { get {...} set {...} }

public virtual AttributeTargets ValidOn { get {...} }  
 }

public enum AttributeTargets  
 {  
 Assembly = 0x0001,  
 Module = 0x0002,  
 Class = 0x0004,  
 Struct = 0x0008,  
 Enum = 0x0010,  
 Constructor = 0x0020,  
 Method = 0x0040,  
 Property = 0x0080,  
 Field = 0x0100,  
 Event = 0x0200,  
 Interface = 0x0400,  
 Parameter = 0x0800,  
 Delegate = 0x1000,  
 ReturnValue = 0x2000,

All = Assembly | Module | Class | Struct | Enum | Constructor |   
 Method | Property | Field | Event | Interface | Parameter |   
 Delegate | ReturnValue  
 }  
}

### Atributo Conditional

El atributo Conditional \b habilita la definición de métodos condicionales y clases de atributos condicionales.

namespace System.Diagnostics  
{  
 [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,  
 AllowMultiple = true)]  
 public class ConditionalAttribute: Attribute  
 {  
 public ConditionalAttribute(string conditionString) {...}

public string ConditionString { get {...} }  
 }  
}

#### Métodos condicionales

Un método decorado con el atributo Conditional es un método condicional. El atributo Conditional indica una condición cuando se comprueba un símbolo de compilación condicional. Las llamadas a un método condicional se incluyen u omiten dependiendo de si está definido este símbolo en el punto de la llamada. Si el símbolo se define, la llamada se incluye; en caso contrario, la llamada (incluida la evaluación de los parámetros y receptor de la llamada) se omite.

Un método condicional está sujeto a las siguientes restricciones:

* El método condicional debe ser un método perteneciente a una declaración de clase (class-declaration) o de struct (struct-declaration). Si se especifica el atributo Conditional en un método de declaración de interfaz, se produce un error en tiempo de compilación.
* El método condicional debe tener un tipo de valor devuelto void.
* El método condicional no debe estar marcado con el modificador override. Sin embargo, un método condicional puede marcarse con el modificador virtual. Las invalidaciones de dicho método son condicionales de forma implícita, y no se deben marcar explícitamente con un atributo Conditional.
* El método condicional no debe ser una implementación de un método de interfaz. En caso contrario, se producirá un error en tiempo de compilación.

Asimismo, se produce un error en tiempo de compilación si se utiliza un método condicional en una expresión de creación de delegados (delegate-creation-expression). El ejemplo

#define DEBUG

using System;  
using System.Diagnostics;

class Class1   
{  
 [Conditional("DEBUG")]  
 public static void M() {  
 Console.WriteLine("Executed Class1.M");  
 }  
}

class Class2  
{  
 public static void Test() {  
 Class1.M();  
 }  
}

declara Class1.M como método condicional. El método Test de Class2 llama a este método. Dado que se define el símbolo de compilación condicional DEBUG, si se llama a Class2.Test, este llamará a M. Si el símbolo DEBUG no se hubiera definido, Class2.Test no llamaría a Class1.M.

Es importante observar que la inclusión o exclusión de una llamada a un método condicional está controlada por los símbolos de compilación condicional existentes en el lugar de la llamada. En el ejemplo

Archivo class1.cs:

using System.Diagnostics;

class Class1   
{  
 [Conditional("DEBUG")]  
 public static void F() {  
 Console.WriteLine("Executed Class1.F");  
 }  
}

Archivo class2.cs:

#define DEBUG

class Class2  
{  
 public static void G() {  
 Class1.F(); // F is called  
 }  
}

Archivo class3.cs:

#undef DEBUG

class Class3  
{  
 public static void H() {  
 Class1.F(); // F is not called  
 }  
}

las clases Class2 y Class3 contienen llamadas al método condicional Class1.F, que será o no condicional, en función de si DEBUG está definido o no. Dado que este símbolo está definido en el contexto de Class2 pero no en el de Class3, la llamada a F en Class2 se mantiene, mientras que se omite la llamada a F en Class3.

El uso de métodos condicionales en una cadena de herencia puede resultar confuso. Las llamadas realizadas a un método condicional mediante base, de la forma base.M, están sujetas a las reglas normales de llamadas a métodos condicionales. En el ejemplo

Archivo class1.cs:

using System;  
using System.Diagnostics;

class Class1   
{  
 [Conditional("DEBUG")]  
 public virtual void M() {  
 Console.WriteLine("Class1.M executed");  
 }  
}

Archivo class2.cs:

using System;

class Class2: Class1  
{  
 public override void M() {  
 Console.WriteLine("Class2.M executed");  
 base.M(); // base.M is not called!  
 }  
}

Archivo class3.cs:

#define DEBUG

using System;

class Class3  
{  
 public static void Test() {  
 Class2 c = new Class2();  
 c.M(); // M is called  
 }  
}

Class2 incluye una llamada al método M definido en su clase base. Esta llamada se omite porque el método base es condicional debido a la presencia del símbolo DEBUG, que no está definido. Así, el método únicamente escribe en la consola “Class2.M executed”. Un uso juicioso de las declaraciones pp (pp-declaration) puede eliminar tales problemas.

#### Clases de atributos condicionales

Una clase de atributos (§17.1) decorada con uno o más atributos Conditional es una clase de atributos condicionales. Una clase de atributos condicionales se asocia, por lo tanto, con los símbolos de compilación condicional declarados en sus atributos Conditional. Este ejemplo:

using System;  
using System.Diagnostics;  
[Conditional("ALPHA")]  
[Conditional("BETA")]  
public class TestAttribute : Attribute {}

declara TestAttribute como una clase de atributos condicionales asociada a los símbolos de compilaciones condicionales ALPHA y BETA.

Las especificaciones de atributos (§17.2) de un atributo condicional se incluyen si uno o más de sus símbolos de compilación condicional asociado se define en el lugar de la especificación; de lo contrario, se omite la especificación de atributos.

Es importante observar que la inclusión o exclusión de la especificación de un atributo de una clase de atributos condicionales está controlada por los símbolos de compilación condicional existentes en el lugar de la especificación. En el ejemplo

Archivo test.cs:

using System;  
using System.Diagnostics;

[Conditional("DEBUG")]

public class TestAttribute : Attribute {}

Archivo class1.cs:

#define DEBUG

[Test] // TestAttribute is specified

class Class1 {}

Archivo class2.cs:

#undef DEBUG

[Test] // TestAttribute is not specified

class Class2 {}

las clases Class1 y Class2 están decoradas con el atributo Test, que es condicional en función de si DEBUG está o no definido. Dado que este símbolo está definido en el contexto de Class1 pero no de Class2, se incluye la especificación del atributo Test en Class1, mientras que se omite la especificación del atributo Test en Class2.

### Atributo Obsolete

El atributo Obsolete se utiliza para marcar tipos y miembros de tipos que ya no se deberían utilizar.

namespace System  
{  
 [AttributeUsage(  
 AttributeTargets.Class |   
 AttributeTargets.Struct |  
 AttributeTargets.Enum |   
 AttributeTargets.Interface |   
 AttributeTargets.Delegate |  
 AttributeTargets.Method |   
 AttributeTargets.Constructor |  
 AttributeTargets.Property |   
 AttributeTargets.Field |  
 AttributeTargets.Event,  
 Inherited = false)  
 ]  
 public class ObsoleteAttribute: Attribute  
 {  
 public ObsoleteAttribute() {...}

public ObsoleteAttribute(string message) {...}

public ObsoleteAttribute(string message, bool error) {...}

public string Message { get {...} }

public bool IsError { get {...} }  
 }  
}

Si un programa utiliza un tipo o miembro decorado con el atributo Obsolete, el compilador emite una advertencia o genera un error. En concreto, si no se proporciona ningún parámetro de error, o si se proporciona y su valor es false, el compilador emite una advertencia. Si se especifica un parámetro de error y su valor es true, el compilador genera un error.

En el ejemplo

[Obsolete("This class is obsolete; use class B instead")]  
class A  
{  
 public void F() {}  
}

class B  
{  
 public void F() {}  
}

class Test  
{  
 static void Main() {  
 A a = new A(); // Warning  
 a.F();  
 }  
}

la clase A está decorada con el atributo Obsolete. Cada uso de A en Main da como resultado una advertencia que incluye el mensaje especificado en el código anterior, “This class is obsolete; use class B instead.”

### Atributos de información del llamador

Para propósitos como el registro y la creación de informes, a veces resulta útil que un miembro de función obtenga cierta información en tiempo de compilación acerca del código de llamada. Los atributos de información del llamador proporcionan una manera de pasar esa información de forma transparente.

Cuando se anota un parámetro opcional con uno de los atributos de información del llamador, la omisión del argumento correspondiente en una llamada no hace necesariamente que se sustituya el valor de parámetro predeterminado. En su lugar, si la información especificada sobre el contexto de llamada está disponible, esa información se pasará como valor del argumento.

Por ejemplo:

using System.Runtime.CompilerServices

…

public void Log(  
 [CallerLineNumber] int line = -1,  
 [CallerFilePath] string path = null,  
 [CallerMemberName] string name = null  
)  
{  
 Console.WriteLine((line < 0) ? "No line" : "Line "+ line);  
 Console.WriteLine((path == null) ? "No file path" : path);  
 Console.WriteLine((name == null) ? "No member name" : name);  
}

Una llamada a Log() sin argumentos imprimiría el número de línea y la ruta de acceso de archivo de la llamada, así como el nombre del miembro en el que se produjo la llamada.

Los atributos de información del llamador pueden aparecer en parámetros opcionales en cualquier parte, incluso en las declaraciones de delegado. Sin embargo, los atributos de información del llamador concretos tienen restricciones en cuanto a los tipos de parámetros que pueden atribuir, por lo que siempre habrá una conversión implícita de un valor sustituido al tipo de parámetro.

Se produce un error al tener el mismo atributo de información del llamador en un parámetro de la parte de definición y de implementación de una declaración de método parcial. Solo se aplican los atributos de información del llamador en la parte de la definición, mientras que los atributos de información del llamador que solo aparecen en la parte de implementación se omiten.

La información del llamador no afecta a la resolución de sobrecarga. Debido a que los parámetros opcionales atribuidos se siguen omitiendo del código fuente del llamador, la resolución de sobrecarga omite esos parámetros de la misma forma que omite otros parámetros opcionales omitidos (§7.5.3).

La información del llamador solo se sustituye cuando se llama explícitamente a una función en el código fuente. Las invocaciones implícitas, como las llamadas implícitas de constructor primario no tienen una ubicación de origen y no sustituirán a la información del llamador. Además, las llamadas que se enlazan dinámicamente no sustituirán a la información del llamador. Cuando se omite un parámetro de atributos de información del llamador en tales casos, se usa en su lugar el valor predeterminado especificado del parámetro.

Una excepción son las expresiones de consulta. Se consideran expansiones sintácticas y, si las llamadas que expanden omiten parámetros opcionales con atributos de información del llamador, se sustituirá la información del llamador. La ubicación utilizaba la ubicación de la cláusula de consulta a partir de la que se generó la llamada.

Si se especifica más de un atributo de información del llamador en un parámetro dado, se prefieren en el orden siguiente: CallerLineNumber, CallerFilePath, CallerMemberName.

#### Atributo CallerLineNumber

Se permite System.Runtime.CompilerServices.CallerLineNumberAttribute en los parámetros opcionales cuando hay una conversión implícita estándar (§6.3.1) del valor constante int.MaxValue al tipo del parámetro. Esto garantiza que cualquier número de línea no negativo hasta ese valor pueda pasarse sin error.

Si una invocación de función de una ubicación del código fuente omite un parámetro opcional con CallerLineNumberAttribute, un literal numérico que representa el número de línea de dicha ubicación se usa como argumento de la invocación en lugar de utilizar el valor de parámetro predeterminado.

Si la invocación ocupa varias líneas, la línea elegida depende de la implementación.

Observe que el número de línea puede verse afectado por las directivas de #line (§2.5.7).

#### Atributo CallerFilePath

Se permite System.Runtime.CompilerServices.CallerFilePathAttribute en los parámetros opcionales cuando hay una conversión implícita estándar (§6.3.1) de string al tipo del parámetro.

Si una invocación de función de una ubicación del código fuente omite un parámetro opcional con CallerFilePathAttribute, un literal de cadena que representa la ruta de acceso de archivo de dicha ubicación se usa como argumento de la invocación en lugar de utilizar el valor de parámetro predeterminado.

El formato de la ruta de acceso al archivo depende de la implementación.

Observe que la ruta de acceso de archivo puede verse afectada por las directivas de #line (§2.5.7).

#### Atributo CallerMemberName

Se permite System.Runtime.CompilerServices.CallerMemberNameAttribute en los parámetros opcionales cuando hay una conversión implícita estándar (§6.3.1) de string al tipo del parámetro.

Si una invocación de función de una ubicación dentro del cuerpo de un miembro de función o dentro de un atributo aplicado al propio miembro de función o su tipo de valor devuelto, parámetros o parámetros de tipo en el código fuente omite un parámetro opcional con CallerMemberNameAttribute, se usa un literal de cadena que representa el nombre de ese miembro como argumento para la invocación en lugar del valor de parámetro predeterminado.

Para las invocaciones que tienen lugar en métodos genéricos, solo se usa el propio nombre de método, si la lista de parámetros de tipo.

Para las invocaciones que tienen lugar en implementaciones de miembro de interfaz explícitas, solo se usa el propio nombre de método, sin la calificación de interfaz anterior.

Para las invocaciones que tienen lugar dentro de los descriptores de acceso de propiedad o de evento, el nombre de miembro que se usa es el de la propiedad o evento en sí.

Para las invocaciones que tienen lugar dentro de los descriptores de acceso de indizador, el nombre de miembro que se usa es el suministrado por IndexerNameAttribute (§17.5.2.1) en el miembro indizador, si está presente, o de lo contrario, el nombre predeterminado Item.

Para las invocaciones que tienen lugar en las declaraciones de constructores de instancia, constructores estáticos, destructores y operadores, el nombre de miembro utilizado depende de la implementación.

## Atributos para interoperabilidad

Nota: esta sección solo se aplica a la implementación de C# de Microsoft .NET.

### Interoperabilidad con componentes COM y Win32

El motor de tiempo de ejecución de .NET proporciona un gran número de atributos que permiten que los programas de C# interactúen con componentes escritos mediante bibliotecas .dll COM y Win32. Por ejemplo, el atributo DllImport puede utilizarse en un método static extern para indicar que la implementación del método se encuentra en una biblioteca DLL Win32. Estos atributos se encuentran en el espacio de nombres System.Runtime.InteropServices; se puede encontrar información detallada referente a estos atributos en la documentación del runtime de .NET Framework.

### Interoperabilidad con otros lenguajes .NET

#### Atributo IndexerName

Los indizadores se implementan en .NET utilizando propiedades indizadas y tienen un nombre en los metadatos .NET. Si no hay ningún atributo IndexerName presente para un indizador, se utiliza de forma predeterminada el nombre Item. El atributo IndexerName permite al programador invalidar este valor predeterminado y especificar un nombre distinto.

namespace System.Runtime.CompilerServices.CSharp  
{  
 [AttributeUsage(AttributeTargets.Property)]  
 public class IndexerNameAttribute: Attribute  
 {  
 public IndexerNameAttribute(string indexerName) {...}

public string Value { get {...} }   
 }  
}

# Código no seguro

El lenguaje C# básico, tal como se define en los capítulos precedentes, difiere notablemente de C y C++ en su omisión de los punteros como un tipo de datos. C# proporciona en su lugar referencias, así como la capacidad de crear objetos administrados por un recolector de elementos no utilizados. Este diseño, unido a otras características, hace de C# un lenguaje mucho más seguro que C o C++. En el lenguaje básico de C# no es posible usar una variable no inicializada, un puntero “pendiente” o una expresión que indice una matriz más allá de sus límites. De esta manera se eliminan categorías enteras de errores que suelen afectar a los programas desarrollados en C y C++.

Si bien casi todas las construcciones de tipos de puntero en C o C++ poseen un tipo de referencia equivalente en C#, hay situaciones donde el acceso a tipos de puntero se convierte en una necesidad. Por ejemplo, al desear interactuar con el sistema operativo subyacente, obtener acceso a un dispositivo asignado a memoria o implementar un algoritmo sensible al tiempo puede no lograrse nuestro objetivo sin disponer de acceso a punteros. Para solucionar esta necesidad, C# proporciona la capacidad de crear código no seguro.

En el código no seguro es posible declarar y operar con punteros, realizar conversiones de punteros a tipos enteros, tomar las direcciones de variables, etc. En cierto sentido, escribir código no seguro se parece mucho a escribir código C dentro de un programa en C#.

El código no seguro es, en realidad, una característica “segura” desde el punto de vista tanto de programadores como de usuarios. El código no seguro debe estar marcado con claridad mediante el modificador unsafe, de manera que los programadores no puedan usar características no seguras por accidente, y el motor de ejecución garantice que no se pueda ejecutar el código no seguro en un entorno que no sea de confianza.

## Contextos no seguros

Las características no seguras de C# están disponibles solo en contextos no seguros. Se introduce un contexto no seguro incluyendo un modificador unsafe en la declaración de un tipo o miembro, o utilizando una instrucción unsafe (unsafe-statement).

* Una declaración de una clase, struct, interfaz o delegado puede incluir un modificador unsafe, en cuyo caso toda la extensión textual de la declaración (incluido el cuerpo de la clase, struct o interfaz) se considera como contexto no seguro.
* Una declaración de un campo, método, propiedad, evento, indizador, operador, constructor de instancia, constructor estático o destructor puede incluir el modificador unsafe, en cuyo caso toda la extensión textual de dicha declaración de miembro se considera como contexto no seguro.
* Una instrucción unsafe (unsafe-statement) habilita el uso de un contexto no seguro dentro de un bloque (block). Toda la extensión textual del bloque (block) asociado se considera contexto no seguro.

Las extensiones de gramática asociadas se muestran a continuación. Para mayor brevedad, se utilizan puntos suspensivos (...) para representar elementos que aparecen explicados en capítulos anteriores.

class-modifier:  
...  
unsafe

struct-modifier:  
...  
unsafe

interface-modifier:  
...  
unsafe

delegate-modifier:  
...  
unsafe

field-modifier:  
...  
unsafe

method-modifier:  
...  
unsafe

property-modifier:  
...  
unsafe

event-modifier:  
...  
unsafe

indexer-modifier:  
...  
unsafe

operator-modifier:  
...  
unsafe

constructor-modifier:  
...  
unsafe

destructor-declaration:  
attributesopt externopt unsafeopt ~ identifier ( ) destructor-body  
attributesopt unsafeopt externopt ~ identifier ( ) destructor-body

static-constructor-modifiers:  
externopt unsafeopt static  
unsafeopt externopt static  
externopt static unsafeopt   
unsafeopt static externopt  
static externopt unsafeopt  
static unsafeopt externopt

embedded-statement:  
...  
unsafe-statement

unsafe-statement:  
unsafe block

En el ejemplo

public unsafe struct Node  
{  
 public int Value;  
 public Node\* Left;  
 public Node\* Right;  
}

el modificador unsafe especificado en la declaración de struct convierte toda la extensión textual de la declaración en contexto no seguro. Así, es posible declarar los campos Left y Right como tipo de puntero. El ejemplo anterior también se puede escribir de la siguiente forma:

public struct Node  
{  
 public int Value;  
 public unsafe Node\* Left;  
 public unsafe Node\* Right;  
}

Aquí, los modificadores unsafe de las declaraciones de campo las convierten en contextos no seguros.

Aparte de establecer un contexto no seguro, que permite así el uso de tipos de puntero, el modificador unsafe no tiene ningún efecto en un tipo o miembro. En el ejemplo

public class A  
{  
 public unsafe virtual void F() {  
 char\* p;  
 ...  
 }  
}

public class B: A  
{  
 public override void F() {  
 base.F();  
 ...  
 }  
}

el modificador unsafe del método F en A simplemente hace que la extensión textual de F se convierta en un contexto no seguro en el cual pueden utilizarse las características no seguras del lenguaje. En la invalidación de F de B, no hay necesidad de volver a especificar el modificador unsafe, a menos que el propio método F de B requiera el acceso a características no seguras.

Esta situación es ligeramente diferente cuando un tipo de puntero forma parte de la firma del método.

public unsafe class A  
{  
 public virtual void F(char\* p) {...}  
}

public class B: A  
{  
 public unsafe override void F(char\* p) {...}  
}

Aquí, debido a que la signatura de F incluye un tipo de puntero, solo se puede escribir en un contexto no seguro. Sin embargo, el contexto no seguro se puede introducir convirtiendo toda la clase en no segura, como en el caso de A, o al incluir un modificador unsafe en la declaración de método, como en el caso de B.

## Tipos de puntero

En un contexto no seguro, un tipo (type) (§4) puede ser un tipo de puntero (pointer-type), así como un tipo de valor (value-type) o un tipo de referencia (reference-type). Sin embargo, un tipo de puntero (*pointer-type*) también puede utilizarse en una expresión typeof (§7.6.10.6) fuera de un contexto no seguro ya que dicho uso no es no seguro.

type:  
...  
pointer-type

Un tipo de puntero (pointer-type) se escribe como tipo no administrado (unmanaged-type) o como la palabra clave void seguida de un token \*:

pointer-type:  
unmanaged-type \*  
void \*

unmanaged-type:  
type

El tipo especificado antes del asterisco (\*) en un tipo de puntero se denomina tipo referente del tipo de puntero. Representa el tipo de la variable a la que señala un valor del tipo de puntero.

Al contrario que las referencias (valores de tipos de referencia), los punteros no están sometidos al seguimiento del recolector de elementos no utilizados, el cual no conoce los punteros ni los datos a los que estos apuntan. Por este motivo, un puntero no puede señalar a una referencia o a un struct que contenga referencias, y el tipo referente de un puntero debe ser un tipo no administrado (unmanaged-type).

Un tipo no administrado (unmanaged-type) es cualquier tipo que no sea un tipo de referencia (reference-type) o un tipo construido y no contenga campos de tipo de referencia ni de tipo construido en ningún nivel de anidamiento. En otras palabras, un tipo no administrado (unmanaged-type) es uno de los siguientes:

* sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal o bool.
* Cualquier tipo enum (enum-type).
* Cualquier tipo de puntero (pointer-type).
* Cualquier tipo struct (struct-type) definido por el usuario que no sea un tipo construido y que contenga únicamente campos de tipos no administrados (unmanaged-type).

La regla intuitiva para mezclar punteros y referencias es que los referentes de las referencias (objetos) pueden contener punteros, pero los referentes de punteros no pueden contener referencias.

En la tabla siguiente se dan algunos ejemplos de tipos de punteros:

|  |  |
| --- | --- |
| **Ejemplo** | **Descripción** |
| byte\* | Puntero a byte |
| char\* | Puntero a char |
| int\*\* | Puntero a un puntero a int |
| int\*[] | Matriz unidimensional de punteros a int |
| void\* | Puntero a un tipo desconocido |

Para una implementación dada, todos los tipos de puntero deben tener el mismo tamaño y representación.

A diferencia de C y C++, cuando se declaran múltiples punteros en una misma declaración, en C# el carácter \* se escribe junto al tipo subyacente únicamente, no como puntuación de prefijo en cada nombre de puntero. Por ejemplo:

int\* pi, pj; // NOT as int \*pi, \*pj;

El valor de un puntero de tipo T\* representa la dirección (address) de una variable de tipo T. El operador de direccionamiento indirecto de puntero \* (§18.5.1) puede usarse para obtener acceso a esta variable. Por ejemplo, dada una variable P de tipo int\*, la expresión \*P denota la variable int hallada en la dirección contenida en P.

Al igual que una referencia de objeto, un puntero puede ser null. Aplicar el operador de direccionamiento indirecto a un puntero null da como resultado un comportamiento definido según la implementación. Un puntero con el valor null se representa por medio de todos los bits cero.

El tipo void\* representa un puntero a un tipo desconocido. Dado que el tipo referente es desconocido, el operador de direccionamiento indirecto no se puede aplicar a un puntero de tipo void\*, ni se pueden realizar operaciones aritméticas en este tipo de puntero. Sin embargo, un puntero de tipo void\* se puede convertir a cualquier otro tipo de puntero (y viceversa).

Los tipos de puntero forman una categoría independiente de tipos. A diferencia de los tipos de referencia y los tipos de valor, los tipos de puntero no se derivan de object y no existen conversiones entre tipos de puntero y object. En particular, las operaciones boxing y unboxing (§4.3) no se permiten en punteros. Sin embargo, se permiten las conversiones entre diferentes tipos de punteros y entre tipos de punteros y tipos enteros. Este proceso se describe en §18.4.

Un tipo de puntero (pointer-type) no se puede usar como argumento de tipo (§4.4) y la inferencia de tipos (§7.5.2) da error en llamadas de métodos genéricos que habrían inferido un argumento de tipo para que fuera un tipo de puntero.

Un tipo de puntero (pointer-type) no se puede usar como tipo de una expresión constitutiva de una operación enlazada dinámicamente (§7.2.2).

Puede utilizarse un tipo de puntero (pointer-type) como tipo de un campo volátil (§10.5.3).

Si bien pueden pasarse punteros como parámetros ref u out, hacerlo puede causar un comportamiento indefinido, ya que un puntero puede señalar a una variable local que ya no existe cuando el método llamado devuelve un valor, o el objeto fijo al que apuntaba ya no es de tipo fijado. Por ejemplo:

using System;

class Test  
{  
 static int value = 20;

unsafe static void F(out int\* pi1, ref int\* pi2) {  
 int i = 10;  
 pi1 = &i;

fixed (int\* pj = &value) {  
 // ...  
 pi2 = pj;  
 }  
 }

static void Main() {  
 int i = 10;  
 unsafe {  
 int\* px1;  
 int\* px2 = &i;

F(out px1, ref px2);

Console.WriteLine("\*px1 = {0}, \*px2 = {1}",  
 \*px1, \*px2); // undefined behavior  
 }  
 }  
}

Un método puede devolver un valor de un tipo, y ese tipo puede ser un puntero. Por ejemplo, dado un puntero a una secuencia contigua de valores int, el número de elementos de la secuencia y otro valor int, el método siguiente devuelve la dirección del valor indicado en la secuencia, si se encuentra una coincidencia; en cualquier otro caso devuelve null:

unsafe static int\* Find(int\* pi, int size, int value) {  
 for (int i = 0; i < size; ++i) {  
 if (\*pi == value)   
 return pi;  
 ++pi;  
 }  
 return null;  
}

En un contexto no seguro, hay varias construcciones disponibles para operar con punteros:

* Puede utilizarse el operador \* para realizar direccionamiento indirecto de punteros (§18.5.1).
* Puede utilizarse el operador -> para obtener acceso a un miembro de un struct a través de un puntero (§18.5.2).
* Puede utilizarse el operador [] para indizar un puntero (§18.5.3).
* Puede utilizarse el operador & para obtener la dirección de una variable (§18.5.4).
* Pueden utilizarse los operadores ++ y -- para incrementar y disminuir punteros (§18.5.5).
* Se pueden utilizar los operadores + y - para realizar aritmética de punteros (§18.5.6).
* Pueden utilizarse los operadores ==, !=, <, >, <= y => para comparar punteros (§18.5.7).
* El operador stackalloc puede utilizarse para asignar memoria de la pila de llamadas (§18.7).
* Puede utilizarse la instrucción fixed para fijar temporalmente una variable y obtener su dirección (§18.6).

## Variables fijas y móviles

El operador de dirección (§18.5.4) y la instrucción fixed (§18.6) dividen las variables en dos categorías: variables fijas y variables móviles.

Las variables fijas se encuentran en ubicaciones de almacenamiento que no se ven afectadas por el funcionamiento del recolector de elementos no utilizados. Los ejemplos de variables fijas incluyen variables locales, parámetros de valor y variables creados mediante punteros de desreferenciación. Las variables móviles, por otro lado, se encuentran en ubicaciones de almacenamiento sujetas a la reubicación o eliminación por parte del recolector de elementos no utilizados. Los ejemplos de variables móviles incluyen campos de objetos y elementos de matrices.

El operador & (§18.5.4) permite obtener la dirección de una variable fija sin restricciones. Sin embargo, debido a que una variable móvil está sujeta a la reubicación o eliminación por el recolector de elementos no utilizados, sólo se puede obtener su dirección mediante una instrucción fixed (§18.6), y la dirección es válida sólo durante el ciclo de duración de esa instrucción fixed.

En términos precisos, una variable fija es una de las siguientes:

* Una variable resultante de un nombre simple (simple-name) (§7.6.2) que hace referencia a una variable local o un parámetro de valor, a menos que la variable sea capturada por una función anónima.
* Una variable resultante de un acceso a miembro (member-access) (§7.6.4) de la forma V.I, donde V es una variable fija de tipo struct (struct-type).
* Una variable resultante de una expresión de direccionamiento indirecto de puntero (pointer-indirection-expression) (§18.5.1) de la forma \*P, un acceso a miembro de puntero (pointer-member-access) (§18.5.2) de la forma P->I o un acceso a elemento de puntero (pointer-element-access) (§18.5.3) de la forma P[E].

Todas las demás variables se clasifican como variables móviles.

Observe que un campo estático se clasifica también como variable móvil. Observe, igualmente, que un parámetro ref u out se clasifica como variable móvil, aun cuando el argumento dado para ese parámetro sea una variable fija. Por último, debe señalarse que una variable producida mediante la desreferenciación a un puntero siempre se clasifica como una variable fija.

## Conversiones de puntero

En un contexto no seguro, el conjunto de conversiones implícitas disponibles (§6.1) se extiende hasta incluir las siguientes conversiones implícitas de puntero:

* De cualquier tipo de puntero (pointer-type) al tipo void\*.
* Del literal null a cualquier tipo de puntero (pointer-type).

En un contexto no seguro, el conjunto de conversiones explícitas disponibles (§6.2) se extiende hasta incluir las siguientes conversiones explícitas de puntero:

* De cualquier tipo de puntero (pointer-type) a cualquier otro tipo de puntero.
* De sbyte, byte, short, ushort, int, uint, long o ulong a cualquier tipo de puntero (pointer-type).
* De cualquier tipo de puntero (pointer-type) a sbyte, byte, short, ushort, int, uint, long o ulong.

Por último, en un contexto no seguro, el conjunto de conversiones implícitas estándar (§6.3.1) incluye las siguientes conversiones de puntero:

* De cualquier tipo de puntero (pointer-type) al tipo void\*.
* Del literal null a cualquier tipo de puntero (pointer-type).

Las conversiones entre dos tipos de puntero nunca afectan al valor real del puntero. En otras palabras, una conversión de un tipo de puntero a otro no tiene efecto en la dirección subyacente dada por el puntero.

Cuando se convierte un tipo de puntero a otro, si el puntero resultante no está correctamente alineado para el tipo a que se apunta, el comportamiento será indefinido si se desreferencia al resultado. En general, el concepto “correctamente alineado” es transitivo: si un puntero a un tipo A está correctamente alineado para un puntero a un tipo B que, a su vez, está correctamente alineado para un puntero a un tipo C, entonces un puntero a un tipo A está correctamente alineado para un puntero a un tipo C.

Considérese el siguiente caso en el que se obtiene acceso a una variable de un tipo determinado mediante un puntero a otro tipo:

char c = 'A';  
char\* pc = &c;  
void\* pv = pc;  
int\* pi = (int\*)pv;  
int i = \*pi; // undefined  
\*pi = 123456; // undefined

Cuando se convierte un tipo de puntero a un puntero a byte, el resultado apunta al byte de dirección inferior de la variable. Los sucesivos incrementos del resultado, hasta alcanzar el tamaño de la variable, dan como resultado punteros a los bytes restantes de dicha variable. Por ejemplo, el método siguiente muestra cada uno de los ocho bytes de un tipo double como un valor hexadecimal:

using System;

class Test  
{  
 unsafe static void Main() {  
 double d = 123.456e23;  
 unsafe {  
 byte\* pb = (byte\*)&d;  
 for (int i = 0; i < sizeof(double); ++i)  
 Console.Write("{0:X2} ", \*pb++);  
 Console.WriteLine();  
 }  
 }  
}

Por supuesto, los resultados producidos dependen del carácter big-endian o little endian.

Las asignaciones entre punteros y enteros se definen según su implementación. Sin embargo, en arquitecturas de CPU de 32 y 64 bits, con un espacio de direcciones lineal, las conversiones de punteros a o desde tipos enteros tienen lugar exactamente igual que las conversiones de valores uint o ulong, respectivamente, a o desde dichos tipos enteros.

### Matrices de punteros

En un contexto no seguro, se pueden construir matrices de punteros. En matrices de punteros, solamente se permiten algunas de las conversiones que se aplican a otros tipos de matrices:

* La conversión de referencia implícita (§6.1.6) desde cualquier tipo de matriz (array-type) a System.Array y las interfaces que implementa también se aplica a matrices de punteros. No obstante, cualquier intento de acceso a los elementos de la matriz a través de System.Array o las interfaces que implementa provocará una excepción en tiempo de ejecución, ya que los tipos de puntero no son convertibles a object.
* Las conversiones de referencia implícita y explícita (§6.1.6, §6.2.4) desde un tipo de matriz unidimensional S[] a System.Collections.Generic.IList<T> y sus interfaces base genéricas nunca se aplican a matrices de punteros, ya que los tipos de puntero no se pueden usar como argumentos de tipo y no hay conversiones desde tipos de puntero a tipos que no son puntero.
* La conversión de referencia explícita (§6.2.4) de System.Array y las interfaces que implementa en cualquier tipo de matriz (array-type) también se aplica a matrices de punteros.
* Las conversiones de referencia explícita (§6.2.4) de System.Collections.Generic.IList<S> y sus interfaces base en un tipo de matriz unidimensional T[] nunca se aplica a matrices de punteros, ya que los tipos de puntero no se pueden usar como argumentos de tipo y no hay conversiones desde tipos de puntero a tipos que no son puntero.

Estas restricciones significan que la expansión para la instrucción foreach sobre matrices descrita en §8.8.4 no se puede aplicar a matrices de punteros. En su lugar una instrucción foreach con la siguiente estructura:

foreach (V v in x) embedded-statement

donde el tipo de x es un tipo de matriz con el formato T[,,…,], n es el número de dimensiones menos 1 y T o V es un tipo de puntero, se expande usando bucles for anidados de la siguiente manera:

{  
 T[,,…,] a = x;  
 V v;  
 for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)  
 for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)  
 …  
 for (int in = a.GetLowerBound(n); in <= a.GetUpperBound(n); in++) {  
 v = (V)a.GetValue(i0,i1,…,in);  
 embedded-statement  
 }  
}

Las variables a, i0, i1, … in no son visibles ni accesibles para la expresión x o la instrucción incrustada (embedded-statement) o cualquier otro código fuente del programa. La variable v es de solo lectura en la instrucción incrustada. Si no hay una conversión explícita (§18.4) desde T (tipo de elemento) a V, se produce un error y no se realizan más pasos. Si x tiene el valor null, se inicia una excepción System.NullReferenceException en tiempo de ejecución.

## Punteros en expresiones

En un contexto no seguro, una expresión puede dar como resultado un tipo de puntero, pero fuera de ese contexto supone un error en tiempo de compilación que una expresión sea del tipo de puntero. En términos más precisos, fuera de un contexto no seguro se produce un error en tiempo de compilación si cualquier nombre simple (simple-name) (§7.6.2), acceso a miembro (member-access) (§7.6.4), expresión de invocación (invocation-expression) (§7.6.5) o acceso a elemento (element-access) (§7.6.6) es de un tipo de puntero.

En un contexto no seguro, una expresión primaria sin creación de matriz (primary-no-array-creation-expression) (§7.6) y una expresión unaria (unary-expression) (§7.7) permiten las siguientes construcciones adicionales:

primary-no-array-creation-expression:  
...  
pointer-member-access  
pointer-element-access  
sizeof-expression

unary-expression:  
...  
pointer-indirection-expression  
addressof-expression

Estas construcciones se describen en las siguientes secciones. La precedencia y asociatividad de los operadores no seguros está implicada en la gramática del lenguaje.

### Direccionamiento indirecto de punteros

Una expresión de direccionamiento indirecto de punteros (pointer-indirection-expression) se compone de un asterisco (\*) seguido de una expresión unaria (unary-expression).

pointer-indirection-expression:  
\* unary-expression

El operador unario \* indica el direccionamiento indirecto de punteros (pointer indirection) y se utiliza para obtener la variable a la que apunta un puntero. El resultado de evaluar \*P, donde P es una expresión de un tipo de puntero T\*, es una variable de tipo T. Se produce un error en tiempo de compilación al aplicar el operador unario \* a una expresión de tipo void\* o a una expresión que no sea de tipo de puntero.

El efecto de aplicar el operador unario \* a un puntero null se define por su implementación. En concreto, no hay garantía de que esta operación produzca una excepción System.NullReferenceException.

Si se asignó un valor no válido al puntero, el comportamiento del operador unario \* es indefinido. Entre los valores no válidos para desreferenciar a un puntero desde un operador unario \* se encuentran una dirección incorrectamente alineada para el tipo al que se apunta (vea el ejemplo de la §18.4) y la dirección de una variable después de finalizar su duración.

A los efectos del análisis definitivo de asignaciones, una variable producida al evaluar una expresión de la forma \*P se considera inicialmente asignada (§5.3.1).

### Acceso a miembros de puntero

Un acceso a miembros de puntero (pointer-member-access) se compone de una expresión primaria (primary-expression), seguida de un token “->”, que a su vez viene seguido de un identificador (identifier) y de una lista de argumentos de tipo (type-argument-list) opcional.

pointer-member-access:  
primary-expression -> identifier type-argument-listopt

En un acceso a miembro de puntero de la forma P->I, P debe ser una expresión de un tipo de puntero diferente de void\*, e I debe denotar un miembro accesible del tipo al que apunta P.

Un acceso a miembro de puntero de la forma P->I se evalúa exactamente como (\*P).I. Para obtener una descripción del operador de direccionamiento indirecto de puntero (\*), vea la sección §18.5.1. Para obtener una descripción del operador de acceso a miembros (.), vea la sección §7.6.4.

En el ejemplo

using System;

struct Point  
{  
 public int x;  
 public int y;

public override string ToString() {  
 return "(" + x + "," + y + ")";  
 }  
}

class Test  
{  
 static void Main() {  
 Point point;  
 unsafe {  
 Point\* p = &point;  
 p->x = 10;  
 p->y = 20;  
 Console.WriteLine(p->ToString());  
 }  
 }  
}

el operador -> se utiliza para obtener acceso a campos e invocar un método de un struct a través de un puntero. Dado que la operación P->I es precisamente equivalente a (\*P).I, podría haberse escrito igualmente el método Main:

class Test  
{  
 static void Main() {  
 Point point;  
 unsafe {  
 Point\* p = &point;  
 (\*p).x = 10;  
 (\*p).y = 20;  
 Console.WriteLine((\*p).ToString());  
 }  
 }  
}

### Acceso a elementos de puntero

Un acceso a elementos de puntero (pointer-element-access) se compone de una expresión primaria sin creación de matriz (primary-no-array-creation-expression) seguida de una expresión delimitada por “[” y “]”.

pointer-element-access:  
primary-no-array-creation-expression [ expression ]

En un acceso a elemento de puntero de la forma P[E], P debe ser una expresión de un tipo de puntero diferente de void\*, y E debe ser una expresión que pueda convertirse implícitamente a int, uint, long o ulong.

Un acceso a elemento de puntero de la forma P[E] se evalúa exactamente como \*(P + E). Para obtener una descripción del operador de direccionamiento indirecto de puntero (\*), vea la sección §18.5.1. Para obtener una descripción del operador de suma de punteros (+), vea la sección §18.5.6.

En el ejemplo

class Test  
{  
 static void Main() {  
 unsafe {  
 char\* p = stackalloc char[256];  
 for (int i = 0; i < 256; i++) p[i] = (char)i;  
 }  
 }  
}

se utiliza un acceso a elemento de puntero para inicializar el búfer de caracteres de un bucle for. Dado que la operación P[E] es precisamente equivalente a \*(P + E), el ejemplo podría haberse escrito igualmente de esta forma:

class Test  
{  
 static void Main() {  
 unsafe {  
 char\* p = stackalloc char[256];  
 for (int i = 0; i < 256; i++) \*(p + i) = (char)i;  
 }  
 }  
}

El operador de acceso a elementos de puntero no comprueba errores de tipo "fuera de límite", por otro lado, el comportamiento al obtener acceso a un elemento fuera de límites no está definido. Ocurre lo mismo en C y C++.

### Operador address-of

Una expresión de dirección (addressof-expression) se compone de un símbolo de Y comercial (&) seguido de una expresión unaria (unary-expression).

addressof-expression:  
& unary-expression

Dada una expresión E que es del tipo T y se clasifica como una variable fija (§18.3), la construcción &E calcula la dirección de la variable proporcionada por E. El tipo del resultado es T\* y se clasifica como un valor. Se produce un error en tiempo de compilación si no se clasifica E como una variable, si se clasifica E como una variable local de solo lectura o si E denota una variable móvil. En este último caso, se puede usar una instrucción fija (§18.6) para “fijar” temporalmente la variable antes de obtener su dirección. Como se afirmaba en la sección §7.6.4, fuera de un constructor de instancia o constructor estático de una clase o struct que define un campo readonly, dicho campo se considera un valor, no una variable. Como tal, no se puede capturar su dirección. Igualmente, no se puede tomar la dirección de una constante.

El operador & no requiere que su argumento esté asignado definitivamente, pero después de una operación &, la variable a la que se aplica el operador se considera definitivamente asignada en la ruta de ejecución donde tiene lugar la operación. Es responsabilidad del programador asegurarse de que se produce una correcta inicialización de la variable en esta situación.

En el ejemplo

using System;

class Test  
{  
 static void Main() {  
 int i;  
 unsafe {  
 int\* p = &i;  
 \*p = 123;  
 }  
 Console.WriteLine(i);  
 }  
}

i se considera definitivamente asignada después de la operación &i utilizada para inicializar p. La asignación a \*p tiene como resultado la inicialización de i, pero la inclusión de esta inicialización corresponde al programador, y no habrá ningún error en tiempo de compilación si se quita la asignación.

Las reglas de asignación definitiva para el operador & permiten evitar tal inicialización redundante de variables locales. Por ejemplo, muchas interfaces API externas toman un puntero a una estructura que rellena la API. Las llamadas a esas API suelen pasar la dirección de una variable de struct local, y sin esta regla, se requeriría una inicialización redundante de la variable.

### Incremento y decremento de punteros

En un contexto no seguro, los operadores ++ y ‑‑ (§7.6.9 y §7.7.5) se pueden aplicar a las variables de todos los tipos de puntero excepto void\*. Así, por cada tipo de puntero T\*, se definen de manera implícita los siguientes operadores:

T\* operator ++(T\* x);

T\* operator --(T\* x);

Los operadores generan los mismos resultados que x + 1 y x - 1, respectivamente (§18.5.6). En otras palabras, para una variable de puntero de tipo T\*, el operador ++ agrega sizeof(T) a la dirección contenida en la variable, y el operador ‑‑ resta sizeof(T) de la dirección contenida en la variable.

Si una operación de incremento o disminución de puntero desborda el dominio del tipo de puntero, se define dependiendo de la implementación, pero no se producen excepciones.

### Aritmética con punteros

En un contexto no seguro, los operadores + y - (§7.8.4 y §7.8.5) se pueden aplicar a los valores de todos los tipos de puntero excepto void\*. Así, por cada tipo de puntero T\*, se definen de manera implícita los siguientes operadores:

T\* operator +(T\* x, int y);  
T\* operator +(T\* x, uint y);  
T\* operator +(T\* x, long y);  
T\* operator +(T\* x, ulong y);

T\* operator +(int x, T\* y);  
T\* operator +(uint x, T\* y);  
T\* operator +(long x, T\* y);  
T\* operator +(ulong x, T\* y);

T\* operator –(T\* x, int y);  
T\* operator –(T\* x, uint y);  
T\* operator –(T\* x, long y);  
T\* operator –(T\* x, ulong y);

long operator –(T\* x, T\* y);

Dada una expresión P de un tipo de puntero T\* y una expresión N de tipo int, uint, long o ulong, las expresiones P + N y N + P calculan el valor de puntero de tipo T\* que resulta de agregar N \* sizeof(T) a la dirección dada por P. De igual forma, la expresión P - N calcula el valor de puntero de tipo T\* que resulta de restar N \* sizeof(T) de la dirección dada por P.

Dadas dos expresiones, P y Q, de un tipo de puntero T\*, la expresión P – Q calcula la diferencia entre las direcciones dadas por P y Q y a continuación divide la diferencia por sizeof(T). El tipo del resultado siempre es long. En efecto, P - Q se calcula como ((long)(P) - (long)(Q)) / sizeof(T).

Por ejemplo:

using System;

class Test  
{

static void Main() {  
 unsafe {  
 int\* values = stackalloc int[20];  
 int\* p = &values[1];  
 int\* q = &values[15];  
 Console.WriteLine("p - q = {0}", p - q);  
 Console.WriteLine("q - p = {0}", q - p);  
 }  
 }  
}

produce el resultado:

p - q = -14  
q - p = 14

Si una operación de aritmética de punteros desborda el dominio del tipo de puntero, se trunca el resultado dependiendo de la implementación, pero no se producen excepciones.

### Comparación de punteros

En un contexto no seguro, los operadores ==, !=, <, >, <= y => (§7.10) pueden aplicarse a valores de todos los tipos de puntero. Los operadores de comparación de punteros son:

bool operator ==(void\* x, void\* y);

bool operator !=(void\* x, void\* y);

bool operator <(void\* x, void\* y);

bool operator >(void\* x, void\* y);

bool operator <=(void\* x, void\* y);

bool operator >=(void\* x, void\* y);

Dado que existe una conversión implícita entre cualquier tipo de puntero y el tipo void\*, pueden compararse operandos de cualquier tipo de puntero mediante estos operadores. Los operadores de comparación comparan las direcciones proporcionadas por los dos operandos como si fueran enteros sin signo.

### Operador sizeof

El operador sizeof devuelve el número de bytes ocupados por una variable de un tipo dado. El tipo especificado como operando para sizeof debe ser de un tipo no administrado (unmanaged-type) (§18.2).

sizeof-expression:  
sizeof ( unmanaged-type )

El resultado del operador sizeof es un valor de tipo int. Para ciertos tipos predefinidos, el operador sizeof produce un valor constante, como se muestra en la tabla siguiente.

|  |  |
| --- | --- |
| **Expresión** | **Resultado** |
| sizeof(sbyte) | 1 |
| sizeof(byte) | 1 |
| sizeof(short) | 2 |
| sizeof(ushort) | 2 |
| sizeof(int) | 4 |
| sizeof(uint) | 4 |
| sizeof(long) | 8 |
| sizeof(ulong) | 8 |
| sizeof(char) | 2 |
| sizeof(float) | 4 |
| sizeof(double) | 8 |
| sizeof(bool) | 1 |

Para todos los demás tipos, el resultado del operador sizeof se define según la implementación y se clasifica como un valor, no como una constante.

El orden en que se empaquetan los miembros en un struct no está especificado.

Para fines de alineación, puede haber un relleno sin nombre al comienzo de un struct, dentro de un struct y al final de un struct. El contenido de los bits utilizados para el relleno es indeterminado.

Al aplicarse a un operando con tipo de struct, el resultado es el número total de bytes de una variable de ese tipo, incluidos los bytes de relleno.

## Instrucción fixed

En un contexto no seguro, la instrucción incrustada (embedded-statement) (§8) permite una construcción adicional, la instrucción fixed, que se utiliza para “fijar” una variable móvil de manera que su dirección permanece constante en toda la duración de la instrucción.

embedded-statement:  
...  
fixed-statement

fixed-statement:  
fixed ( pointer-type fixed-pointer-declarators ) embedded-statement

fixed-pointer-declarators:  
fixed-pointer-declarator  
fixed-pointer-declarators , fixed-pointer-declarator

fixed-pointer-declarator:  
identifier = fixed-pointer-initializer

fixed-pointer-initializer:  
& variable-reference  
expression

Cada declarador de puntero fijo (fixed-pointer-declarator) declara una variable local del tipo de puntero (pointer-type) dado e inicializa dicha variable con la dirección calculada por el inicializador de puntero fijo (fixed-pointer-initializer) correspondiente. Una variable local declarada en una instrucción fixed es accesible en cualquier inicializador de puntero fijo (fixed-pointer-initializer) situado a la derecha de la declaración de la variable, y en la instrucción incrustada (embedded-statement) de la instrucción fixed. Una variable local declarada por una instrucción fixed se considera de solo lectura. Se producirá un error en tiempo de compilación si la instrucción incrustada intenta modificar esta variable local (por medio de asignación o utilizando los operadores ++ y ‑‑) o bien si la pasa como un parámetro ref u out.

Un inicializador de puntero fijo (fixed-pointer-initializer) puede ser uno de los siguientes:

* El token “&” seguido de una referencia de variable (variable-reference) (§5.3.3) a una variable móvil (§18.3) de un tipo no administrado T, siempre que el tipo T\* sea implícitamente convertible al tipo de puntero proporcionado en la instrucción fixed. En este caso, el inicializador calcula la dirección de la variable dada, y se garantiza que ésta permanecerá en una dirección fija durante toda la ejecución de la instrucción fixed.
* Una expresión de un tipo de matriz (array-type) con elementos de un tipo no administrado T, siempre que el tipo T\* sea implícitamente convertible al tipo de puntero proporcionado por la instrucción fixed. En este caso, el inicializador calcula la dirección del primer elemento de la matriz, y se garantiza que toda la matriz permanecerá en una dirección fija durante toda la ejecución de la instrucción fixed. El comportamiento de la instrucción fixed se define según sea su implementación si la expresión de matriz es nula o la matriz contiene cero elementos.
* Una expresión de tipo string, siempre que el tipo char\* sea implícitamente convertible al tipo de puntero proporcionado en la instrucción fixed. En este caso, el inicializador calcula la dirección del primer carácter de la cadena, y se garantiza que toda la cadena permanecerá en una dirección fija durante toda la ejecución de la instrucción fixed. El comportamiento de la instrucción fixed se define según sea su implementación si la expresión de cadena es nula.
* Una expresión de nombre simple (simple-name) o acceso a miembro (member-access) que hace referencia a un miembro de búfer fijo de una variable móvil, siempre que el tipo del miembro de búfer fijo sea implícitamente convertible al tipo de puntero dado en la instrucción fixed. En este caso, el inicializador calcula un puntero en el primer elemento del búfer fijo (§18.7.2) y se garantiza que el búfer fijo permanecerá en una dirección fija para la duración de la instrucción fixed.

Por cada dirección calculada por un inicializador de puntero fijo (fixed-pointer-initializer), la instrucción fixed garantiza que la variable a la que hace referencia la dirección no está sujeta a reubicación o eliminación por el recolector de elementos no utilizados en toda la duración de la instrucción fixed. Por ejemplo, si la dirección calculada por un inicializador de puntero fijo (fixed-pointer-initializer) hace referencia a un campo de un objeto o un elemento de una instancia de matriz, la instrucción fixed garantiza que la instancia de objeto contenedora no se reubicará ni eliminará en toda la duración de la instrucción.

Es responsabilidad del programador asegurarse de que los punteros creados mediante instrucciones fixed no permanecen tras la ejecución de dichas instrucciones. Por ejemplo, cuando se pasan punteros creados con instrucciones fixed a interfaces API externas, es responsabilidad del programador asegurarse de que las API no retienen memoria de estos punteros.

Los objetos fijos pueden provocar la fragmentación del montón al no poder moverse. Por este motivo, los objetos deben ser fijos sólo cuando sea absolutamente necesario y por el menor tiempo posible.

En el ejemplo

class Test  
{  
 static int x;  
 int y;

unsafe static void F(int\* p) {  
 \*p = 1;  
 }

static void Main() {  
 Test t = new Test();  
 int[] a = new int[10];  
 unsafe {  
 fixed (int\* p = &x) F(p);  
 fixed (int\* p = &t.y) F(p);  
 fixed (int\* p = &a[0]) F(p);  
 fixed (int\* p = a) F(p);  
 }  
 }  
}

se muestran varios usos de la instrucción fixed. La primera instrucción fija y obtiene la dirección de un campo estático, la segunda fija y obtiene la dirección de un campo de instancia, y la tercera fija y obtiene la dirección de un elemento de matriz. En cada caso hubiera sido un error utilizar el operador & habitual, ya que las variables se clasifican todas como variables móviles.

La cuarta instrucción fixed del ejemplo anterior produce idénticos resultados que la tercera.

En este ejemplo de la instrucción fixed se usa string:

class Test  
{  
 static string name = "xx";

unsafe static void F(char\* p) {  
 for (int i = 0; p[i] != '\0'; ++i)  
 Console.WriteLine(p[i]);  
 }

static void Main() {  
 unsafe {  
 fixed (char\* p = name) F(p);  
 fixed (char\* p = "xx") F(p);  
 }  
 }  
}

En una matriz de contexto no seguro, los elementos de matrices unidimensionales se almacenan por orden de índice creciente, comenzando con el índice 0 y terminando con el índice Length – 1. Los elementos de matrices multidimensionales se almacenan de manera que los índices de la dimensión del extremo derecho se incrementan en primer lugar, a continuación la dimensión situada inmediatamente a su izquierda, etc., siguiendo por la izquierda. Dentro de una instrucción fixed que obtiene un puntero p para una instancia de matriz a, los valores de puntero comprendidos entre p y p + a.Length - 1 representan direcciones de los elementos de la matriz. De igual forma, las variables comprendidas en el intervalo de p[0] a p[a.Length - 1] representan los propios elementos de la matriz. Dada la forma en que se almacenan las matrices, se puede tratar una matriz de cualquier dimensión como si fuera lineal.

Por ejemplo:

using System;

class Test  
{  
 static void Main() {  
 int[,,] a = new int[2,3,4];  
 unsafe {  
 fixed (int\* p = a) {  
 for (int i = 0; i < a.Length; ++i) // treat as linear  
 p[i] = i;  
 }  
 }

for (int i = 0; i < 2; ++i)  
 for (int j = 0; j < 3; ++j) {  
 for (int k = 0; k < 4; ++k)  
 Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k, a[i,j,k]);  
 Console.WriteLine();  
 }  
 }  
}

produce el resultado:

[0,0,0] = 0 [0,0,1] = 1 [0,0,2] = 2 [0,0,3] = 3  
[0,1,0] = 4 [0,1,1] = 5 [0,1,2] = 6 [0,1,3] = 7  
[0,2,0] = 8 [0,2,1] = 9 [0,2,2] = 10 [0,2,3] = 11  
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15  
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19  
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23

En el ejemplo

class Test  
{  
 unsafe static void Fill(int\* p, int count, int value) {  
 for (; count != 0; count--) \*p++ = value;  
 }

static void Main() {  
 int[] a = new int[100];  
 unsafe {  
 fixed (int\* p = a) Fill(p, 100, -1);  
 }  
 }  
}

se utiliza una instrucción fixed para fijar una matriz de manera que pueda pasarse su dirección a un método que toma un puntero.

En el siguiente ejemplo:

unsafe struct Font  
{  
 public int size;  
 public fixed char name[32];  
}

class Test  
{  
 unsafe static void PutString(string s, char\* buffer, int bufSize) {  
 int len = s.Length;  
 if (len > bufSize) len = bufSize;  
 for (int i = 0; i < len; i++) buffer[i] = s[i];  
 for (int i = len; i < bufSize; i++) buffer[i] = (char)0;  
 }

Font f;

unsafe static void Main()  
 {  
 Test test = new Test();  
 test.f.size = 10;  
 fixed (char\* p = test.f.name) {  
 PutString("Times New Roman", p, 32);  
 }  
 }  
}

se utiliza una instrucción fixed para fijar un búfer fijo de un struct de manera que pueda utilizarse su dirección como un puntero.

Un valor char\* producido al fijar una instancia de cadena siempre apunta a una cadena terminada en NULL Dentro de una instrucción fixed que obtiene un puntero p a una instancia de cadena s, los valores de puntero comprendidos entre p y p + s.Length - 1 representan direcciones de los caracteres de la cadena, y el valor de puntero p + s.Length siempre apunta a un carácter nulo (el carácter con valor '\0').

La modificación de objetos de tipo administrado mediante punteros fijos puede producir un comportamiento indefinido. Por ejemplo, dado que las cadenas son invariables, es responsabilidad del programador asegurarse de que los caracteres de una cadena fija a los que hace referencia un puntero no se modifiquen.

La terminación automática en null de cadenas es especialmente cómoda al llamar a interfaces API externas que esperan cadenas “de estilo C”. Observe, sin embargo, que no se permite una instancia de cadena que contenga caracteres null. Si se detectan dichos caracteres nulos, la cadena aparecerá truncada al tratarla como char\* terminada en NULL.

## Búferes fijos

Los búferes fijos se utilizan para declarar matrices en línea “de estilo C” como miembros de structs, y son útiles principalmente para interactuar con API no administradas.

### Declaraciones de búferes fijos

Un búfer fijo es un miembro que representa el almacenamiento para un búfer de longitud fija de variables de un tipo determinado. Una declaración de búfer fijo introduce uno o más búferes fijos de un tipo de elemento dado. Los búferes fijos sólo se permiten en declaraciones de structs y sólo pueden darse en contextos no seguros (§18.1).

struct-member-declaration:  
…  
fixed-size-buffer-declaration

fixed-size-buffer-declaration:  
attributesopt fixed-size-buffer-modifiersopt fixed buffer-element-type  
 fixed-size-buffer-declarators ;

fixed-size-buffer-modifiers:  
fixed-size-buffer-modifier  
fixed-size-buffer-modifier fixed-size-buffer-modifiers

fixed-size-buffer-modifier:  
new  
public  
protected  
internal  
private  
unsafe

buffer-element-type:  
type

fixed-size-buffer-declarators:  
fixed-size-buffer-declarator  
fixed-size-buffer-declarator , fixed-size-buffer-declarators

fixed-size-buffer-declarator:  
identifier [ constant-expression ]

Es posible que una declaración de búfer fijo incluya un conjunto de atributos (§17), un modificador new (§10.2.2), una combinación válida de los cuatro modificadores de acceso (§10.2.3) y un modificador unsafe (§18.1). Los atributos y modificadores se aplican a todos los miembros declarados por la declaración de búfer fijo. Es un error que el mismo modificador aparezca varias veces en una declaración de búfer fijo.

Una declaración de búfer fijo no puede incluir el modificador static.

El tipo de elemento de búfer de una declaración de búfer fijo especifica el tipo de elemento del búfer o de los búferes introducido por la declaración. El tipo de elemento de búfer debe ser uno de los tipos predefinidos sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double o bool.

El tipo de elemento de búfer viene seguido de una lista de declaradores de búfer fijo, cada uno de los cuales introduce un nuevo miembro. Un declarador de búfer fijo está formado por un identificador que denomina al miembro, seguido de una expresión constante entre los tokens [ y ]. La expresión constante denota el número de elementos del miembro introducido por el declarador de búfer fijo. El tipo de la expresión constante se debe convertir implícitamente en el tipo int y el valor debe ser un entero positivo distinto de cero.

Se garantiza que los elementos de un búfer fijo se organicen de manera secuencial en memoria.

Una declaración de búfer fijo que declara varios búferes fijos equivale a varias declaraciones de una declaración de búfer fijo con los mismos atributos y tipos de elementos. Por ejemplo:

unsafe struct A  
{  
 public fixed int x[5], y[10], z[100];  
}

equivale a

unsafe struct A  
{  
 public fixed int x[5];  
 public fixed int y[10];  
 public fixed int z[100];  
}

### Búferes fijos en expresiones

La búsqueda de miembros (§7.3) de un miembro de búfer fijo procede exactamente de la misma manera que la búsqueda de miembros de un campo.

Se puede hacer referencia a un búfer fijo en una expresión con un nombre simple (simple-name) (§7.5.2) o un acceso a miembro (member-access) (§7.5.4).

Cuando se hace referencia a un miembro de búfer fijo como un nombre simple, el efecto es el mismo que un acceso a miembros con la estructura this.I, donde I es el miembro de búfer fijo.

En un acceso a miembro de la estructura E.I, si E es de un tipo struct y una búsqueda de miembros de I en dicho tipo struct identifica un miembro fijo, E.I se evalúa y clasifica de la siguiente manera:

* Si la expresión E.I no ocurre en un contexto no seguro, se genera un error en tiempo de compilación.
* Si E se clasifica como un valor, se genera un error de compilación.
* De lo contrario, si E es una variable móvil (§18.3) y la expresión E.I no es un inicializador de puntero fijo (fixed-pointer-initializer) (§18.6), se genera un error en tiempo de compilación.
* De lo contrario, E hace referencia a una variable fija y el resultado de la expresión es un puntero al primer elemento del miembro de búfer fijo I en E. El resultado es del tipo S\*, donde S es el tipo de elemento de I y se clasifica como un valor.

A los elementos subsiguientes del búfer fijo se puede obtener acceso con operaciones de puntero desde el primer elemento. Al contrario que ocurre con el acceso a matrices, el acceso a los elementos de un búfer fijo es una operación no segura y su intervalo no se comprueba.

En el siguiente ejemplo se declara y utiliza un struct con un miembro de búfer fijo.

unsafe struct Font  
{  
 public int size;  
 public fixed char name[32];  
}

class Test  
{  
 unsafe static void PutString(string s, char\* buffer, int bufSize) {  
 int len = s.Length;  
 if (len > bufSize) len = bufSize;  
 for (int i = 0; i < len; i++) buffer[i] = s[i];  
 for (int i = len; i < bufSize; i++) buffer[i] = (char)0;  
 }

unsafe static void Main()  
 {  
 Font f;  
 f.size = 10;  
 PutString("Times New Roman", f.name, 32);  
 }  
}

### Comprobación de asignación definitiva

El búfer fijo no está sujeto a una comprobación de asignación definitiva (§5.3) y los miembros de búfer fijo se omiten con el fin de comprobar la asignación definitiva de variables de tipo struct.

Cuando la variable de struct más externa de un miembro de búfer fijo es una variable estática, una variable de instancia de una instancia de clase o un elemento de array, los elementos de búfer fijo se inicializan automáticamente a sus valores predeterminados (§5.2). En el resto de casos, el contenido inicial de un búfer fijo no está definido.

## Asignación de pila

En un contexto no seguro, una declaración de variable local (§8.5.1) puede incluir un inicializador de asignación de pila que asigne memoria de la pila de llamadas.

local-variable-initializer:  
…  
stackalloc-initializer

stackalloc-initializer:  
stackalloc unmanaged-type [ expression ]

El tipo no administrado (unmanaged-type) indica el tipo de los elementos que se almacenarán en la ubicación recién asignada, y la expresión (expression) indica el número de estos elementos. Tomados juntos, especifican el tamaño de asignación requerido. Dado que el tamaño de una asignación de pila no puede ser negativo, supone un error en tiempo de compilación especificar el número de elementos como una expresión constante (constant-expression) que se evalúa como un número negativo.

Un inicializador de asignación de pila de la forma stackalloc T[E] requiere que T sea un tipo no administrado (§18.2) y E sea una expresión de tipo int. La construcción asigna E \* sizeof(T) bytes de la pila de llamadas y devuelve un puntero de tipo T\* al bloque recién asignado. Si E es un valor negativo, el comportamiento es indefinido. Si E es cero, no se realiza ninguna asignación, y el puntero devuelto se define según su implementación. Si no hay suficiente memoria disponible para asignar un bloque del tamaño dado, se produce una excepción System.StackOverflowException.

El contenido de la memoria recién asignada es indefinido.

No se permiten los inicializadores de asignación de pila en bloques catch o finally (§8.10).

No hay forma de liberar explícitamente la memoria asignada con stackalloc. Todos los bloques asignados de memoria de la pila y creados durante la ejecución de un miembro de función se descartan automáticamente cuando éste devuelve un valor. Esto se corresponde con la función alloca, una extensión habitual en las implementaciones de C y C++.

En el ejemplo

using System;

class Test  
{  
 static string IntToString(int value) {  
 int n = value >= 0? value: -value;  
 unsafe {  
 char\* buffer = stackalloc char[16];  
 char\* p = buffer + 16;  
 do {  
 \*--p = (char)(n % 10 + '0');  
 n /= 10;  
 } while (n != 0);  
 if (value < 0) \*--p = '-';  
 return new string(p, 0, (int)(buffer + 16 - p));  
 }  
 }

static void Main() {  
 Console.WriteLine(IntToString(12345));  
 Console.WriteLine(IntToString(-999));  
 }  
}

un inicializador stackalloc se utiliza en el método IntToString para asignar un búfer de 16 caracteres en la pila. El búfer queda descartado automáticamente cuando el método devuelve un valor.

## Asignación dinámica de memoria

Salvo por el operador stackalloc, C# no proporciona construcciones predefinidas para administrar memoria no recolectada por el recolector de elementos no utilizados. Tales servicios suelen proporcionarlos bibliotecas de clases auxiliares o importadas directamente del sistema operativo subyacente. Por ejemplo, la clase Memory de abajo muestra cómo es posible obtener acceso desde C# a las funciones del montón de un sistema operativo subyacente:

using System;  
using System.Runtime.InteropServices;

public unsafe class Memory  
{  
 // Handle for the process heap. This handle is used in all calls to the  
 // HeapXXX APIs in the methods below.

static int ph = GetProcessHeap();

// Private instance constructor to prevent instantiation.

private Memory() {}

// Allocates a memory block of the given size. The allocated memory is  
 // automatically initialized to zero.

public static void\* Alloc(int size) {  
 void\* result = HeapAlloc(ph, HEAP\_ZERO\_MEMORY, size);  
 if (result == null) throw new OutOfMemoryException();  
 return result;  
 }

// Copies count bytes from src to dst. The source and destination  
 // blocks are permitted to overlap.

public static void Copy(void\* src, void\* dst, int count) {  
 byte\* ps = (byte\*)src;  
 byte\* pd = (byte\*)dst;  
 if (ps > pd) {  
 for (; count != 0; count--) \*pd++ = \*ps++;  
 }  
 else if (ps < pd) {  
 for (ps += count, pd += count; count != 0; count--) \*--pd = \*--ps;  
 }  
 }

// Frees a memory block.

public static void Free(void\* block) {  
 if (!HeapFree(ph, 0, block)) throw new InvalidOperationException();  
 }

// Re-allocates a memory block. If the reallocation request is for a  
 // larger size, the additional region of memory is automatically  
 // initialized to zero.

public static void\* ReAlloc(void\* block, int size) {  
 void\* result = HeapReAlloc(ph, HEAP\_ZERO\_MEMORY, block, size);  
 if (result == null) throw new OutOfMemoryException();  
 return result;  
 }

// Returns the size of a memory block.

public static int SizeOf(void\* block) {  
 int result = HeapSize(ph, 0, block);  
 if (result == -1) throw new InvalidOperationException();  
 return result;  
 }

// Heap API flags

const int HEAP\_ZERO\_MEMORY = 0x00000008;

// Heap API functions

[DllImport("kernel32")]  
 static extern int GetProcessHeap();

[DllImport("kernel32")]  
 static extern void\* HeapAlloc(int hHeap, int flags, int size);

[DllImport("kernel32")]  
 static extern bool HeapFree(int hHeap, int flags, void\* block);

[DllImport("kernel32")]  
 static extern void\* HeapReAlloc(int hHeap, int flags,  
 void\* block, int size);

[DllImport("kernel32")]  
 static extern int HeapSize(int hHeap, int flags, void\* block);  
}

A continuación se muestra un ejemplo que utiliza la clase Memory:

class Test  
{  
 static void Main() {  
 unsafe {  
 byte\* buffer = (byte\*)Memory.Alloc(256);  
 try {  
 for (int i = 0; i < 256; i++) buffer[i] = (byte)i;  
 byte[] array = new byte[256];  
 fixed (byte\* p = array) Memory.Copy(buffer, p, 256);   
 }  
 finally {  
 Memory.Free(buffer);  
 }  
 for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);  
 }  
 }  
}

En el ejemplo se asignan 256 bytes de memoria mediante Memory.Alloc y se inicializa el bloque de memoria con valores crecientes de 0 a 255. A continuación, se asigna una matriz de bytes de 256 elementos y se utiliza Memory.Copy para copiar el contenido del bloque de memoria en la matriz de bytes. Por último, se libera el bloque de memoria mediante Memory.Free y se envía el contenido de la matriz de bytes como resultado a la consola.

1. Comentarios de la documentación

C# proporciona un mecanismo para que los programadores puedan documentar el código mediante una sintaxis de comentarios que contiene texto XML. En los archivos de código fuente, es posible utilizar comentarios con cierto formato para indicar a una herramienta que genere texto XML a partir de estos comentarios y de los elementos de código fuente a los que precede. Los comentarios que utilizan esta sintaxis se denominan comentarios de documentación. Deben preceder inmediatamente a un tipo definido por el usuario (como una clase, un delegado o una interfaz) o a un miembro (como un campo, un evento, una propiedad o un método). La herramienta de generación XML se denomina generador de documentación. \t "See documentation comment" (Este generador podría ser, pero no es necesario que sea el propio compilador de C#). El resultado producido por el generador de documentación se denomina archivo de documentación. \t "See documentation comment" \b Un archivo de documentación se usa como entrada para un visor de documentación; \t "See documentation comment" \b \b una herramienta pensada para producir algún tipo de información y su documentación asociada.

Esta especificación sugiere el uso de un conjunto de etiquetas en los comentarios de esta clase de documentación, sin embargo, siempre que se respete la correcta sintaxis del lenguaje XML, se puede utilizar el conjunto de etiquetas que se desee para este fin.

* 1. Introducción

Es posible utilizar comentarios con un formato determinado para indicar a una herramienta que genere texto XML a partir de los mismos y de los elementos de código fuente a los que preceden. Tales comentarios pueden ser: comentarios de una sola línea que comienzan con tres barras diagonales (///), o comentarios delimitados que van precedidos de una barra diagonal y dos asteriscos (/\*\*). Deben preceder inmediatamente al tipo definido por el usuario (como una clase, un delegado o una interfaz) o al miembro (como un campo, un evento, una propiedad o un método) al que hacen referencia los comentarios. Las secciones de atributos (§17.2) se consideran parte de las declaraciones; por lo tanto, los comentarios de documentación deben preceder a los atributos que se aplican a un tipo o a un miembro.

**Sintaxis:**

single-line-doc-comment:  
/// input-charactersopt

delimited-doc-comment:  
/\*\* delimited-comment-textopt \*/

En un comentario de una sola línea (single-line-doc-comment), si aparece un carácter de espacio en blanco (whitespace) a continuación de los caracteres /// en cada uno de los comentarios de documentación de una sola línea (single-line-doc-comments) adyacentes al comentario de documentación de una sola línea (single-line-doc-comment) actual, el carácter de espacio en blanco (whitespace) no se incluirá en los resultados XML.

En un comentario de documentación delimitado (delimited-doc-comment), si el primer carácter distinto del espacio en blanco (whitespace) es un asterisco (asterisk ) y el mismo patrón de caracteres opcionales de espacio en blanco y un carácter de asterisco se repite al principio de cada una de las líneas dentro del comentario de documentación delimitado (delimited-doc-comment), entonces los caracteres del patrón repetido no se incluyen en el código XML resultante. El patrón puede incluir caracteres de espacio en blanco (whitespace) antes y después del carácter de asterisco (asterisk ).

**Ejemplo:**

/// <summary>Class <c>Point</c> models a point in a two-dimensional  
/// plane.</summary>  
///  
public class Point   
{  
 /// <summary>method <c>draw</c> renders the point.</summary>  
 void draw() {…}  
}

El texto de los comentarios de documentación debe estar correctamente estructurado según las reglas del lenguaje XML (http://www.w3.org/TR/REC-xml). Si hay incorrecciones en el texto XML, se generará una advertencia y el archivo de documentación incluirá un comentario que hará referencia al error encontrado.

Aunque los desarrolladores pueden crear su propio juego de etiquetas, existe un juego recomendado en la sección §A.2. Algunas de las etiquetas recomendadas tienen significados especiales:

* La etiqueta <param> se utiliza para describir parámetros. Si se utiliza dicha etiqueta, el generador de documentación deberá comprobar que el parámetro especificado existe y que todos los parámetros se encuentran descritos en los comentarios de documentación. Si esta comprobación encuentra algún error, el generador de documentación emitirá una advertencia.
* El atributo cref \b se puede asociar a cualquier etiqueta para proporcionar una referencia a un elemento de código. El generador de documentación debe comprobar si existe dicho elemento de código. Si la comprobación no tiene éxito, el generador de documentación emite una advertencia. Al buscar un nombre descrito en un atributo cref, el generador de la documentación debe respetar la visibilidad de espacio de nombres según las instrucciones using que aparecen dentro del código fuente. Para los elementos de código que son genéricos, la sintaxis genérica normal (es decir, “List<T>”) no se puede usar porque produce XML no válido. Se pueden usar llaves en lugar de corchetes (es decir, “List{T}”), o se puede usar la sintaxis de escape XML (es decir, “List&lt;T&gt;”).
* La etiqueta <summary> está pensada para su uso en un visor de documentación con el fin de mostrar información adicional sobre un tipo o un miembro.
* La etiqueta <include> incluye información de un archivo XML externo.

Observe que el archivo de documentación no proporciona información completa sobre el tipo y los miembros (por ejemplo, no contiene ninguna información de tipos). Para obtener tal información acerca de un tipo o un miembro, el archivo de documentación debe utilizarse conjuntamente con el mecanismo de reflexión sobre el tipo o el miembro real.

* 1. Etiquetas recomendadas

El generador de documentación debe aceptar y procesar todas aquellas etiquetas que sean válidas según las reglas de XML. Las siguientes etiquetas proporcionan la funcionalidad habitual en la documentación de usuario. (Por supuesto, se pueden utilizar otras etiquetas).

|  |  |  |
| --- | --- | --- |
| **Etiqueta** | **Sección** | **Finalidad** |
| <c> | A.2.1 | Establecer un tipo de fuente de código para un texto |
| <code> | A.2.2 | Establecer una o más líneas de código fuente o indicar el final de un programa |
| <example> | A.2.3 | Indicar un ejemplo |
| <exception> | A.2.4 | Identifica las excepciones que puede producir un método |
| <include> | A.2.5 | Incluye XML procedente de un archivo externo |
| <list> | A.2.6 | Crear una lista o una tabla |
| <para> | A.2.7 | Permite agregar una estructura al texto |
| <param> | A.2.8 | Describe un parámetro para un método o constructor |
| <paramref> | A.2.9 | Identifica una palabra como nombre de parámetro |
| <permission> | A.2.10 | Documenta la accesibilidad de seguridad de un miembro |
| <remark> | A.2.11 | Describe información adicional acerca de un tipo |
| <returns> | A.2.12 | Describe el valor devuelto de un método |
| <see> | A.2.13 | Especifica un vínculo |
| <seealso> | A.2.14 | Genera una entrada de tipo Vea también |
| <summary> | A.2.15 | Describe un tipo o un miembro de un tipo |
| <value> | A.2.16 | Describe una propiedad |
| <typeparam> |  | Describe un parámetro de tipo genérico |
| <typeparamref> |  | Identifica una palabra como nombre de parámetro de tipo |

* + 1. <c> \t "See <c>" \b

Esta etiqueta proporciona un mecanismo para indicar que se debe establecer una fuente especial para un fragmento de texto dentro de una descripción, tal como la que se usa para un bloque de código. (Para líneas de código real, use <code> (§A.2.2).

**Sintaxis:**

<c>*text*</c>

**Ejemplo:**

/// <summary>Class <c>Point</c> models a point in a two-dimensional  
/// plane.</summary>

public class Point   
{  
 // ...  
}

* + 1. <code> \t "See <code>" \b

Esta etiqueta se utiliza para establecer una o más líneas de código fuente o para indicar el final de un programa en alguna fuente especial. Para pequeños fragmentos de código en texto narrativo, utilice <c> (§A.2.1).

**Sintaxis:**

<code>*source code or program output*</code>

**Ejemplo:**

/// <summary>This method changes the point's location by  
/// the given x- and y-offsets.  
/// <example>For example:  
/// <code>  
/// Point p = new Point(3,5);  
/// p.Translate(-1,3);  
/// </code>  
/// results in <c>p</c>'s having the value (2,8).  
/// </example>  
/// </summary>

public void Translate(int xor, int yor) {  
 X += xor;  
 Y += yor;  
}

* + 1. <example> \t "See <example>" \b

Esta etiqueta permite código de ejemplo dentro de un comentario, para especificar cómo puede utilizarse un método u otro miembro de biblioteca. Generalmente, esto también supone la utilización de la etiqueta <code> (§A.2.2).

**Sintaxis:**

<example>*description*</example>

**Ejemplo:**

Vea <code> (§A.2.2) para obtener un ejemplo.

* + 1. <exception> \t "See <exception>" \b

Esta etiqueta proporciona un medio para documentar las excepciones que puede producir un método.

**Sintaxis:**

<exception cref="*member*">*description*</exception>

donde

cref="*member*"

El nombre de un miembro. El generador de documentación comprueba si el miembro en cuestión existe y traduce member al nombre del elemento canónico en el archivo de documentación.

*description*

Descripción de las circunstancias en las que se inicia la excepción.

**Ejemplo:**

public class DataBaseOperations  
{  
 /// <exception cref="MasterFileFormatCorruptException"></exception>  
 /// <exception cref="MasterFileLockedOpenException"></exception>  
 public static void ReadRecord(int flag) {  
 if (flag == 1)  
 throw new MasterFileFormatCorruptException();  
 else if (flag == 2)  
 throw new MasterFileLockedOpenException();  
 // …  
 }   
}

* + 1. <include>

Esta etiqueta permite incluir información de un documento XML externo al archivo de código fuente. El archivo externo debe ser un documento XML sin errores; al documento se aplica una expresión XPath para especificar qué parte de la sintaxis XML del documento debe incluirse. La etiqueta <include> se sustituye entonces por el texto XML seleccionado del documento externo.

**Sintaxis:**

<include file="filename" path="xpath" />

donde

file="filename"

Nombre de un archivo XML externo. El nombre del archivo se interpreta en relación con el archivo que contiene la etiqueta include.

path="xpath"

Una expresión XPath que selecciona parte del código XML del archivo XML externo.

**Ejemplo:**

Si el código fuente contuviera una declaración como la siguiente:

/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/\*' />  
public class IntList { … }

y el archivo externo “docs.xml” tuviera el siguiente contenido:

<?xml version="1.0"?>  
<extradoc>  
 <class name="IntList">  
 <summary>  
 Contains a list of integers.  
 </summary>  
 </class>  
 <class name="StringList">  
 <summary>  
 Contains a list of integers.  
 </summary>  
 </class>  
</extradoc>

entonces la documentación resultante sería la misma que si el código fuente contuviera:

/// <summary>  
/// Contains a list of integers.  
/// </summary>  
public class IntList { … }

* + 1. <list> \t "See <list>" \b

Esta etiqueta se utiliza para crear una lista o tabla de elementos. Puede contener un bloque <listheader> para definir la fila de encabezado de una tabla o de una lista de definiciones. (Cuando se define una tabla, solo es necesario suministrar una entrada para *term* en el encabezado.)

Cada elemento de la lista se especifica con un bloque <item>. Al crear una lista de definiciones, es necesario especificar tanto *term* como *description*. Sin embargo, para una tabla, lista con viñetas o lista numerada, solo es necesario especificar *description*.

**Sintaxis:**

<list type="bullet" | "number" | "table">  
 <listheader>  
 <term>*term*</term>  
 <description>*description*</description>  
 </listheader>  
 <item>  
 <term>*term*</term>  
 <description>*description*</description>  
 </item>  
 …  
 <item>  
 <term>*term*</term>  
 <description>*description*</description>  
 </item>  
</list>

donde

*term*

El término que se desea definir, cuya definición se encuentra en *description*.

*description*

Puede ser el elemento de una lista numerada o con viñetas, o la definición de un *term*.

**Ejemplo:**

public class MyClass  
{  
 /// <summary>Here is an example of a bulleted list:  
 /// <list type="bullet">  
 /// <item>  
 /// <description>Item 1.</description>  
 /// </item>  
 /// <item>  
 /// <description>Item 2.</description>  
 /// </item>  
 /// </list>  
 /// </summary>  
 public static void Main () {  
 // ...  
 }  
}

* + 1. <para> \t "See <para>" \b

Este etiqueta debe utilizarse dentro de otras, como <summary> (§A.2.11) o <returns> (§A.2.12) y permite agregar la estructura al texto.

**Sintaxis:**

<para>*content*</para>

donde

*content*

Texto del párrafo.

**Ejemplo:**

/// <summary>This is the entry point of the Point class testing program.  
/// <para>This program tests each method and operator, and  
/// is intended to be run after any non-trvial maintenance has  
/// been performed on the Point class.</para></summary>  
public static void Main() {  
 // ...  
}

* + 1. <param> \t "See <param>" \b

Esta etiqueta se utiliza para describir un parámetro para un método, constructor o indizador.

**Sintaxis:**

<param name="*name*">*description*</param>

donde

*name*

Nombre del parámetro.

*description*

Una descripción del parámetro.

**Ejemplo:**

/// <summary>This method changes the point's location to  
/// the given coordinates.</summary>  
/// <param name="xor">the new x-coordinate.</param>  
/// <param name="yor">the new y-coordinate.</param>  
public void Move(int xor, int yor) {  
 X = xor;  
 Y = yor;  
}

* + 1. <paramref> \t "See <paramref>" \b

Esta etiqueta se utiliza para indicar que una palabra es un parámetro. El archivo de documentación se puede procesar de manera que aplique un formato diferente a este parámetro.

**Sintaxis:**

<paramref name="*name*"/>

donde

*name*

Nombre del parámetro.

**Ejemplo:**

/// <summary>This constructor initializes the new Point to  
/// (<paramref name="xor"/>,<paramref name="yor"/>).</summary>  
/// <param name="xor">the new Point's x-coordinate.</param>  
/// <param name="yor">the new Point's y-coordinate.</param>

public Point(int xor, int yor) {  
 X = xor;  
 Y = yor;  
}

* + 1. <permission> \t "See <permission>" \b

Esta etiqueta permite documentar la accesibilidad de seguridad de un miembro.

**Sintaxis:**

<permission cref="*member*">*description*</permission>

donde

cref="*member*"

El nombre de un miembro. El generador de documentación comprueba si el elemento de código dado existe y traduce member al nombre del elemento canónico en el archivo de documentación.

*description*

Descripción del acceso al miembro.

**Ejemplo:**

/// <permission cref="System.Security.PermissionSet">Everyone can  
/// access this method.</permission>

public static void Test() {  
 // ...  
}

* + 1. <remark> \t "See <remarks>" \b

Esta etiqueta se utiliza para especificar información adicional sobre un tipo. (Use <summary> (§A.2.15) para describir el propio tipo y los miembros de un tipo.)

**Sintaxis:**

<remark>*description*</remark>

donde

*description*

El texto de la nota.

**Ejemplo:**

/// <summary>Class <c>Point</c> models a point in a   
/// two-dimensional plane.</summary>  
/// <remark>Uses polar coordinates</remark>  
public class Point   
{  
 // ...  
}

* + 1. <returns> \t "See <returns>" \b

Esta etiqueta se utiliza para describir el valor devuelto de un método.

**Sintaxis:**

<returns>*description*</returns>

donde

*description*

Descripción del valor devuelto.

**Ejemplo:**

/// <summary>Report a point's location as a string.</summary>  
/// <returns>A string representing a point's location, in the form (x,y),  
/// without any leading, trailing, or embedded whitespace.</returns>  
public override string ToString() {  
 return "(" + X + "," + Y + ")";  
}

* + 1. <see> \t "See <see>" \b

Esta etiqueta permite especificar un vínculo dentro del texto. Utilice <seealso> (§A.2.14) para indicar el texto que debe aparecer en una sección Vea también.

**Sintaxis:**

<see cref="*member*"/>

donde

cref="*member*"

El nombre de un miembro. El generador de documentación comprueba si el elemento de código dado existe y cambia member por el nombre del elemento en el archivo de documentación generado.

**Ejemplo:**

/// <summary>This method changes the point's location to  
/// the given coordinates.</summary>  
/// <see cref="Translate"/>  
public void Move(int xor, int yor) {  
 X = xor;  
 Y = yor;  
}

/// <summary>This method changes the point's location by  
/// the given x- and y-offsets.  
/// </summary>  
/// <see cref="Move"/>  
public void Translate(int xor, int yor) {  
 X += xor;  
 Y += yor;  
}

* + 1. <seealso> \t "See <seealso>" \b

Esta etiqueta permite generar una entrada para la sección Vea también. Utilice <see> (§A.2.13) para especificar un vínculo desde dentro del texto.

**Sintaxis:**

<seealso cref="*member*"/>

donde

cref="*member*"

El nombre de un miembro. El generador de documentación comprueba si el elemento de código dado existe y cambia member por el nombre del elemento en el archivo de documentación generado.

**Ejemplo:**

/// <summary>This method determines whether two Points have the same  
/// location.</summary>  
/// <seealso cref="operator=="/>  
/// <seealso cref="operator!="/>  
public override bool Equals(object o) {  
 // ...  
}

* + 1. <summary> \t "See <summary>" \b

Esta etiqueta se utiliza para describir un tipo o un miembro de un tipo. Utilice <remark> (§A.2.11) para describir el propio tipo.

**Sintaxis:**

<summary>*description*</summary>

donde

*description*

Resumen del tipo o miembro.

**Ejemplo:**

/// <summary>This constructor initializes the new Point to (0,0).</summary>  
public Point() : this(0,0) {  
}

* + 1. <value> \t "See <value>" \b

Esta etiqueta permite describir una propiedad.

**Sintaxis:**

<value>*property* *description*</value>

donde

*property description*

Descripción de la propiedad.

**Ejemplo:**

/// <value>Property <c>X</c> represents the point's x-coordinate.</value>  
public int X  
{  
 get { return x; }  
 set { x = value; }  
}

* + 1. <typeparam>

Esta etiqueta se utiliza para describir un parámetro de tipo genérico para un método, delegado, interfaz, struct o clase.

**Sintaxis:**

<typeparam name="*name*">*description*</typeparam>

donde

*name*

Nombre del parámetro de tipo.

*description*

Descripción del parámetro de tipo.

**Ejemplo:**

/// <summary>A generic list class.</summary>  
/// <typeparam name="T">The type stored by the list.</typeparam>  
public class MyList<T> {  
 ...  
}

* + 1. <typeparamref>

Esta etiqueta se utiliza para indicar que una palabra es un parámetro de tipo. El archivo de documentación se puede procesar de manera que aplique un formato diferente a este parámetro de tipo.

**Sintaxis:**

<typeparamref name="*name*"/>

donde

*name*

Nombre del parámetro de tipo.

**Ejemplo:**

/// <summary>This method fetches data and returns a list of <typeparamref name=”T”> ”/>”> .</summary>  
/// <param name="string">query to execute</param>  
  
public List<T> FetchData<T>(string query) {  
 ...  
}

* 1. Procesamiento del archivo de documentación

El generador de documentación genera una cadena de id. para cada elemento del código fuente marcado con un comentario de documentación. Esta cadena de id. identifica de forma única un elemento fuente. Un visor de documentación puede utilizar una cadena de id. para identificar el correspondiente elemento de metadatos/reflexión al que se aplica la documentación.

El archivo de documentación no es una representación jerárquica del código fuente, es una lista sencilla con una cadena de id. generada para cada elemento.

* + 1. Formato de cadena de Id.

El generador de documentación cumple las siguientes reglas cuando genera las cadenas de id.:

* No se coloca ningún espacio en blanco en la cadena.
* La primera parte de la cadena identifica el tipo de miembro que se desea documentar por medio de un único carácter seguido de dos puntos. Se definen los siguientes tipos de miembros:

|  |  |
| --- | --- |
| **Carácter** | **Descripción** |
| E | Evento |
| F | Campo |
| M | Método (incluidos constructores, destructores y operadores) |
| N | Espacio de nombres |
| P | Propiedad (incluidos los indizadores) |
| T | Tipo (tal como class, delegate, enum, interface y struct) |
| ! | Cadena de error; el resto de la cadena proporciona información acerca del error. Por ejemplo, el generador de documentación genera información de error para vínculos que no se pueden resolver. |

* La segunda parte de la cadena es el nombre completo del elemento, empezando en la raíz del espacio de nombres. El nombre del elemento, los tipos contenedores y el espacio de nombres se separan mediante puntos. Si el nombre del elemento ya contiene puntos, éstos se reemplazan por caracteres # (U+0023). (Se supone que ningún elemento tiene este carácter en su nombre).
* Para los métodos y propiedades con argumentos, sigue la lista de argumentos entre paréntesis. Si no existen argumentos, los paréntesis se omiten. Los argumentos se separan por comas. La codificación de cada argumento es la misma que la firma CLI, como se indica a continuación:
* Los argumentos aparecen representados por su nombre de documentación, que se basa en su nombre completo, modificado como se indica a continuación:

Los argumentos que representan tipos genéricos tienen un carácter “’” anexo seguido por el número de parámetros de tipo

Los argumentos que contienen el modificador out o ref llevan un signo @ tras su nombre de tipo. Los argumentos que se pasan por valor o a través de params no tienen una notación especial.

Los argumentos que son matrices se representan como [ lowerbound : size , … , lowerbound : size ], donde el número de comas es el rango menos 1, y los límites inferiores y tamaños de cada dimensión, si se conocen, se representan en formato decimal. Si no se especifica un límite ni un tamaño inferior, se omite. Si el límite inferior (lowerbound) y el tamaño (size) de una dimensión particular se omiten, también se omite el signo “:”. Las matrices escalonadas se representan con un “[]” por nivel.

Los argumentos con tipos de puntero distintos de void se representan con un \* a continuación del nombre de tipo. Un puntero void se representa mediante un nombre de tipo de System.Void.

Los argumentos que hacen referencia a parámetros de tipos genéricos definidos en tipos se codifican mediante el carácter “`” seguido por el índice de base cero del parámetro de tipo.

Los argumentos que usan parámetros de tipos genéricos definidos en métodos utilizan una doble marca inversa “``” en lugar de “`”, que se usa para los tipos.

Los argumentos que hacen referencia a tipos genéricos construidos se codifican mediante el tipo genérico, seguido por “{“, seguido por una lista separada por comas de los argumentos de tipo, seguida por “}”.

* + 1. Ejemplos de cadena de identificador

Los siguientes ejemplos muestran cada uno un fragmento de código C#, junto con la cadena de id. que se produce a partir de cada elemento de código fuente que puede tener un comentario de documentación:

* Los tipos se representan con su nombre completo, incrementado con información genérica.

enum Color { Red, Blue, Green }

namespace Acme  
{  
 interface IProcess {...}

struct ValueType {...}

class Widget: IProcess  
 {  
 public class NestedClass {...}

public interface IMenuItem {...}

public delegate void Del(int i);

public enum Direction { North, South, East, West }  
 }

class MyList<T>  
 {  
 class Helper<U,V> {...}  
 }  
}

"T:Color"  
"T:Acme.IProcess"  
"T:Acme.ValueType"  
"T:Acme.Widget"  
"T:Acme.Widget.NestedClass"  
"T:Acme.Widget.IMenuItem"  
"T:Acme.Widget.Del"  
"T:Acme.Widget.Direction"  
”T:Acme.MyList`1”  
”T:Acme.MyList`1.Helper`2”

* Los campos se representan con su nombre completo:

namespace Acme  
{  
 struct ValueType  
 {  
 private int total;  
 }

class Widget: IProcess  
 {  
 public class NestedClass  
 {  
 private int value;  
 }

private string message;  
 private static Color defaultColor;  
 private const double PI = 3.14159;  
 protected readonly double monthlyAverage;  
 private long[] array1;  
 private Widget[,] array2;  
 private unsafe int \*pCount;  
 private unsafe float \*\*ppValues;  
 }  
}

"F:Acme.ValueType.total"  
"F:Acme.Widget.NestedClass.value"  
"F:Acme.Widget.message"  
"F:Acme.Widget.defaultColor"  
"F:Acme.Widget.PI"  
"F:Acme.Widget.monthlyAverage"  
"F:Acme.Widget.array1"  
"F:Acme.Widget.array2"  
"F:Acme.Widget.pCount"  
"F:Acme.Widget.ppValues"

* Constructores.

namespace Acme  
{  
 class Widget: IProcess  
 {  
 static Widget() {...}

public Widget() {...}

public Widget(string s) {...}  
 }  
}

"M:Acme.Widget.#cctor"  
"M:Acme.Widget.#ctor"  
"M:Acme.Widget.#ctor(System.String)"

* Destructores.

namespace Acme  
{  
 class Widget: IProcess  
 {  
 ~Widget() {...}  
 }  
}

"M:Acme.Widget.Finalize"

* Métodos.

namespace Acme  
{  
 struct ValueType  
 {  
 public void M(int i) {...}  
 }

class Widget: IProcess  
 {  
 public class NestedClass  
 {  
 public void M(int i) {...}  
 }

public static void M0() {...}  
 public void M1(char c, out float f, ref ValueType v) {...}  
 public void M2(short[] x1, int[,] x2, long[][] x3) {...}  
 public void M3(long[][] x3, Widget[][,,] x4) {...}  
 public unsafe void M4(char \*pc, Color \*\*pf) {...}  
 public unsafe void M5(void \*pv, double \*[][,] pd) {...}  
 public void M6(int i, params object[] args) {...}  
 }

class MyList<T>  
 {  
 public void Test(T t) { }  
 }

class UseList  
 {  
 public void Process(MyList<int> list) { }  
 public MyList<T> GetValues<T>(T inputValue) { return null; }  
 }  
}

"M:Acme.ValueType.M(System.Int32)"  
"M:Acme.Widget.NestedClass.M(System.Int32)"  
"M:Acme.Widget.M0"  
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"  
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"  
"M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"  
"M:Acme.Widget.M4(System.Char\*,Color\*\*)"  
"M:Acme.Widget.M5(System.Void\*,System.Double\*[0:,0:][])"  
"M:Acme.Widget.M6(System.Int32,System.Object[])"  
”M:Acme.MyList`1.Test(`0)”  
”M:Acme.UseList.Process(Acme.MyList{System.Int32})”  
”M:Acme.UseList.GetValues``(``0)”

* Propiedades e indizadores.

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public int Width { get {...} set {...} }  
 public int this[int i] { get {...} set {...} }  
 public int this[string s, int i] { get {...} set {...} }  
 }  
}

"P:Acme.Widget.Width"  
"P:Acme.Widget.Item(System.Int32)"  
"P:Acme.Widget.Item(System.String,System.Int32)"

* Eventos.

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public event Del AnEvent;  
 }  
}

"E:Acme.Widget.AnEvent"

* Operadores unarios.

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public static Widget operator+(Widget x) {...}  
 }  
}

"M:Acme.Widget.op\_UnaryPlus(Acme.Widget)"

El conjunto completo de nombres de función de operador unario utilizados es el siguiente: op\_UnaryPlus, op\_UnaryNegation, op\_LogicalNot, op\_OnesComplement, op\_Increment, op\_Decrement, op\_True y op\_False.

* Operadores binarios.

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public static Widget operator+(Widget x1, Widget x2) {...}  
 }  
}

"M:Acme.Widget.op\_Addition(Acme.Widget,Acme.Widget)"

El conjunto completo de nombres de función de operador binario utilizados es el siguiente: op\_Addition, op\_Subtraction, op\_Multiply, op\_Division, op\_Modulus, op\_BitwiseAnd, op\_BitwiseOr, op\_ExclusiveOr, op\_LeftShift, op\_RightShift, op\_Equality, op\_Inequality, op\_LessThan, op\_LessThanOrEqual, op\_GreaterThan y op\_GreaterThanOrEqual.

* Los operadores de conversión tienen un carácter final “~” seguido del tipo de valor devuelto.

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public static explicit operator int(Widget x) {...}  
 public static implicit operator long(Widget x) {...}  
 }  
}

"M:Acme.Widget.op\_Explicit(Acme.Widget)~System.Int32"  
"M:Acme.Widget.op\_Implicit(Acme.Widget)~System.Int64"

* 1. Un ejemplo
     1. Código fuente C#

En el ejemplo siguiente se muestra el código fuente de una clase Point:

namespace Graphics  
{  
  
/// <summary>Class <c>Point</c> models a point in a two-dimensional plane.  
/// </summary>  
public class Point   
{

/// <summary>Instance variable <c>x</c> represents the point's  
 /// x-coordinate.</summary>  
 private int x;

/// <summary>Instance variable <c>y</c> represents the point's  
 /// y-coordinate.</summary>  
 private int y;

/// <value>Property <c>X</c> represents the point's x-coordinate.</value>  
 public int X  
 {  
 get { return x; }  
 set { x = value; }  
 }

/// <value>Property <c>Y</c> represents the point's y-coordinate.</value>  
 public int Y  
 {  
 get { return y; }  
 set { y = value; }  
 }

/// <summary>This constructor initializes the new Point to  
 /// (0,0).</summary>  
 public Point() : this(0,0) {}

/// <summary>This constructor initializes the new Point to  
 /// (<paramref name="xor"/>,<paramref name="yor"/>).</summary>  
 /// <param><c>xor</c> is the new Point's x-coordinate.</param>  
 /// <param><c>yor</c> is the new Point's y-coordinate.</param>  
 public Point(int xor, int yor) {  
 X = xor;  
 Y = yor;  
 }

/// <summary>This method changes the point's location to  
 /// the given coordinates.</summary>  
 /// <param><c>xor</c> is the new x-coordinate.</param>  
 /// <param><c>yor</c> is the new y-coordinate.</param>  
 /// <see cref="Translate"/>  
 public void Move(int xor, int yor) {  
 X = xor;  
 Y = yor;  
 }

/// <summary>This method changes the point's location by  
 /// the given x- and y-offsets.  
 /// <example>For example:  
 /// <code>  
 /// Point p = new Point(3,5);  
 /// p.Translate(-1,3);  
 /// </code>  
 /// results in <c>p</c>'s having the value (2,8).  
 /// </example>  
 /// </summary>  
 /// <param><c>xor</c> is the relative x-offset.</param>  
 /// <param><c>yor</c> is the relative y-offset.</param>  
 /// <see cref="Move"/>  
 public void Translate(int xor, int yor) {  
 X += xor;  
 Y += yor;  
 }

/// <summary>This method determines whether two Points have the same  
 /// location.</summary>  
 /// <param><c>o</c> is the object to be compared to the current object.  
 /// </param>  
 /// <returns>True if the Points have the same location and they have  
 /// the exact same type; otherwise, false.</returns>  
 /// <seealso cref="operator=="/>  
 /// <seealso cref="operator!="/>  
 public override bool Equals(object o) {  
 if (o == null) {  
 return false;  
 }

if (this == o) {  
 return true;  
 }

if (GetType() == o.GetType()) {  
 Point p = (Point)o;  
 return (X == p.X) && (Y == p.Y);  
 }  
 return false;  
 }

/// <summary>Report a point's location as a string.</summary>  
 /// <returns>A string representing a point's location, in the form (x,y),  
 /// without any leading, training, or embedded whitespace.</returns>  
 public override string ToString() {  
 return "(" + X + "," + Y + ")";  
 }

/// <summary>This operator determines whether two Points have the same  
 /// location.</summary>  
 /// <param><c>p1</c> is the first Point to be compared.</param>  
 /// <param><c>p2</c> is the second Point to be compared.</param>  
 /// <returns>True if the Points have the same location and they have  
 /// the exact same type; otherwise, false.</returns>  
 /// <seealso cref="Equals"/>  
 /// <seealso cref="operator!="/>  
 public static bool operator==(Point p1, Point p2) {  
 if ((object)p1 == null || (object)p2 == null) {  
 return false;  
 }  
  
 if (p1.GetType() == p2.GetType()) {  
 return (p1.X == p2.X) && (p1.Y == p2.Y);  
 }  
  
 return false;  
 }

/// <summary>This operator determines whether two Points have the same  
 /// location.</summary>  
 /// <param><c>p1</c> is the first Point to be compared.</param>  
 /// <param><c>p2</c> is the second Point to be compared.</param>  
 /// <returns>True if the Points do not have the same location and the  
 /// exact same type; otherwise, false.</returns>  
 /// <seealso cref="Equals"/>  
 /// <seealso cref="operator=="/>  
 public static bool operator!=(Point p1, Point p2) {  
 return !(p1 == p2);  
 }

/// <summary>This is the entry point of the Point class testing  
 /// program.  
 /// <para>This program tests each method and operator, and  
 /// is intended to be run after any non-trvial maintenance has  
 /// been performed on the Point class.</para></summary>  
 public static void Main() {  
 // class test code goes here  
 }  
}  
}

* + 1. XML resultante

Éste es el resultado producido por un generador de documentación habiéndose proporcionado al mismo el código fuente para la clase Point, como se mostró anteriormente:

<?xml version="1.0"?>  
<doc>  
 <assembly>  
 <name>Point</name>  
 </assembly>  
 <members>  
 <member name="T:Graphics.Point">  
 <summary>Class <c>Point</c> models a point in a two-dimensional  
 plane.  
 </summary>  
 </member>

<member name="F:Graphics.Point.x">  
 <summary>Instance variable <c>x</c> represents the point's  
 x-coordinate.</summary>  
 </member>

<member name="F:Graphics.Point.y">  
 <summary>Instance variable <c>y</c> represents the point's  
 y-coordinate.</summary>  
 </member>

<member name="M:Graphics.Point.#ctor">  
 <summary>This constructor initializes the new Point to  
 (0,0).</summary>  
 </member>

<member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">  
 <summary>This constructor initializes the new Point to  
 (<paramref name="xor"/>,<paramref name="yor"/>).</summary>  
 <param><c>xor</c> is the new Point's x-coordinate.</param>  
 <param><c>yor</c> is the new Point's y-coordinate.</param>  
 </member>

<member name="M:Graphics.Point.Move(System.Int32,System.Int32)">  
 <summary>This method changes the point's location to  
 the given coordinates.</summary>  
 <param><c>xor</c> is the new x-coordinate.</param>  
 <param><c>yor</c> is the new y-coordinate.</param>  
 <see cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>  
 </member>

<member  
 name="M:Graphics.Point.Translate(System.Int32,System.Int32)">  
 <summary>This method changes the point's location by  
 the given x- and y-offsets.  
 <example>For example:  
 <code>  
 Point p = new Point(3,5);  
 p.Translate(-1,3);  
 </code>  
 results in <c>p</c>'s having the value (2,8).  
 </example>  
 </summary>  
 <param><c>xor</c> is the relative x-offset.</param>  
 <param><c>yor</c> is the relative y-offset.</param>  
 <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>  
 </member>

<member name="M:Graphics.Point.Equals(System.Object)">  
 <summary>This method determines whether two Points have the same  
 location.</summary>  
 <param><c>o</c> is the object to be compared to the current  
 object.  
 </param>  
 <returns>True if the Points have the same location and they have  
 the exact same type; otherwise, false.</returns>  
 <seealso  
 cref="M:Graphics.Point.op\_Equality(Graphics.Point,Graphics.Point)"/>  
 <seealso  
 cref="M:Graphics.Point.op\_Inequality(Graphics.Point,Graphics.Point)"/>  
 </member>

<member name="M:Graphics.Point.ToString">  
 <summary>Report a point's location as a string.</summary>  
 <returns>A string representing a point's location, in the form  
 (x,y),  
 without any leading, training, or embedded whitespace.</returns>  
 </member>

<member  
 name="M:Graphics.Point.op\_Equality(Graphics.Point,Graphics.Point)">  
 <summary>This operator determines whether two Points have the  
 same  
 location.</summary>  
 <param><c>p1</c> is the first Point to be compared.</param>  
 <param><c>p2</c> is the second Point to be compared.</param>  
 <returns>True if the Points have the same location and they have  
 the exact same type; otherwise, false.</returns>  
 <seealso cref="M:Graphics.Point.Equals(System.Object)"/>  
 <seealso  
 cref="M:Graphics.Point.op\_Inequality(Graphics.Point,Graphics.Point)"/>  
 </member>

<member  
 name="M:Graphics.Point.op\_Inequality(Graphics.Point,Graphics.Point)">  
 <summary>This operator determines whether two Points have the  
 same  
 location.</summary>  
 <param><c>p1</c> is the first Point to be compared.</param>  
 <param><c>p2</c> is the second Point to be compared.</param>  
 <returns>True if the Points do not have the same location and  
 the  
 exact same type; otherwise, false.</returns>  
 <seealso cref="M:Graphics.Point.Equals(System.Object)"/>  
 <seealso  
 cref="M:Graphics.Point.op\_Equality(Graphics.Point,Graphics.Point)"/>  
 </member>

<member name="M:Graphics.Point.Main">  
 <summary>This is the entry point of the Point class testing  
 program.  
 <para>This program tests each method and operator, and  
 is intended to be run after any non-trvial maintenance has  
 been performed on the Point class.</para></summary>  
 </member>

<member name="P:Graphics.Point.X">  
 <value>Property <c>X</c> represents the point's  
 x-coordinate.</value>  
 </member>

<member name="P:Graphics.Point.Y">  
 <value>Property <c>Y</c> represents the point's  
 y-coordinate.</value>  
 </member>  
 </members>  
</doc>

1. Gramática

Este apéndice contiene resúmenes de las gramáticas léxicas y sintácticas que pertenecen al documento principal, así como de las extensiones de gramática del código no seguro. Las construcciones de gramática aparecen aquí en el mismo orden en que aparecen en el documento principal.

* 1. Gramática léxica

input:  
input-sectionopt

input-section:  
input-section-part  
input-section input-section-part

input-section-part:  
input-elementsopt new-line  
pp-directive

input-elements:  
input-element  
input-elements input-element

input-element:  
whitespace  
comment  
token

* + 1. Terminadores de línea

new-line:  
Carriage return character (U+000D)  
Line feed character (U+000A)  
Carriage return character (U+000D) followed by line feed character (U+000A)  
Next line character (U+0085)  
Line separator character (U+2028)  
Paragraph separator character (U+2029)

* + 1. Comentarios

comment:  
single-line-comment  
delimited-comment

single-line-comment:  
// input-charactersopt

input-characters:  
input-character  
input-characters input-character

input-character:  
Any Unicode character except a new-line-character

new-line-character:  
Carriage return character (U+000D)  
Line feed character (U+000A)  
Next line character (U+0085)  
Line separator character (U+2028)  
Paragraph separator character (U+2029)

delimited-comment:  
/\* delimited-comment-textopt asterisks /

delimited-comment-text:  
delimited-comment-section  
delimited-comment-text delimited-comment-section

delimited-comment-section:  
/  
asterisksopt not-slash-or-asterisk

asterisks:  
\*  
asterisks \*

not-slash-or-asterisk:  
Any Unicode character except / or \*

* + 1. Espacio en blanco

whitespace:  
Any character with Unicode class Zs  
Horizontal tab character (U+0009)  
Vertical tab character (U+000B)  
Form feed character (U+000C)

* + 1. Tokens

token:  
identifier  
keyword  
integer-literal  
real-literal  
character-literal  
string-literal  
operator-or-punctuator

* + 1. Secuencias de escape de caracteres Unicode

unicode-escape-sequence:  
\u hex-digit hex-digit hex-digit hex-digit  
\U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit

* + 1. Identificadores

identifier:  
available-identifier  
@ identifier-or-keyword

available-identifier:  
An identifier-or-keyword that is not a keyword

identifier-or-keyword:  
identifier-start-character identifier-part-charactersopt

identifier-start-character:  
letter-character  
\_ (the underscore character U+005F)

identifier-part-characters:  
identifier-part-character  
identifier-part-characters identifier-part-character

identifier-part-character:  
letter-character  
decimal-digit-character  
connecting-character  
combining-character  
formatting-character

letter-character:  
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl   
A unicode-escape-sequence representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:  
A Unicode character of classes Mn or Mc   
A unicode-escape-sequence representing a character of classes Mn or Mc

decimal-digit-character:  
A Unicode character of the class Nd   
A unicode-escape-sequence representing a character of the class Nd

connecting-character:   
A Unicode character of the class Pc  
A unicode-escape-sequence representing a character of the class Pc

formatting-character:   
A Unicode character of the class Cf  
A unicode-escape-sequence representing a character of the class Cf

* + 1. Palabras clave

keyword: one of  
abstract as base bool break  
byte case catch char checked  
class const continue decimal default  
delegate do double else enum  
event explicit extern false finally  
fixed float for foreach goto  
if implicit in int interface  
internal is lock long namespace  
new null object operator out  
override params private protected public  
readonly ref return sbyte sealed  
short sizeof stackalloc static string  
struct switch this throw true  
try typeof uint ulong unchecked  
unsafe ushort using virtual void  
volatile while

* + 1. Literales

literal:  
boolean-literal  
integer-literal  
real-literal  
character-literal  
string-literal  
null-literal

boolean-literal:  
true  
false

integer-literal:  
decimal-integer-literal  
hexadecimal-integer-literal

decimal-integer-literal:  
decimal-digits integer-type-suffixopt

decimal-digits:  
decimal-digit  
decimal-digits decimal-digit

decimal-digit: one of  
0 1 2 3 4 5 6 7 8 9

integer-type-suffix: one of  
U u L l UL Ul uL ul LU Lu lU lu

hexadecimal-integer-literal:  
0x hex-digits integer-type-suffixopt  
0X hex-digits integer-type-suffixopt

hex-digits:  
hex-digit  
hex-digits hex-digit

hex-digit: one of  
0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

real-literal:  
decimal-digits . decimal-digits exponent-partopt real-type-suffixopt  
. decimal-digits exponent-partopt real-type-suffixopt  
decimal-digits exponent-part real-type-suffixopt  
decimal-digits real-type-suffix

exponent-part:  
e signopt decimal-digits  
E signopt decimal-digits

sign: one of  
+ -

real-type-suffix: one of  
F f D d M m

character-literal:  
' character '

character:  
single-character  
simple-escape-sequence  
hexadecimal-escape-sequence  
unicode-escape-sequence

single-character:  
 Any character except ' (U+0027), \ (U+005C), and new-line-character

simple-escape-sequence: one of  
\' \" \\ \0 \a \b \f \n \r \t \v

hexadecimal-escape-sequence:  
\x hex-digit hex-digitopt hex-digitopt hex-digitopt

string-literal:  
regular-string-literal  
verbatim-string-literal

regular-string-literal:  
" regular-string-literal-charactersopt "

regular-string-literal-characters:  
regular-string-literal-character  
regular-string-literal-characters regular-string-literal-character

regular-string-literal-character:  
single-regular-string-literal-character  
simple-escape-sequence  
hexadecimal-escape-sequence  
unicode-escape-sequence

single-regular-string-literal-character:  
Any character except " (U+0022), \ (U+005C), and new-line-character

verbatim-string-literal:  
@" verbatim-string-literal-charactersopt "

verbatim-string-literal-characters:  
verbatim-string-literal-character  
verbatim-string-literal-characters verbatim-string-literal-character

verbatim-string-literal-character:  
single-verbatim-string-literal-character  
quote-escape-sequence

single-verbatim-string-literal-character:  
any character except "

quote-escape-sequence:  
""

null-literal:  
null

* + 1. Operadores y signos de puntuación

operator-or-punctuator: one of  
{ } [ ] ( ) . , : ;  
+ - \* / % & | ^ ! ~  
= < > ? ?? :: ++ -- && ||  
-> == != <= >= += -= \*= /= %=  
&= |= ^= << <<= =>

right-shift:  
>|>

right-shift-assignment:  
>|>=

* + 1. Directivas de preprocesamiento

pp-directive:  
pp-declaration  
pp-conditional  
pp-line  
pp-diagnostic  
pp-region   
pp-pragma

conditional-symbol:  
Any identifier-or-keyword except true or false

pp-expression:  
whitespaceopt pp-or-expression whitespaceopt

pp-or-expression:  
pp-and-expression  
pp-or-expression whitespaceopt || whitespaceopt pp-and-expression

pp-and-expression:  
pp-equality-expression  
pp-and-expression whitespaceopt && whitespaceopt pp-equality-expression

pp-equality-expression:  
pp-unary-expression  
pp-equality-expression whitespaceopt == whitespaceopt pp-unary-expression  
pp-equality-expression whitespaceopt != whitespaceopt pp-unary-expression

pp-unary-expression:  
pp-primary-expression  
! whitespaceopt pp-unary-expression

pp-primary-expression:  
true  
false  
conditional-symbol  
( whitespaceopt pp-expression whitespaceopt )

pp-declaration:  
whitespaceopt # whitespaceopt define whitespace conditional-symbol pp-new-line  
whitespaceopt # whitespaceopt undef whitespace conditional-symbol pp-new-line

pp-new-line:  
whitespaceopt single-line-commentopt new-line

pp-conditional:  
pp-if-section pp-elif-sectionsopt pp-else-sectionopt pp-endif

pp-if-section:  
whitespaceopt # whitespaceopt if whitespace pp-expression pp-new-line conditional-sectionopt

pp-elif-sections:  
pp-elif-section  
pp-elif-sections pp-elif-section

pp-elif-section:  
whitespaceopt # whitespaceopt elif whitespace pp-expression pp-new-line conditional-sectionopt

pp-else-section:  
whitespaceopt # whitespaceopt else pp-new-line conditional-sectionopt

pp-endif:  
whitespaceopt # whitespaceopt endif pp-new-line

conditional-section:  
input-section  
skipped-section

skipped-section:  
skipped-section-part  
skipped-section skipped-section-part

skipped-section-part:  
skipped-charactersopt new-line  
pp-directive

skipped-characters:  
whitespaceopt not-number-sign input-charactersopt

not-number-sign:  
Any input-character except #

pp-diagnostic:  
whitespaceopt # whitespaceopt error pp-message  
whitespaceopt # whitespaceopt warning pp-message

pp-message:  
new-line  
whitespace input-charactersopt new-line

pp-region:  
pp-start-region conditional-sectionopt pp-end-region

pp-start-region:  
whitespaceopt # whitespaceopt region pp-message

pp-end-region:  
whitespaceopt # whitespaceopt endregion pp-message

pp-line:  
whitespaceopt # whitespaceopt line whitespace line-indicator pp-new-line

line-indicator:  
decimal-digits whitespace file-name   
decimal-digits  
default   
hidden

file-name:  
" file-name-characters "

file-name-characters:  
file-name-character  
file-name-characters file-name-character

file-name-character:  
Any input-character except "

pp-pragma:  
whitespaceopt # whitespaceopt pragma whitespace pragma-body pp-new-line

pragma-body:  
pragma-warning-body

pragma-warning-body:  
warning whitespace warning-action  
warning whitespace warning-action whitespace warning-list

warning-action:  
disable  
restore

warning-list:  
decimal-digits  
warning-list whitespaceopt , whitespaceopt decimal-digits

* 1. Gramática sintáctica
     1. Conceptos básicos

namespace-name:  
namespace-or-type-name

type-name:  
namespace-or-type-name

namespace-or-type-name:  
identifier type-argument-listopt  
namespace-or-type-name . identifier type-argument-listoptqualified-alias-member

* + 1. Tipos

type:  
value-type  
reference-type   
type-parameter

value-type:  
struct-type  
enum-type

struct-type:  
type-name  
simple-type   
nullable-type

simple-type:  
numeric-type  
bool

numeric-type:  
integral-type  
floating-point-type  
decimal

integral-type:  
sbyte  
byte  
short  
ushort  
int  
uint  
long  
ulong  
char

floating-point-type:  
float  
double

nullable-type:  
non-nullable-value-type ?

non-nullable-value-type:  
type

enum-type:  
type-name

reference-type:  
class-type  
interface-type  
array-type  
delegate-type

class-type:  
type-name  
object  
dynamic  
string

interface-type:  
type-name

rank-specifiers:  
rank-specifier  
rank-specifiers rank-specifier

rank-specifier:  
[ dim-separatorsopt ]

dim-separators:  
,  
dim-separators ,

delegate-type:  
type-name

type-argument-list:  
< type-arguments >

type-arguments:  
type-argument  
type-arguments , type-argument

type-argument:  
type

type-parameter:  
identifier

* + 1. Variables

variable-reference:  
expression

* + 1. Expresiones

argument-list:  
argument  
argument-list , argument

argument:  
argument-nameopt argument-value

argument-name:  
identifier :

argument-value:  
expression  
ref variable-reference  
out variable-reference

primary-expression:   
primary-no-array-creation-expression  
array-creation-expression

primary-no-array-creation-expression:  
literal  
simple-name  
parenthesized-expression  
member-access  
invocation-expression  
element-access  
this-access  
base-access  
post-increment-expression  
post-decrement-expression  
object-creation-expression  
delegate-creation-expression  
anonymous-object-creation-expression  
typeof-expression  
 checked-expression  
unchecked-expression   
default-value-expression  
anonymous-method-expression

simple-name:  
identifier type-argument-listopt

parenthesized-expression:  
( expression )

member-access:  
primary-expression . identifier type-argument-listopt  
predefined-type . identifier type-argument-listopt  
qualified-alias-member . identifier

predefined-type: one of  
bool byte char decimal double float int long  
object sbyte short string uint ulong ushort

invocation-expression:  
primary-expression ( argument-listopt )

element-access:  
primary-no-array-creation-expression [ argument-list ]

this-access:  
this

base-access:  
base . identifier  
base [ argument-list ]

post-increment-expression:  
primary-expression ++

post-decrement-expression:  
primary-expression --

object-creation-expression:  
new type ( argument-listopt ) object-or-collection-initializeropt   
new type object-or-collection-initializer

object-or-collection-initializer:  
object-initializer  
collection-initializer

object-initializer:  
{ member-initializer-listopt }  
{ member-initializer-list , }

member-initializer-list:  
member-initializer  
member-initializer-list , member-initializer

member-initializer:  
identifier = initializer-value

initializer-value:  
expression  
object-or-collection-initializer

collection-initializer:  
{ element-initializer-list }  
{ element-initializer-list , }

element-initializer-list:  
element-initializer  
element-initializer-list , element-initializer

element-initializer:  
non-assignment-expression  
{ expression-list }

expression-list:  
expression  
expression-list , expression

array-creation-expression:  
new non-array-type [ expression-list ] rank-specifiersopt array-initializeropt  
new array-type array-initializer   
new rank-specifier array-initializer

delegate-creation-expression:  
new delegate-type ( expression )

anonymous-object-creation-expression:  
new anonymous-object-initializer

anonymous-object-initializer:  
{ member-declarator-listopt }  
{ member-declarator-list , }

member-declarator-list:  
member-declarator  
member-declarator-list , member-declarator

member-declarator:  
simple-name  
member-access  
identifier = expression

typeof-expression:  
typeof ( type )  
typeof ( unbound-type-name )  
typeof ( void )

unbound-type-name:  
identifier generic-dimension-specifieropt  
identifier :: identifier generic-dimension-specifieropt  
unbound-type-name . identifier generic-dimension-specifieropt

generic-dimension-specifier:  
< commasopt >

commas:  
,  
commas ,

checked-expression:  
checked ( expression )

unchecked-expression:  
unchecked ( expression )

default-value-expression:  
default ( type )

unary-expression:  
primary-expression  
+ unary-expression  
- unary-expression  
! unary-expression  
~ unary-expression  
pre-increment-expression  
pre-decrement-expression  
cast-expression

pre-increment-expression:  
++ unary-expression

pre-decrement-expression:  
-- unary-expression

cast-expression:  
( type ) unary-expression

multiplicative-expression:  
unary-expression  
multiplicative-expression \* unary-expression  
multiplicative-expression / unary-expression  
multiplicative-expression % unary-expression

additive-expression:  
multiplicative-expression  
additive-expression + multiplicative-expression  
additive-expression – multiplicative-expression

shift-expression:  
additive-expression   
shift-expression << additive-expression  
shift-expression right-shift additive-expression

relational-expression:  
shift-expression  
relational-expression < shift-expression  
relational-expression > shift-expression  
relational-expression <= shift-expression  
relational-expression >= shift-expression  
relational-expression is type  
relational-expression as type

equality-expression:  
relational-expression  
equality-expression == relational-expression  
equality-expression != relational-expression

and-expression:  
equality-expression  
and-expression & equality-expression

exclusive-or-expression:  
and-expression  
exclusive-or-expression ^ and-expression

inclusive-or-expression:  
exclusive-or-expression  
inclusive-or-expression | exclusive-or-expression

conditional-and-expression:  
inclusive-or-expression  
conditional-and-expression && inclusive-or-expression

conditional-or-expression:  
conditional-and-expression  
conditional-or-expression || conditional-and-expression

null-coalescing-expression:  
conditional-or-expression  
conditional-or-expression ?? null-coalescing-expression

conditional-expression:  
null-coalescing-expression  
null-coalescing-expression ? expression : expression

lambda-expression:  
anonymous-function-signature => anonymous-function-body

anonymous-method-expression:  
delegate explicit-anonymous-function-signatureopt block

anonymous-function-signature:  
explicit-anonymous-function-signature   
implicit-anonymous-function-signature

explicit-anonymous-function-signature:  
( explicit-anonymous-function-parameter-list*opt* )

explicit-anonymous-function-parameter-list:  
explicit-anonymous-function-parameter  
explicit-anonymous-function-parameter-list , explicit-anonymous-function-parameter

explicit-anonymous-function-parameter:  
anonymous-function-parameter-modifieropt type identifier

anonymous-function-parameter-modifier:   
ref  
out

implicit-anonymous-function-signature:  
( implicit-anonymous-function-parameter-list*opt* )  
implicit-anonymous-function-parameter

implicit-anonymous-function-parameter-list:  
implicit-anonymous-function-parameter  
implicit-anonymous-function-parameter-list , implicit-anonymous-function-parameter

implicit-anonymous-function-parameter:  
identifier

anonymous-function-body:  
expression  
block

query-expression:  
from-clause query-body

from-clause:  
from typeopt identifier in expression

query-body:  
query-body-clausesopt select-or-group-clause query-continuationopt

query-body-clauses:  
query-body-clause  
query-body-clauses query-body-clause

query-body-clause:  
from-clause  
let-clause  
where-clause  
join-clause  
join-into-clause  
orderby-clause

let-clause:  
let identifier = expression

where-clause:  
where boolean-expression

join-clause:  
join typeopt identifier in expression on expression equals expression

join-into-clause:  
join typeopt identifier in expression on expression equals expression into identifier

orderby-clause:  
orderby orderings

orderings:  
ordering  
orderings , ordering

ordering:  
expression ordering-directionopt

ordering-direction:  
ascending  
descending

select-or-group-clause:  
select-clause  
group-clause

select-clause:  
select expression

group-clause:  
group expression by expression

query-continuation:  
into identifier query-body

assignment:  
unary-expression assignment-operator expression

assignment-operator:  
=  
+=  
-=  
\*=  
/=  
%=  
&=  
|=  
^=  
<<=  
right-shift-assignment

expression:   
non-assignment-expression  
assignment

non-assignment-expression:  
conditional-expression  
lambda-expression  
query-expression

constant-expression:  
expression

boolean-expression:  
expression

* + 1. Instrucciones

statement:  
labeled-statement  
declaration-statement  
embedded-statement

embedded-statement:  
block  
empty-statement  
expression-statement  
selection-statement  
iteration-statement  
jump-statement  
try-statement  
checked-statement  
unchecked-statement  
lock-statement  
using-statement   
yield-statement

block:  
{ statement-listopt }

statement-list:  
statement  
statement-list statement

empty-statement:  
;

labeled-statement:  
identifier : statement

declaration-statement:  
local-variable-declaration ;  
local-constant-declaration ;

local-variable-declaration:  
local-variable-type local-variable-declarators

local-variable-type:  
type  
var

local-variable-declarators:  
local-variable-declarator  
local-variable-declarators , local-variable-declarator

local-variable-declarator:  
identifier  
identifier = local-variable-initializer

local-variable-initializer:  
expression  
array-initializer

local-constant-declaration:  
const type constant-declarators

constant-declarators:  
constant-declarator  
constant-declarators , constant-declarator

constant-declarator:  
identifier = constant-expression

expression-statement:  
statement-expression ;

statement-expression:  
invocation-expression  
object-creation-expression  
assignment  
post-increment-expression  
post-decrement-expression  
pre-increment-expression  
pre-decrement-expression

selection-statement:  
if-statement  
switch-statement

if-statement:  
if ( boolean-expression ) embedded-statement  
if ( boolean-expression ) embedded-statement else embedded-statement

switch-statement:  
switch ( expression ) switch-block

switch-block:  
{ switch-sectionsopt }

switch-sections:  
switch-section  
switch-sections switch-section

switch-section:  
switch-labels statement-list

switch-labels:  
switch-label  
switch-labels switch-label

switch-label:  
case constant-expression :  
default :

iteration-statement:  
while-statement  
do-statement  
for-statement  
foreach-statement

while-statement:  
while ( boolean-expression ) embedded-statement

do-statement:  
do embedded-statement while ( boolean-expression ) ;

for-statement:  
for ( for-initializeropt ; for-conditionopt ; for-iteratoropt ) embedded-statement

for-initializer:  
local-variable-declaration  
statement-expression-list

for-condition:  
boolean-expression

for-iterator:  
statement-expression-list

statement-expression-list:  
statement-expression  
statement-expression-list , statement-expression

foreach-statement:  
foreach ( local-variable-type identifier in expression ) embedded-statement

jump-statement:  
break-statement  
continue-statement  
goto-statement  
return-statement  
throw-statement

break-statement:  
break ;

continue-statement:  
continue ;

goto-statement:  
goto identifier ;  
goto case constant-expression ;  
goto default ;

return-statement:  
return expressionopt ;

throw-statement:  
throw expressionopt ;

try-statement:  
try block catch-clauses  
try block finally-clause  
try block catch-clauses finally-clause

catch-clauses:  
specific-catch-clauses general-catch-clauseopt  
specific-catch-clausesopt general-catch-clause

specific-catch-clauses:  
specific-catch-clause  
specific-catch-clauses specific-catch-clause

specific-catch-clause:  
catch ( class-type identifieropt ) block

general-catch-clause:  
catch block

finally-clause:  
finally block

checked-statement:  
checked block

unchecked-statement:  
unchecked block

lock-statement:  
lock ( expression ) embedded-statement

using-statement:  
using ( resource-acquisition ) embedded-statement

resource-acquisition:  
local-variable-declaration  
expression

yield-statement:  
yield return expression ;  
yield break ;

* + 1. Espacios de nombres

compilation-unit:  
extern-alias-directivesopt using-directivesopt global-attributesopt  
 namespace-member-declarationsopt

namespace-declaration:  
namespace qualified-identifier namespace-body ;opt

qualified-identifier:  
identifier  
qualified-identifier . identifier

namespace-body:  
{ extern-alias-directivesopt using-directivesopt namespace-member-declarationsopt }

extern-alias-directives:  
extern-alias-directive  
extern-alias-directives extern-alias-directive

extern-alias-directive:  
extern alias identifier ;

using-directives:  
using-directive  
using-directives using-directive

using-directive:  
using-alias-directive  
using-namespace-directive

using-alias-directive:  
using identifier = namespace-or-type-name ;

using-namespace-directive:  
using namespace-name ;

namespace-member-declarations:  
namespace-member-declaration  
namespace-member-declarations namespace-member-declaration

namespace-member-declaration:  
namespace-declaration  
type-declaration

type-declaration:  
class-declaration  
struct-declaration  
interface-declaration  
enum-declaration  
delegate-declaration

qualified-alias-member:  
identifier :: identifier type-argument-listopt

* + 1. Clases

class-declaration:  
attributesopt class-modifiersopt partialopt class identifier type-parameter-listopt  
 class-baseopt type-parameter-constraints-clausesopt class-body ;opt

class-modifiers:  
class-modifier  
class-modifiers class-modifier

class-modifier:  
new  
public  
protected  
internal  
private  
abstract  
sealed  
static

type-parameter-list:  
< type-parameters >

type-parameters:  
attributesopt type-parameter  
type-parameters , attributesopt type-parameter

type-parameter:  
identifier

class-base:  
: class-type  
: interface-type-list  
: class-type , interface-type-list

interface-type-list:  
interface-type  
interface-type-list , interface-type

type-parameter-constraints-clauses:  
type-parameter-constraints-clause  
type-parameter-constraints-clauses type-parameter-constraints-clause

type-parameter-constraints-clause:  
where type-parameter : type-parameter-constraints

type-parameter-constraints:  
primary-constraint  
secondary-constraints  
constructor-constraint  
primary-constraint , secondary-constraints  
primary-constraint , constructor-constraint  
secondary-constraints , constructor-constraint  
primary-constraint , secondary-constraints , constructor-constraint

primary-constraint:  
class-type  
class  
struct

secondary-constraints:  
interface-type  
type-parameter  
secondary-constraints , interface-type  
secondary-constraints , type-parameter

constructor-constraint:  
new ( )

class-body:  
{ class-member-declarationsopt }

class-member-declarations:  
class-member-declaration  
class-member-declarations class-member-declaration

class-member-declaration:  
constant-declaration  
field-declaration  
method-declaration  
property-declaration  
event-declaration  
indexer-declaration  
operator-declaration  
constructor-declaration  
destructor-declaration  
static-constructor-declaration  
type-declaration

constant-declaration:  
attributesopt constant-modifiersopt const type constant-declarators ;

constant-modifiers:  
constant-modifier  
constant-modifiers constant-modifier

constant-modifier:  
new  
public  
protected  
internal  
private

constant-declarators:  
constant-declarator  
constant-declarators , constant-declarator

constant-declarator:  
identifier = constant-expression

field-declaration:  
attributesopt field-modifiersopt type variable-declarators ;

field-modifiers:  
field-modifier  
field-modifiers field-modifier

field-modifier:  
new  
public  
protected  
internal  
private  
static  
readonly  
volatile

variable-declarators:  
variable-declarator  
variable-declarators , variable-declarator

variable-declarator:  
identifier  
identifier = variable-initializer

variable-initializer:  
expression  
array-initializer

method-declaration:  
method-header method-body

method-header:  
attributesopt method-modifiersopt partialopt return-type member-name type-parameter-listopt  
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt

method-modifiers:  
method-modifier  
method-modifiers method-modifier

method-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

return-type:  
type  
void

member-name:  
identifier  
interface-type . identifier

method-body:  
block  
;

formal-parameter-list:  
fixed-parameters  
fixed-parameters , parameter-array  
parameter-array

fixed-parameters:  
fixed-parameter  
fixed-parameters , fixed-parameter

fixed-parameter:  
attributesopt parameter-modifieropt type identifier default-argumentopt

default-argument:  
= expression

parameter-modifier:  
ref  
out  
this

parameter-array:  
attributesopt params array-type identifier

property-declaration:  
attributesopt property-modifiersopt type member-name { accessor-declarations }

property-modifiers:  
property-modifier  
property-modifiers property-modifier

property-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

member-name:  
identifier  
interface-type . identifier

accessor-declarations:  
get-accessor-declaration set-accessor-declarationopt  
set-accessor-declaration get-accessor-declarationopt

get-accessor-declaration:  
attributesopt accessor-modifieropt  get accessor-body

set-accessor-declaration:  
attributesopt accessor-modifieropt set accessor-body

accessor-modifier:  
protected  
internal  
private  
protected internal  
internal protected

accessor-body:  
block  
;

event-declaration:  
attributesopt event-modifiersopt event type variable-declarators ;  
attributesopt event-modifiersopt event type member-name { event-accessor-declarations }

event-modifiers:  
event-modifier  
event-modifiers event-modifier

event-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

event-accessor-declarations:  
add-accessor-declaration remove-accessor-declaration  
remove-accessor-declaration add-accessor-declaration

add-accessor-declaration:  
attributesopt add block

remove-accessor-declaration:  
attributesopt remove block

indexer-declaration:  
attributesopt indexer-modifiersopt indexer-declarator { accessor-declarations }

indexer-modifiers:  
indexer-modifier  
indexer-modifiers indexer-modifier

indexer-modifier:  
new  
public  
protected  
internal  
private   
virtual  
sealed  
override  
abstract  
extern

indexer-declarator:  
type this [ formal-parameter-list ]  
type interface-type . this [ formal-parameter-list ]

operator-declaration:  
attributesopt operator-modifiers operator-declarator operator-body

operator-modifiers:  
operator-modifier  
operator-modifiers operator-modifier

operator-modifier:  
public  
static  
extern

operator-declarator:  
unary-operator-declarator  
binary-operator-declarator  
conversion-operator-declarator

unary-operator-declarator:  
type operator overloadable-unary-operator ( type identifier )

overloadable-unary-operator: one of  
+ - ! ~ ++ -- true false

binary-operator-declarator:  
type operator overloadable-binary-operator ( type identifier , type identifier )

overloadable-binary-operator:  
+  
-  
\*  
/  
%  
&  
|  
^  
<<  
right-shift  
==  
!=  
>  
<  
>=  
<=

conversion-operator-declarator:  
implicit operator type ( type identifier )  
explicit operator type ( type identifier )

operator-body:  
block  
;

constructor-declaration:  
attributesopt constructor-modifiersopt constructor-declarator constructor-body

constructor-modifiers:  
constructor-modifier  
constructor-modifiers constructor-modifier

constructor-modifier:  
public  
protected  
internal  
private  
extern

constructor-declarator:  
identifier ( formal-parameter-listopt ) constructor-initializeropt

constructor-initializer:  
: base ( argument-listopt )  
: this ( argument-listopt )

constructor-body:  
block  
;

static-constructor-declaration:  
attributesopt static-constructor-modifiers identifier ( ) static-constructor-body

static-constructor-modifiers:  
externopt static  
static externopt

static-constructor-body:  
block  
;

destructor-declaration:  
attributesopt externopt ~ identifier ( ) destructor-body

destructor-body:  
block  
;

* + 1. Structs

struct-declaration:  
attributesopt struct-modifiersopt partialopt struct identifier type-parameter-listopt  
 struct-interfacesopt type-parameter-constraints-clausesopt struct-body ;opt

struct-modifiers:  
struct-modifier  
struct-modifiers struct-modifier

struct-modifier:  
new  
public  
protected  
internal  
private

struct-interfaces:  
: interface-type-list

struct-body:  
{ struct-member-declarationsopt }

struct-member-declarations:  
struct-member-declaration  
struct-member-declarations struct-member-declaration

struct-member-declaration:  
constant-declaration  
field-declaration  
method-declaration  
property-declaration  
event-declaration  
indexer-declaration  
operator-declaration  
constructor-declaration  
static-constructor-declaration  
type-declaration

* + 1. Matrices

array-type:  
non-array-type rank-specifiers

non-array-type:  
type

rank-specifiers:  
rank-specifier  
rank-specifiers rank-specifier

rank-specifier:  
[ dim-separatorsopt ]

dim-separators:  
,  
dim-separators ,

array-initializer:  
{ variable-initializer-listopt }  
{ variable-initializer-list , }

variable-initializer-list:  
variable-initializer  
variable-initializer-list , variable-initializer

variable-initializer:  
expression  
array-initializer

* + 1. Interfaces

interface-declaration:  
attributesopt interface-modifiersopt partialopt interface   
 identifier variant-type-parameter-listopt interface-baseopt  
 type-parameter-constraints-clausesopt interface-body ;opt

interface-modifiers:  
interface-modifier  
interface-modifiers interface-modifier

interface-modifier:  
new  
public  
protected  
internal  
private

variant-type-parameter-list:  
< variant-type-parameters >

variant-type-parameters:  
attributesopt variance-annotationopt  type-parameter  
variant-type-parameters , attributesopt variance-annotationopt type-parameter

variance-annotation:  
in  
out

interface-base:  
: interface-type-list

interface-body:  
{ interface-member-declarationsopt }

interface-member-declarations:  
interface-member-declaration  
interface-member-declarations interface-member-declaration

interface-member-declaration:  
interface-method-declaration  
interface-property-declaration  
interface-event-declaration  
interface-indexer-declaration

interface-method-declaration:  
attributesopt newopt return-type identifier type-parameter-list  
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

interface-property-declaration:  
attributesopt newopt type identifier { interface-accessors }

interface-accessors:  
attributesopt get ;  
attributesopt set ;  
attributesopt get ; attributesopt set ;  
attributesopt set ; attributesopt get ;

interface-event-declaration:  
attributesopt newopt event type identifier ;

interface-indexer-declaration:  
attributesopt newopt type this [ formal-parameter-list ] { interface-accessors }

* + 1. Enumeraciones

enum-declaration:  
attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt

enum-base:  
: integral-type

enum-body:  
{ enum-member-declarationsopt }  
{ enum-member-declarations , }

enum-modifiers:  
enum-modifier  
enum-modifiers enum-modifier

enum-modifier:  
new  
public  
protected  
internal  
private

enum-member-declarations:  
enum-member-declaration  
enum-member-declarations , enum-member-declaration

enum-member-declaration:  
attributesopt identifier  
attributesopt identifier = constant-expression

* + 1. Delegados

delegate-declaration:  
attributesopt delegate-modifiersopt delegate return-type   
 identifier variant-type-parameter-listopt   
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

delegate-modifiers:  
delegate-modifier  
delegate-modifiers delegate-modifier

delegate-modifier:  
new  
public  
protected  
internal  
private

* + 1. Atributos

global-attributes:  
global-attribute-sections

global-attribute-sections:  
global-attribute-section  
global-attribute-sections global-attribute-section

global-attribute-section:  
[ global-attribute-target-specifier attribute-list ]  
[ global-attribute-target-specifier attribute-list , ]

global-attribute-target-specifier:  
global-attribute-target :

global-attribute-target:  
assembly  
module

attributes:  
attribute-sections

attribute-sections:  
attribute-section  
attribute-sections attribute-section

attribute-section:  
[ attribute-target-specifieropt attribute-list ]  
[ attribute-target-specifieropt attribute-list , ]

attribute-target-specifier:  
attribute-target :

attribute-target:  
field  
event  
method  
param  
property  
return  
type

attribute-list:  
attribute  
attribute-list , attribute

attribute:  
attribute-name attribute-argumentsopt

attribute-name:  
 type-name

attribute-arguments:  
( positional-argument-listopt )  
( positional-argument-list , named-argument-list )  
( named-argument-list )

positional-argument-list:  
positional-argument  
positional-argument-list , positional-argument

positional-argument:  
argument-nameopt attribute-argument-expression

named-argument-list:  
named-argument  
named-argument-list , named-argument

named-argument:  
identifier = attribute-argument-expression

attribute-argument-expression:  
expression

* 1. Extensiones de la gramática para el código no seguro

class-modifier:  
...  
unsafe

struct-modifier:  
...  
unsafe

interface-modifier:  
...  
unsafe

delegate-modifier:  
...  
unsafe

field-modifier:  
...  
unsafe

method-modifier:  
...  
unsafe

property-modifier:  
...  
unsafe

event-modifier:  
...  
unsafe

indexer-modifier:  
...  
unsafe

operator-modifier:  
...  
unsafe

constructor-modifier:  
...  
unsafe

destructor-declaration:  
attributesopt externopt unsafeopt ~ identifier ( ) destructor-body  
attributesopt unsafeopt externopt ~ identifier ( ) destructor-body

static-constructor-modifiers:  
externopt unsafeopt static  
unsafeopt externopt static  
externopt static unsafeopt   
unsafeopt static externopt  
static externopt unsafeopt  
static unsafeopt externopt

embedded-statement:  
...  
unsafe-statement   
fixed-statement

unsafe-statement:  
unsafe block

type:  
...  
pointer-type

pointer-type:  
unmanaged-type \*  
void \*

unmanaged-type:  
type

primary-no-array-creation-expression:  
...  
pointer-member-access  
pointer-element-access  
sizeof-expression

unary-expression:  
...  
pointer-indirection-expression  
addressof-expression

pointer-indirection-expression:  
\* unary-expression

pointer-member-access:  
primary-expression -> identifier type-argument-listopt

pointer-element-access:  
primary-no-array-creation-expression [ expression ]

addressof-expression:  
& unary-expression

sizeof-expression:  
sizeof ( unmanaged-type )

fixed-statement:  
fixed ( pointer-type fixed-pointer-declarators ) embedded-statement

fixed-pointer-declarators:  
fixed-pointer-declarator  
fixed-pointer-declarators , fixed-pointer-declarator

fixed-pointer-declarator:  
identifier = fixed-pointer-initializer

fixed-pointer-initializer:  
& variable-reference  
expression

struct-member-declaration:  
…  
fixed-size-buffer-declaration

fixed-size-buffer-declaration:  
attributesopt fixed-size-buffer-modifiersopt fixed buffer-element-type  
 fixed-size-buffer-declarators ;

fixed-size-buffer-modifiers:  
fixed-size-buffer-modifier  
fixed-size-buffer-modifier fixed-size-buffer-modifiers

fixed-size-buffer-modifier:  
new  
public  
protected  
internal  
private  
unsafe

buffer-element-type:  
type

fixed-size-buffer-declarators:  
fixed-size-buffer-declarator  
fixed-size-buffer-declarator fixed-size-buffer-declarators

fixed-size-buffer-declarator:  
identifier [ constant-expression ]

local-variable-initializer:  
…  
stackalloc-initializer

stackalloc-initializer:  
stackalloc unmanaged-type [ expression ]

1. Referencias

Unicode Consortium. The Unicode Standard, Version 3.0. Addison-Wesley, Reading, Massachusetts (EE.UU.), 2000, ISBN 0-201-616335-5.

IEEE. IEEE Standard for Binary Floating-Point Arithmetic (Estándar IEEE para Aritmética Binaria de Punto Flotante). ANSI/IEEE Standard 754-1985. Disponible en <http://www.ieee.org>.

ISO/IEC. C++. ANSI/ISO/IEC 14882:1998.