Especificación del lenguaje  
Microsoft®Visual Basic®

Versión 11.0

Paul Vick, Lucian Wischik

Microsoft Corporation

La información incluida en este documento representa la visión actual de Microsoft Corporation con respecto a los temas que se tratan a fecha de publicación. Microsoft debe responder a las condiciones en constante cambio del mercado, por lo que esta información no se debe interpretar como un compromiso por parte de Microsoft; Microsoft tampoco puede garantizar la precisión de la información tras la fecha de publicación.

Esta Especificación del lenguaje se debe usar solamente con fines informativos. MICROSOFT NO OFRECE NINGUNA GARANTÍA, YA SEA EXPLÍCITA, IMPLÍCITA NI REGLAMENTARIA, CON RESPECTO A LA INFORMACIÓN DE ESTE DOCUMENTO.

Es responsabilidad del usuario el cumplimiento de toda la legislación aplicable en materia de copyright. Sin limitación de los derechos protegidos por copyright, ninguna parte del presente documento podrá ser reproducida, almacenada o introducida en un sistema de recuperación, o bien transmitida en ninguna forma o medio (electrónico, mecánico, mediante fotocopia o grabación, etc.), ni con ningún propósito, sin la autorización expresa y por escrito de Microsoft Corporation.

Microsoft puede ser titular de patentes, solicitudes de patentes, marcas, derechos de autor u otros derechos de propiedad intelectual sobre los contenidos de este documento. Este documento no otorga ninguna licencia sobre estas patentes, marcas, derechos de autor u otros derechos de propiedad intelectual, a menos que se indique expresamente en un contrato escrito de licencia de Microsoft.

Salvo que se indique lo contrario, los ejemplos de sociedades, organizaciones, productos, nombres de dominio, direcciones de correo electrónico, logotipos, personas, lugares y acontecimientos aquí citados son ficticios; con ellos no se pretende, ni se debe deducir, relación alguna con sociedades, organizaciones, productos, nombres de dominio, direcciones de correo electrónico, logotipos, personas, lugares o acontecimientos reales.

© 2012 Microsoft Corporation. Reservados todos los derechos.

Microsoft, MS-DOS, Visual Basic, Windows 2000, Windows 95, Windows 98, Windows ME, Windows NT, Windows XP, Windows Vista y Windows son marcas comerciales o marcas registradas de Microsoft Corporation en Estados Unidos y en otros países.

Los nombres de las compañías y de los productos reales mencionados en este documento pueden ser marcas comerciales de sus respectivos propietarios.

Tabla de contenido

[1. Introduction 1](#_Toc327273727)

[1.1 Notación gramatical 1](#_Toc327273728)

[1.2 Compatibilidad 2](#_Toc327273729)

[1.2.1 Tipos de interrupciones de compatibilidad 2](#_Toc327273730)

[1.2.2 Criterios de impacto 3](#_Toc327273731)

[1.2.3 Desuso del lenguaje 4](#_Toc327273732)

[2. Gramática léxica 5](#_Toc327273733)

[2.1 Caracteres y líneas 5](#_Toc327273734)

[2.1.1 Terminadores de línea 5](#_Toc327273735)

[2.1.2 Continuación de línea 5](#_Toc327273736)

[2.1.3 Espacio en blanco 7](#_Toc327273737)

[2.1.4 Comentarios 7](#_Toc327273738)

[2.2 Identificadores 7](#_Toc327273739)

[2.2.1 Caracteres de tipo 9](#_Toc327273740)

[2.3 Palabras clave 10](#_Toc327273741)

[2.4 Literales 11](#_Toc327273742)

[2.4.1 Literales booleanos 11](#_Toc327273743)

[2.4.2 Literales enteros 11](#_Toc327273744)

[2.4.3 Literales de punto flotante 12](#_Toc327273745)

[2.4.4 Literales de cadena 13](#_Toc327273746)

[2.4.5 Literales de carácter 14](#_Toc327273747)

[2.4.6 Literales de fecha 14](#_Toc327273748)

[2.4.7 Nothing 15](#_Toc327273749)

[2.5 Separadores 15](#_Toc327273750)

[2.6 Caracteres de operador 15](#_Toc327273751)

[3. Directivas de preprocesamiento 17](#_Toc327273752)

[3.1 Compilación condicional 17](#_Toc327273753)

[3.1.1 Directivas de constantes condicionales 18](#_Toc327273754)

[3.1.2 Directivas de compilación condicional 19](#_Toc327273755)

[3.2 Directivas de código fuente externo 20](#_Toc327273756)

[3.3 Directivas de región 20](#_Toc327273757)

[3.4 Directivas de suma de comprobación externa 21](#_Toc327273758)

[4. Conceptos generales 23](#_Toc327273759)

[4.1 Declaraciones 23](#_Toc327273760)

[4.1.1 Sobrecargas y firmas 24](#_Toc327273761)

[4.2 Ámbito 25](#_Toc327273762)

[4.3 Herencia 26](#_Toc327273763)

[4.3.1 Clases MustInherit y NotInheritable 27](#_Toc327273764)

[4.3.2 Interfaces y herencia múltiple 27](#_Toc327273765)

[4.3.3 Sombreado 29](#_Toc327273766)

[4.4 Implementación 34](#_Toc327273767)

[4.4.1 Método de implementación 37](#_Toc327273768)

[4.5 Polimorfismo 39](#_Toc327273769)

[4.5.1 Métodos de invalidación 40](#_Toc327273770)

[4.6 Accesibilidad 44](#_Toc327273771)

[4.6.1 Tipos constituyentes 46](#_Toc327273772)

[4.7 Nombres de tipos y de espacios de nombres 46](#_Toc327273773)

[4.7.1 Resolución de nombres calificados para los espacios de nombres y tipos 48](#_Toc327273774)

[4.7.2 Resolución de nombres no calificados para los espacios de nombres y tipos 48](#_Toc327273775)

[4.8 Variables 50](#_Toc327273776)

[4.9 Tipos y métodos genéricos 50](#_Toc327273777)

[4.9.1 Parámetros de tipo 51](#_Toc327273778)

[4.9.2 Restricciones de tipo 53](#_Toc327273779)

[4.9.3 Varianza de los parámetros de tipo 57](#_Toc327273780)

[5. Atributos 61](#_Toc327273781)

[5.1 Clases de atributos 62](#_Toc327273782)

[5.2 Bloques de atributos 64](#_Toc327273783)

[5.2.1 Nombres de atributo 65](#_Toc327273784)

[5.2.2 Argumentos de atributos 65](#_Toc327273785)

[6. Archivos de código fuente y espacios de nombres 69](#_Toc327273786)

[6.1 Inicio y terminación de un programa 69](#_Toc327273787)

[6.2 Opciones de compilación 70](#_Toc327273788)

[6.2.1 Instrucción Option Explicit 70](#_Toc327273789)

[6.2.2 Instrucción Option Strict 71](#_Toc327273790)

[6.2.3 Instrucción Option Compare 71](#_Toc327273791)

[6.2.4 Comprobaciones de desbordamiento con enteros 72](#_Toc327273792)

[6.2.5 Instrucción Option Infer 72](#_Toc327273793)

[6.3 Instrucción Imports 72](#_Toc327273794)

[6.3.1 Alias de importación 73](#_Toc327273795)

[6.3.2 Importaciones de espacios de nombres 76](#_Toc327273796)

[6.3.3 Importaciones de espacios de nombres XML 77](#_Toc327273797)

[6.4 Espacios de nombres 78](#_Toc327273798)

[6.4.1 Declaraciones de espacio de nombres 79](#_Toc327273799)

[6.4.2 Miembros de espacio de nombres 80](#_Toc327273800)

[7. Tipos 81](#_Toc327273801)

[7.1 Tipos de valor y tipos de referencia 81](#_Toc327273802)

[7.1.1 Tipos de valor que aceptan valores NULL 82](#_Toc327273803)

[7.2 Implementación de interfaces 83](#_Toc327273804)

[7.3 Tipos primitivos 84](#_Toc327273805)

[7.4 Enumeraciones 86](#_Toc327273806)

[7.4.1 Miembros de enumeraciones 86](#_Toc327273807)

[7.4.2 Valores de enumeración 86](#_Toc327273808)

[7.5 Clases 88](#_Toc327273809)

[7.5.1 Especificación de clase base 89](#_Toc327273810)

[7.5.2 Miembros de clase 90](#_Toc327273811)

[7.6 Estructuras 90](#_Toc327273812)

[7.6.1 Miembros de estructura 91](#_Toc327273813)

[7.7 Módulos estándar 92](#_Toc327273814)

[7.7.1 Miembros de módulo estándar 93](#_Toc327273815)

[7.8 Interfaces 93](#_Toc327273816)

[7.8.1 Herencia de interfaces 94](#_Toc327273817)

[7.8.2 Miembros de interfaz 96](#_Toc327273818)

[7.9 Matrices 96](#_Toc327273819)

[7.10 Delegados 99](#_Toc327273820)

[7.11 Tipos parciales 100](#_Toc327273821)

[7.12 Tipos construidos 102](#_Toc327273822)

[7.12.1 Tipos cerrados y abiertos 102](#_Toc327273823)

[7.13 Tipos especiales 103](#_Toc327273824)

[8. Conversiones 105](#_Toc327273825)

[8.1 Conversiones implícitas y explícitas 105](#_Toc327273826)

[8.2 Conversiones booleanas 106](#_Toc327273827)

[8.3 Conversiones numéricas 106](#_Toc327273828)

[8.4 Conversiones de referencias 107](#_Toc327273829)

[8.4.1 Conversiones de varianza de referencia 107](#_Toc327273830)

[8.4.2 Conversiones de delegados anónimos 109](#_Toc327273831)

[8.5 Conversiones de matrices 109](#_Toc327273832)

[8.6 Conversiones de tipo de valor 111](#_Toc327273833)

[8.6.1 Conversiones de tipos de valor que admiten null 114](#_Toc327273834)

[8.7 Conversiones de cadenas 115](#_Toc327273835)

[8.8 Conversiones widening 115](#_Toc327273836)

[8.9 Conversiones de restricción 118](#_Toc327273837)

[8.10 Conversiones de parámetros de tipo 120](#_Toc327273838)

[8.11 Conversiones definidas por el usuario 120](#_Toc327273839)

[8.11.1 Conversión widening más específica 122](#_Toc327273840)

[8.11.2 Conversión narrowing más específica 123](#_Toc327273841)

[8.12 Conversiones nativas 124](#_Toc327273842)

[8.13 Tipo dominante 124](#_Toc327273843)

[9. Miembros de tipo 125](#_Toc327273844)

[9.1 Implementación de métodos de interfaz 125](#_Toc327273845)

[9.2 Métodos 127](#_Toc327273846)

[9.2.1 Declaraciones de métodos normales, Async e Iterator 129](#_Toc327273847)

[9.2.2 Declaraciones de métodos externas 131](#_Toc327273848)

[9.2.3 Métodos invalidables 133](#_Toc327273849)

[9.2.4 Métodos compartidos 134](#_Toc327273850)

[9.2.5 Parámetros de métodos 135](#_Toc327273851)

[9.2.5.1 Parámetros de valor 135](#_Toc327273852)

[9.2.5.2 Parámetros de referencia 136](#_Toc327273853)

[9.2.5.3 Parámetros opcionales 138](#_Toc327273854)

[9.2.5.4 Parámetros ParamArray 138](#_Toc327273855)

[9.2.6 Control de eventos 139](#_Toc327273856)

[9.2.7 Métodos de extensión 141](#_Toc327273857)

[9.2.8 Métodos parciales 145](#_Toc327273858)

[9.3 Constructores 146](#_Toc327273859)

[9.3.1 Constructores de instancias 146](#_Toc327273860)

[9.3.2 Constructores compartidos 148](#_Toc327273861)

[9.4 Eventos 150](#_Toc327273862)

[9.4.1 Eventos personalizados 153](#_Toc327273863)

[9.5 Constantes 155](#_Toc327273864)

[9.6 Variables compartidas y de instancias 156](#_Toc327273865)

[9.6.1 Variables de solo lectura 158](#_Toc327273866)

[9.6.2 Variables WithEvents 159](#_Toc327273867)

[9.6.3 Inicializadores de variable 161](#_Toc327273868)

[9.6.3.1 Inicializadores regulares 163](#_Toc327273869)

[9.6.3.2 Inicializadores de objetos 163](#_Toc327273870)

[9.6.3.3 Inicializadores de tamaño de matriz 164](#_Toc327273871)

[9.6.4 Clases System.MarshalByRefObject 165](#_Toc327273872)

[9.7 Propiedades 165](#_Toc327273873)

[9.7.1 Declaraciones del descriptor de acceso Get 171](#_Toc327273874)

[9.7.2 Declaraciones del descriptor de acceso Set 172](#_Toc327273875)

[9.7.3 Propiedades predeterminadas 172](#_Toc327273876)

[9.7.4 Propiedades implementadas automáticamente 174](#_Toc327273877)

[9.7.5 Propiedades de iterador 175](#_Toc327273878)

[9.8 Operadores 175](#_Toc327273879)

[9.8.1 Operadores unarios 177](#_Toc327273880)

[9.8.2 Operadores binarios 177](#_Toc327273881)

[9.8.3 Operadores de conversión 178](#_Toc327273882)

[9.8.4 Asignación de operadores 180](#_Toc327273883)

[10. Instrucciones 181](#_Toc327273884)

[10.1 Control Flow 181](#_Toc327273885)

[10.1.1 Métodos normales 181](#_Toc327273886)

[10.1.2 Métodos Iterator 182](#_Toc327273887)

[10.1.3 Métodos asincrónicos 184](#_Toc327273888)

[10.1.4 Bloques y etiquetas 186](#_Toc327273889)

[10.1.5 Parámetros y variables locales 187](#_Toc327273890)

[10.2 Instrucciones de declaración local 189](#_Toc327273891)

[10.2.1 Declaraciones locales implícitas 192](#_Toc327273892)

[10.3 Instrucción With 192](#_Toc327273893)

[10.4 Instrucción SyncLock 193](#_Toc327273894)

[10.5 Instrucciones Event 194](#_Toc327273895)

[10.5.1 Instrucción RaiseEvent 194](#_Toc327273896)

[10.5.2 Instrucciones AddHandler y RemoveHandler 195](#_Toc327273897)

[10.6 Instrucciones de asignación 196](#_Toc327273898)

[10.6.1 Instrucciones de asignación regular 196](#_Toc327273899)

[10.6.2 Instrucciones de asignación compuesta 198](#_Toc327273900)

[10.6.3 Instrucciones de asignación Mid 198](#_Toc327273901)

[10.7 Instrucciones de invocación 199](#_Toc327273902)

[10.8 Instrucciones condicionales 200](#_Toc327273903)

[10.8.1 Instrucciones If...Then...Else 200](#_Toc327273904)

[10.8.2 Instrucciones Select...Case 201](#_Toc327273905)

[10.9 Instrucciones de bucle 203](#_Toc327273906)

[10.9.1 Instrucciones While...End While y Do...Loop 203](#_Toc327273907)

[10.9.2 Instrucciones For...Next 204](#_Toc327273908)

[10.9.3 Instrucciones For Each...Next 206](#_Toc327273909)

[10.10 Instrucciones de control de excepciones 210](#_Toc327273910)

[10.10.1 Instrucciones de control estructurado de excepciones 210](#_Toc327273911)

[10.10.1.1 Bloques Finally 211](#_Toc327273912)

[10.10.1.2 Bloques Catch 211](#_Toc327273913)

[10.10.1.3 Instrucción Throw 212](#_Toc327273914)

[10.10.2 Instrucciones de control no estructurado de excepciones 213](#_Toc327273915)

[10.10.2.1 Instrucción Error 214](#_Toc327273916)

[10.10.2.2 Instrucción On Error 214](#_Toc327273917)

[10.10.2.3 Instrucción Resume 214](#_Toc327273918)

[10.11 Instrucciones de bifurcación 215](#_Toc327273919)

[10.12 Instrucciones de control de matrices 216](#_Toc327273920)

[10.12.1 Instrucción ReDim 216](#_Toc327273921)

[10.12.2 Instrucción Erase 217](#_Toc327273922)

[10.13 Instrucción Using 218](#_Toc327273923)

[10.14 Instrucciones Await 219](#_Toc327273924)

[10.15 Instrucciones yield 219](#_Toc327273925)

[11. Expresiones 221](#_Toc327273926)

[11.1 Clasificación de expresiones 221](#_Toc327273927)

[11.1.1 Reclasificación de expresiones 222](#_Toc327273928)

[11.2 Expresiones constantes 225](#_Toc327273929)

[11.3 Expresiones enlazadas en tiempo de ejecución 226](#_Toc327273930)

[11.4 Expresiones simples 228](#_Toc327273931)

[11.4.1 Expresiones literales 228](#_Toc327273932)

[11.4.2 Expresiones entre paréntesis 228](#_Toc327273933)

[11.4.3 Expresiones de instancia 228](#_Toc327273934)

[11.4.4 Expresiones de nombre simple 228](#_Toc327273935)

[11.4.5 Expresiones AddressOf 230](#_Toc327273936)

[11.5 Expresiones de tipo 231](#_Toc327273937)

[11.5.1 Expresiones GetType 231](#_Toc327273938)

[11.5.2](#_Toc327273939) *[CommaList](#_Toc327273939)* [::=](#_Toc327273939) *[Comma](#_Toc327273939)* [|](#_Toc327273939) *[CommaList](#_Toc327273939)**[Comma](#_Toc327273939)*[Expresiones TypeOf...Is 232](#_Toc327273939)

[11.5.3 Expresiones Is 232](#_Toc327273940)

[11.5.4 Expresiones GetXmlNamespace 232](#_Toc327273941)

[11.6 Expresiones de acceso a miembros 233](#_Toc327273942)

[11.6.1 Nombres de miembros y de tipos idénticos 237](#_Toc327273943)

[11.6.2 Instancias predeterminadas 237](#_Toc327273944)

[11.6.2.1 Instancias predeterminadas y nombres de tipos 238](#_Toc327273945)

[11.6.2.2 Clases de grupo 239](#_Toc327273946)

[11.6.3 Recopilación de métodos de extensión 240](#_Toc327273947)

[11.7 Expresiones de acceso a miembros de diccionario 244](#_Toc327273948)

[11.8 Expresiones de invocación 244](#_Toc327273949)

[11.8.1 Resolución de métodos sobrecargados 245](#_Toc327273950)

[11.8.1.1 Especificidad de los miembros o los tipos dada una lista de argumentos 253](#_Toc327273951)

[11.8.1.2 Calidad de genéricos 254](#_Toc327273952)

[11.8.1.3 Nivel de generalidad 255](#_Toc327273953)

[11.8.2 Aplicabilidad a lista de argumentos 255](#_Toc327273954)

[11.8.3 Pasar y elegir argumentos para parámetros opcionales 257](#_Toc327273955)

[11.8.4 Métodos condicionales 259](#_Toc327273956)

[11.8.5 Inferencia de argumentos de tipo 259](#_Toc327273957)

[11.9 Expresiones de índice 261](#_Toc327273958)

[11.10 Expresiones New 262](#_Toc327273959)

[11.10.1 Expresiones de creación de objetos 263](#_Toc327273960)

[11.10.2 Expresiones de matriz 265](#_Toc327273961)

[11.10.2.1 Expresiones de creación de matrices 265](#_Toc327273962)

[11.10.2.2 Literales de matriz 267](#_Toc327273963)

[11.10.3 Expresiones de creación de delegados 267](#_Toc327273964)

[11.10.4 Expresiones de creación de objetos anónimos 270](#_Toc327273965)

[11.11 Expresiones de conversión 273](#_Toc327273966)

[11.12 Expresiones de operador 274](#_Toc327273967)

[11.12.1 Prioridad y asociatividad de los operadores 275](#_Toc327273968)

[11.12.2 Operandos de objeto 275](#_Toc327273969)

[11.12.3 Resolución de operadores 276](#_Toc327273970)

[11.13 Operadores aritméticos 279](#_Toc327273971)

[11.13.1 Operador unario de signo más 279](#_Toc327273972)

[11.13.2 Operador unario de signo menos 279](#_Toc327273973)

[11.13.3 Operador de suma 280](#_Toc327273974)

[11.13.4 Operador de resta 281](#_Toc327273975)

[11.13.5 Operador de multiplicación 282](#_Toc327273976)

[11.13.6 Operadores de división 282](#_Toc327273977)

[11.13.7 Mod (Operador) 285](#_Toc327273978)

[11.13.8 Operador de exponenciación 285](#_Toc327273979)

[11.14 Operadores relacionales 286](#_Toc327273980)

[11.15 Like (Operador) 288](#_Toc327273981)

[11.16 Operador de concatenación 289](#_Toc327273982)

[11.17 Operadores lógicos 290](#_Toc327273983)

[11.17.1 Cortocircuitar las operaciones lógicas 292](#_Toc327273984)

[11.18 Operadores de desplazamiento 293](#_Toc327273985)

[11.19 Expresiones booleanas 294](#_Toc327273986)

[11.20 Expresiones lambda 295](#_Toc327273987)

[11.20.1 Cierres 299](#_Toc327273988)

[11.21 Expresiones de consulta 301](#_Toc327273989)

[11.21.1 Variables de intervalo 302](#_Toc327273990)

[11.21.2 Tipos consultables 303](#_Toc327273991)

[11.21.3 Indizador de consultas predeterminado 305](#_Toc327273992)

[11.21.4 Operador de consulta From 305](#_Toc327273993)

[11.21.5 Operador de consulta Join 307](#_Toc327273994)

[11.21.6 Operador de consulta Let 308](#_Toc327273995)

[11.21.7 Operador de consulta Select 308](#_Toc327273996)

[11.21.8 Operador de consulta Distinct 309](#_Toc327273997)

[11.21.9 Operador de consulta Where 310](#_Toc327273998)

[11.21.10 Operadores de consulta de partición 311](#_Toc327273999)

[11.21.11 Operador de consulta Order By 311](#_Toc327274000)

[11.21.12 Operador de consulta Group By 313](#_Toc327274001)

[11.21.13 Operador de consulta Aggregate 315](#_Toc327274002)

[11.21.14 Operador de consulta Group Join 316](#_Toc327274003)

[11.22 Expresiones condicionales 317](#_Toc327274004)

[11.23 Expresiones literales XML 317](#_Toc327274005)

[11.23.1 Reglas léxicas 318](#_Toc327274006)

[11.23.2 Expresiones incrustadas 319](#_Toc327274007)

[11.23.3 Documentos XML 319](#_Toc327274008)

[11.23.4 Elementos XML 321](#_Toc327274009)

[11.23.5 Espacios de nombres XML 323](#_Toc327274010)

[11.23.6 Instrucciones de procesamiento XML 324](#_Toc327274011)

[11.23.7 Comentarios XML 325](#_Toc327274012)

[11.23.8 Secciones CDATA 325](#_Toc327274013)

[11.24 Expresiones de acceso a miembros XML 325](#_Toc327274014)

[11.25 Operador Await 327](#_Toc327274015)

[12. Comentarios de documentación 331](#_Toc327274016)

[12.1 Formato de comentario de documentación 331](#_Toc327274017)

[12.2 Etiquetas recomendadas 332](#_Toc327274018)

[12.2.1 <c> 332](#_Toc327274019)

[12.2.2 <code> 333](#_Toc327274020)

[12.2.3 <example> 333](#_Toc327274021)

[12.2.4 <exception> 333](#_Toc327274022)

[12.2.5 <include> 334](#_Toc327274023)

[12.2.6 <list> 334](#_Toc327274024)

[12.2.7 <para> 335](#_Toc327274025)

[12.2.8 <param> 335](#_Toc327274026)

[12.2.9 <paramref> 335](#_Toc327274027)

[12.2.10 <permission> 336](#_Toc327274028)

[12.2.11 <remarks> 336](#_Toc327274029)

[12.2.12 <returns> 336](#_Toc327274030)

[12.2.13 <see> 337](#_Toc327274031)

[12.2.14 <seealso> 337](#_Toc327274032)

[12.2.15 <summary> 337](#_Toc327274033)

[12.2.16 <typeparam> 338](#_Toc327274034)

[12.2.17 <value> 338](#_Toc327274035)

[12.3 Cadenas id. 338](#_Toc327274036)

[12.3.1 Ejemplos de cadena de identificador 339](#_Toc327274037)

[12.4 Ejemplo de comentario de documentación 342](#_Toc327274038)

[13. Resumen de la gramática 349](#_Toc327274039)

[13.1 Gramática léxica 349](#_Toc327274040)

[13.1.1 Caracteres y líneas 349](#_Toc327274041)

[13.1.2 Identificadores 350](#_Toc327274042)

[13.1.3 Palabras clave 350](#_Toc327274043)

[13.1.4 Literales 351](#_Toc327274044)

[13.2 Directivas de preprocesamiento 353](#_Toc327274045)

[13.2.1 Compilación condicional 353](#_Toc327274046)

[13.2.2 Directivas de código fuente externo 354](#_Toc327274047)

[13.2.3 Directivas de región 354](#_Toc327274048)

[13.2.4 Directivas de suma de comprobación externa 354](#_Toc327274049)

[13.3 Gramática sintáctica 354](#_Toc327274050)

[13.3.1 Atributos 355](#_Toc327274051)

[13.3.2 Archivos de código fuente y espacios de nombres 355](#_Toc327274052)

[13.3.3 Tipos 357](#_Toc327274053)

[13.3.4 Miembros de tipo 360](#_Toc327274054)

[13.3.5 Instrucciones 365](#_Toc327274055)

[13.3.6 Expresiones 370](#_Toc327274056)

[14. Lista de cambios 381](#_Toc327274057)

[14.1 Versión 7.1 a versión 8.0 381](#_Toc327274058)

[14.1.1 Cambios importantes 381](#_Toc327274059)

[14.1.2 Cambios menores 382](#_Toc327274060)

[14.1.3 Aclaraciones y erratas 383](#_Toc327274061)

[14.1.4 Varios 385](#_Toc327274062)

[14.2 Versión 8.0 a versión 8.0 (2](#_Toc327274063)[nd](#_Toc327274063) [Edition) 386](#_Toc327274063)

[14.2.1 Cambios menores 386](#_Toc327274064)

[14.2.2 Aclaraciones y erratas 387](#_Toc327274065)

[14.2.3 Varios 388](#_Toc327274066)

[14.3 Versión 8.0 (2](#_Toc327274067)[nd](#_Toc327274067) [Edition) a versión 9.0 388](#_Toc327274067)

[14.3.1 Cambios importantes 388](#_Toc327274068)

[14.3.2 Cambios menores 388](#_Toc327274069)

[14.3.3 Aclaraciones y erratas 389](#_Toc327274070)

[14.3.4 Varios 390](#_Toc327274071)

[14.4 Versión 9.0 a versión 10.0 390](#_Toc327274072)

[14.4.1 Cambios importantes 390](#_Toc327274073)

[14.4.2 Cambios menores 390](#_Toc327274074)

[14.4.3 Aclaraciones y erratas 391](#_Toc327274075)

[14.5 Versión 10.0 a versión 11.0 393](#_Toc327274076)

[14.5.1 Cambios importantes 393](#_Toc327274077)

[14.5.2 Cambios menores 393](#_Toc327274078)

[14.5.3 Aclaraciones y erratas 394](#_Toc327274079)

# Introduction

Microsoft® Visual Basic® es un lenguaje de programación de alto nivel para Microsoft .NET Framework. Si bien se ha diseñado para ser un lenguaje fácil de aprender y accesible, también tiene la potencia suficiente para satisfacer las necesidades de los programadores expertos. El lenguaje de programación Visual Basic cuenta con una sintaxis similar al inglés, lo que facilita la claridad y legibilidad de su código. Siempre que es posible, se usan palabras o frases con significado en lugar de abreviaturas, acrónimos o caracteres especiales. Aunque se suele admitir la sintaxis superflua o innecesaria, no es preciso usarla.

El lenguaje de programación Visual Basic puede estar fuertemente tipado o no. El tipado débil difiere las pesadas tareas de comprobación de tipos hasta que el programa está ya en ejecución. Esto incluye no solo la comprobación de los tipos de las conversiones sino también de las llamadas de métodos, lo que significa que el enlace de una llamada de método se puede retrasar hasta el momento de la ejecución. Esto resulta útil cuando se compilan prototipos u otros programas en los que la velocidad de desarrollo es más importante que la velocidad de ejecución. El lenguaje de programación Visual Basic también proporciona semántica fuertemente tipada que realiza toda la comprobación de tipos en el momento de la compilación y anula el enlace en tiempo de ejecución de las llamadas de métodos. De esta forma consigue el máximo rendimiento y sirve de ayuda para garantizar que las conversiones de tipo son correctas. Esto resulta útil cuando se compilan aplicaciones de producción en las que es importante la velocidad de ejecución y su exactitud.

Este documento describe el lenguaje Visual Basic. Está pensado como descripción completa del lenguaje y no para ser usado como tutorial sobre este ni como manual de referencia del usuario.

## Notación gramatical

Esta especificación describe dos gramáticas: una gramática léxica y una gramática sintáctica. La gramática léxica define cómo se combinan los caracteres para formar tokens; la gramática sintáctica define cómo se pueden combinar los tokens para formar programas de Visual Basic. También existen varias gramáticas secundarias que se usan en las operaciones de preprocesamiento, como la compilación condicional.

Nota   Las gramáticas de esta especificación están diseñadas para ser inteligibles y no formales (es decir, utilizables por LEX o YACC).

Todas las gramáticas utilizan una notación BNF modificada, que consta de un conjunto de producciones constituidas por nombres terminales y no terminales. Los nombres terminales representan uno o varios caracteres Unicode. Los no terminales se definen en una o varias producciones. En las producciones, los nombres no terminales se muestran en cursiva y los terminales en un fixed-width type. El texto con un tipo normal delimitado por metasímbolos de corchete angular representa a elementos terminales informales (por ejemplo, "<todos los caracteres Unicode>"). Cada gramática comienza con el nombre no terminal Start.

La distinción entre mayúsculas y minúsculas carece de importancia en los programas de Visual Basic. Por motivos de simplicidad los elementos terminales se indican con las mayúsculas y minúsculas estándar, pero se puede usar cualquier combinación. Los elementos terminales que son a la vez elementos imprimibles del juego de caracteres ASCII se representan mediante sus caracteres ASCII correspondientes. Visual Basic tampoco distingue anchos a la hora de establecer correspondencias entre elementos terminales y permite que los caracteres Unicode de ancho completo coincidan con sus equivalentes Unicode de ancho medio, pero solo en tokens completos. Un token no coincidirá si contiene una mezcla de caracteres de ancho medio y ancho completo.

Un conjunto de producciones comienza con el nombre de un elemento no terminal, seguido de dos puntos y un signo igual. El lado derecho contiene una producción terminal o no terminal. Un elemento no terminal puede incluir varias producciones separadas por el metasímbolo de barra vertical (|). Los elementos que aparecen incluidos entre metasímbolos de corchete angular ([]) son opcionales. El metasímbolo más (+) a continuación de un elemento indica que este puede aparecer una o varias veces.

Se puede agregar sangrado y saltos de línea para mejorar la legibilidad, pero no forman parte de la producción.

## Compatibilidad

Una importante característica de un lenguaje de programación es la compatibilidad entre sus diferentes versiones. Si una versión más reciente de un lenguaje no acepta el mismo código que una versión anterior o lo interpreta de forma diferente que en la versión anterior, el programador puede sentirse abrumado a la hora de actualizar su código de una versión del lenguaje a otra. Por tanto, se debe mantener la compatibilidad entre las versiones, excepto si las ventajas para los consumidores del lenguaje son claras e inapelables.

Los cambios entre versiones del lenguaje Visual Basic están regidos por la siguiente directiva. El término lenguaje, cuando se usa en este contexto, solo hace referencia a los aspectos sintácticos y semánticos del propio lenguaje Visual Basic y no afecta a ninguna clase de .NET Framework incluida como parte del espacio de nombres Microsoft.VisualBasic (y sus subespacios). Las clases de .NET Framework se cubren en una directiva de compatibilidad y control de versiones independiente que se encuentra fuera del ámbito de este documento.

### Tipos de interrupciones de compatibilidad

En un mundo ideal, habría una compatibilidad del 100% entre la versión existente de Visual Basic y todas sus versiones futuras. Sin embargo, puede haber situaciones en las que la necesidad de una interrupción de la compatibilidad pueda ser mayor que el coste que supondría para los programadores. Estas situaciones son:

Nuevas advertencias. La incorporación de nuevas advertencias no supone, en sí misma, una interrupción de la compatibilidad. Sin embargo, dado que muchos desarrolladores realizan la compilación con la opción "Tratar advertencias como errores" activada, se deben extremar las precauciones al incluir nuevas advertencias.

Nuevas palabras clave. Al incluir nuevas características del lenguaje puede ser necesarias nuevas palabras clave. Se realizará un esfuerzo razonable para elegir palabras clave que reduzcan al mínimo la posibilidad de conflicto con los identificadores de los usuarios y para usar palabras clave existentes donde sea razonable. Se proporcionará ayuda para actualizar proyectos de versiones anteriores y evitar las nuevas palabras clave.

Errores del compilador. Si el comportamiento del compilador no concuerda con el comportamiento documentado en la especificación del lenguaje, puede que resulte necesario corregir el comportamiento del compilador para que se ajuste al documentado.

Error de especificación. Si el comportamiento del compilador es coherente con la especificación del lenguaje pero esta es claramente incorrecta, puede que resulte necesario cambiar la especificación del lenguaje y el comportamiento del compilador. La expresión "claramente incorrecta" significa que el comportamiento documentado es contrario al que esperarían una mayoría clara e inequívoca de usuarios y supone un comportamiento nada deseado para los usuarios.

Ambigüedad de la especificación. Si la especificación del lenguaje debe explicar con detalle lo que sucede en una situación determinada y no lo hace, y el compilador controla la situación de forma incoherente o claramente incorrecta (según la definición del punto anterior), puede que sea necesario aclarar la especificación y corregir el comportamiento del compilador. Dicho de otro modo, si la especificación cubre los casos a, b, d y e, pero no menciona qué sucede en el caso c, y el compilador se comporta de forma incorrecta en el caso c, es posible que resulte necesario documentar qué sucede en el caso c y cambiar el comportamiento del compilador para que se ajuste a la especificación. (Si la especificación es ambigua en cuanto a explicar qué sucede en una situación pero el compilador se comporta de una manera que no es claramente incorrecta, este comportamiento se convierte en la especificación de facto.)

Convertir los errores en tiempo de ejecución en errores en tiempo de compilación. En los casos en que hay una total certeza de que se producirá un error de código en tiempo de ejecución (es decir, si el código de usuario incluye un error inequívoco en él), puede que convenga agregar un error en tiempo de compilación para detectar esta situación.

Omisión de especificación. Si la especificación del lenguaje no permite o impide de forma concreta una situación determinada y el compilador controla la situación de forma no deseada (si el comportamiento del compilador fuese claramente incorrecto, implicaría un error de la especificación, no una omisión), puede que resulte necesario aclarar la especificación y cambiar el comportamiento del compilador. Además del análisis de impacto habitual, los cambios de este tipo se limitan a los casos en los que su impacto se considera mínimo y su ventaja para los desarrolladores extremadamente alta.

Nuevas características. En general, la incorporación de nuevas características no debe cambiar las partes existentes de la especificación del lenguaje ni el comportamiento existente del compilador. Si la incorporación de nuevas características requiere el cambio de la especificación del lenguaje existente, esta interrupción de la compatibilidad solo se considera razonable si el impacto es mínimo y las ventajas de la característica son importantes.

Seguridad. En casos extraordinarios, las cuestiones de seguridad pueden exigir una interrupción de la compatibilidad, como la eliminación o la modificación de una característica inherentemente insegura y que implica un evidente riesgo de seguridad para los usuarios.

Los siguientes casos no se consideran razones aceptables para dar lugar a una interrupción de la compatibilidad:

Comportamiento no deseado o deplorable. Un diseño del lenguaje o un comportamiento del compilador que resulten razonables pero se consideren no deseados o deplorables de forma retrospectiva, no justifican una interrupción de la compatibilidad con las versiones anteriores. En estos casos se debe utilizar el proceso de desuso del lenguaje, que se trata más adelante.

Todo lo demás. Por lo demás, el comportamiento del compilador mantiene su compatibilidad con las versiones anteriores.

### Criterios de impacto

A la hora de plantear si resultaría aceptable una interrupción de la compatibilidad, se utilizan diferentes criterios para determinar el impacto del cambio. Cuanto mayor sea el impacto, más alto será el listón para aceptar la interrupción de compatibilidad.

A continuación se indican estos criterios:

¿Cuál es el ámbito del cambio? En otras palabras, ¿cuántos programas se pueden ver afectados? ¿Cuántos usuarios se pueden ver afectados? ¿Con qué frecuencia se deberá escribir código al que afecta el cambio?

¿Existen soluciones alternativas para conseguir el mismo comportamiento antes de realizar el cambio?

¿Es obvio el cambio? ¿Advertirán los usuarios el cambio al instante o simplemente se ejecutarán sus programas de forma diferente?

¿Se puede solucionar el cambio razonablemente durante una actualización? ¿Es posible escribir una herramienta que localice los casos en los que se produce el cambio con una precisión perfecta y que cambie el código para solucionar la situación?

¿Cuáles son los comentarios de la comunidad sobre el cambio?

### Desuso del lenguaje

Con el paso del tiempo, es posible que dejen de usarse algunas partes del lenguaje o del compilador. Como se ha indicado anteriormente, no resulta aceptable interrumpir la compatibilidad para quitar las características desusadas. En su lugar, se deben seguir los pasos que se indican a continuación:

Para una característica que existe en la versión A de Visual Studio, se deben solicitar comentarios de la comunidad de usuarios sobre el desuso de la característica y se debe proporcionar información detallada antes de adoptar una decisión final sobre el desuso. El proceso de desuso se puede revocar o abandonar en cualquier momento en función de los comentarios de la comunidad.

Se debe publicar una versión completa (es decir, no un lanzamiento) B de Visual Studio con advertencias del compilador que avisen sobre el desuso. Las advertencias deben estar activadas de forma predeterminada y se pueden desactivar. Los desusos se deben documentar claramente en la documentación del producto y en la Web.

Se debe publicar una versión completa C de Visual Studio con advertencias del compilador que no se puedan desactivar.

Posteriormente se debe publicar una versión completa D de Visual Studio con las advertencias del compilador sobre el desuso convertidas en errores del compilador. La publicación de D debe producirse tras el final de la fase de soporte general (5 años a fecha de este documento) de la versión A.

Por último, se puede publicar una versión E de Visual Studio que quite los errores de compilador.

No se permitirán los cambios que no se puedan adaptar a este marco de desuso.

# Gramática léxica

La compilación de un programa de Visual Basic supone en primer lugar la conversión de la secuencia sin procesar de caracteres Unicode en un conjunto ordenado de tokens léxicos. Dado que el lenguaje Visual Basic no tiene un formato libre, el conjunto de tokens se subdivide en una serie de líneas lógicas. La línea lógica se extiende desde el inicio de la secuencia o desde un terminador de línea hasta el siguiente terminador de línea que no esté precedido por una continuación de línea o hasta el final de la secuencia.

Nota Con la incorporación de expresiones literales XML en la versión 9.0 del lenguaje, Visual Basic ya no cuenta con una gramática léxica exclusiva, en el sentido de que el código de Visual Basic se puede convertir en tokens independientemente del contexto sintáctico. Esto se debe al hecho de que XML y Visual Basic cuentan con reglas léxicas diferentes y el conjunto de reglas léxicas que se usan en un momento determinado depende de la construcción sintáctica que se procesa en dicho momento. Esta especificación conserva la sección de gramática léxica como guía para las reglas léxicas del código normal de Visual Basic. Sin embargo, a largo plazo es probable que las reglas léxicas se combinen con las reglas sintácticas.

Start ::= [ LogicalLine+ ]

LogicalLine ::= [ LogicalLineElement+ ] [ Comment ] LineTerminator

LogicalLineElement ::= WhiteSpace | LineContinuation | Token

Token ::= Identifier | Keyword | Literal | Separator | Operator

## Caracteres y líneas

Los programas de Visual Basic están compuestos por caracteres del juego de caracteres Unicode.

Character ::= < any Unicode character except a LineTerminator >

### Terminadores de línea

Los caracteres de salto de línea de Unicode separan líneas lógicas.

LineTerminator ::=  
 < Unicode carriage return character (0x000D) > |  
 < Unicode linefeed character (0x000A) > |  
 < Unicode carriage return character > < Unicode linefeed character > |  
 < Unicode line separator character (0x2028) > |  
 < Unicode paragraph separator character (0x2029) >

### Continuación de línea

Una continuación de línea consta al menos de un carácter de espacio en blanco inmediatamente anterior a un único carácter de subrayado como último carácter (distinto del espacio en blanco) de una línea de texto. La continuación de línea permite que las líneas lógicas abarquen más de una única línea física. Las continuaciones de línea se tratan como si fuesen espacios en blanco, aunque no lo son.

El siguiente programa muestra algunas continuaciones de línea:

Module Test  
 Sub Print( \_  
 Param1 As Integer, \_  
 Param2 As Integer )  
  
 If (Param1 < Param2) Or \_  
 (Param1 > Param2) Then  
 Console.WriteLine("Not equal")  
 End If  
 End Function  
End Module

Algunas posiciones de la gramática sintáctica permiten las continuaciones de línea implícitas. Cuando se encuentra un terminador de línea:

tras una coma (,), un paréntesis de apertura ((), una llave de apertura ({) o una expresión incrustada de apertura (<%=)

tras un calificador de miembro (. o .@ o ...), siempre que se califique algo (es decir, siempre que no se use un contexto With implícito)

antes de un paréntesis de cierre ()), una llave de cierre (}) o una expresión incrustada de cierre (%>)

después de un carácter menor que (<) en un contexto de atributo

antes de un carácter mayor que (>) en un contexto de atributo

después de un carácter mayor que (>) en un contexto de atributo que no esté en el nivel de archivo

antes y después de operadores de consulta (Where, Order, Select, etc.)

después de operadores binarios (+, -, /, \*, etc.) en un contexto de expresión

después de operadores de asignación (=, :=, +=, -=, etc.) en cualquier contexto

el terminador de línea se trata como continuación de línea. El ejemplo anterior también se puede escribir de la siguiente forma:

Module Test  
 Sub Print(  
 Param1 As Integer,  
 Param2 As Integer)  
  
 If (Param1 < Param2) Or  
 (Param1 > Param2) Then  
 Console.WriteLine("Not equal")  
 End If  
 End Function  
End Module

Las continuaciones de línea implícitas solo se deducirán directamente antes o después del token especificado. No se deducirán antes o después de una continuación de línea. Por ejemplo:

Dim y = 10  
' Error: Expression expected for assignment to x  
Dim x = \_  
  
y

No se deducirán continuaciones de línea en contextos de compilación condicional.

Anotación

Esta última restricción es necesaria ya que el texto de los bloques de compilación condicional que no se compilan no tiene por qué ser sintácticamente válido. La instrucción de compilación condicional podría utilizar involuntariamente el texto del bloque, en especial a medida que el lenguaje se amplía en el futuro.

LineContinuation ::= WhiteSpace \_ [ WhiteSpace+ ] LineTerminator

Comma ::= , [ LineTerminator ]

Period ::= . [ LineTerminator ]

OpenParenthesis ::= ( [ LineTerminator ]

CloseParenthesis ::= [ LineTerminator ] )

OpenCurlyBrace ::= { [ LineTerminator ]

CloseCurlyBrace ::= [ LineTerminator ] }

Equals ::= = [ LineTerminator ]

ColonEquals ::= : = [ LineTerminator ]

### Espacio en blanco

El único uso del espacio en blanco es separar los tokens; en los demás casos, se pasa por alto. Las líneas lógicas que solo contienen espacio en blanco se omiten.

Nota   Los terminadores de línea no se consideran espacios en blanco.

WhiteSpace ::=  
 < Unicode blank characters (class Zs) > |  
 < Unicode tab character (0x0009) >

### Comentarios

Un comentario comienza con un carácter de comilla simple o con la palabra clave REM. El carácter de comilla simple puede ser el carácter de comilla simple de ASCII o los caracteres de comilla simple izquierda y derecha de Unicode. Los comentarios pueden comenzar en cualquier posición de una línea de código fuente; el final de la línea física termina el comentario. El compilador omite los caracteres situados entre el comienzo del comentario y el terminador de línea. Por tanto, los comentarios no se pueden extender en varias líneas mediante continuaciones de línea.

Comment ::= CommentMarker [ Character+ ]

CommentMarker ::= SingleQuoteCharacter | REM

SingleQuoteCharacter ::=  
 ' |  
 < Unicode left single-quote character (0x2018) > |  
 < Unicode right single-quote character (0x2019) >

## Identificadores

Un identificador es un nombre. Los identificadores de Visual Basic cumplen el Anexo 15 del estándar Unicode con una excepción: pueden comenzar con un carácter de subrayado (conector). Si un identificador comienza con un carácter de subrayado, debe contener al menos otro carácter de identificador válido para eliminar la ambigüedad con una continuación de línea.

Los identificadores normales no pueden coincidir con las palabras clave, pero sí pueden hacerlo los identificadores de escape y los identificadores con un carácter de tipo. Los identificadores de escape son identificadores delimitados por corchetes. Estos identificadores siguen las mismas reglas que los identificadores normales, excepto que pueden coincidir con palabras clave y no pueden tener caracteres de tipo.

Este ejemplo define una clase denominada class con un método compartido denominado shared que toma un parámetro denominado boolean y, a continuación, llama al método.

Class [class]  
 Shared Sub [shared]([boolean] As Boolean)  
 If [boolean] Then  
 Console.WriteLine("true")  
 Else  
 Console.WriteLine("false")  
 End If  
 End Sub  
End Class  
  
Module [module]  
 Sub Main()  
 [class].[shared](True)  
 End Sub  
End Module

Los identificadores no distinguen entre mayúsculas y minúsculas, por lo que dos identificadores se consideran el mismo si solo se distinguen por este hecho.

Nota   Cuando se comparan identificadores, se usa la asignación uno a uno del estándar Unicode y se omiten las asignaciones de mayúsculas y minúsculas específicas de la configuración regional.

Identifier ::=  
 NonEscapedIdentifier [ TypeCharacter ] |  
 Keyword TypeCharacter |  
 EscapedIdentifier

NonEscapedIdentifier ::= < IdentifierName but not Keyword >

EscapedIdentifier ::= [ IdentifierName ]

IdentifierName ::= IdentifierStart [ IdentifierCharacter+ ]

IdentifierStart ::=  
 AlphaCharacter |  
 UnderscoreCharacter IdentifierCharacter

IdentifierCharacter ::=  
 UnderscoreCharacter |  
 AlphaCharacter |  
 NumericCharacter |  
 CombiningCharacter |  
 FormattingCharacter

AlphaCharacter ::=  
 < Unicode alphabetic character (classes Lu, Ll, Lt, Lm, Lo, Nl) >

NumericCharacter ::= < Unicode decimal digit character (class Nd) >

CombiningCharacter ::= < Unicode combining character (classes Mn, Mc) >

FormattingCharacter ::= < Unicode formatting character (class Cf) >

UnderscoreCharacter ::= < Unicode connection character (class Pc) >

IdentifierOrKeyword ::= Identifier | Keyword

### Caracteres de tipo

Los caracteres de tipo indican el tipo del identificador anterior. El carácter de tipo no se considera parte del identificador. Si una declaración incluye un carácter de tipo, este debe coincidir con el tipo especificado en la propia declaración; en caso contrario, se produce un error en tiempo de compilación. Si la declaración omite el tipo (por ejemplo, si no especifica una cláusula As), el carácter de tipo se usa implícitamente como tipo de la declaración.

No puede existir ningún espacio en blanco entre un identificador y su carácter de tipo. No existen caracteres de tipo para Byte, SByte, UShort, Short, UInteger o ULong, debido a una falta de caracteres adecuados.

Si se agrega un carácter de tipo a un identificador que conceptualmente no dispone de tipo (por ejemplo, un nombre de espacio de nombres) o a un identificador cuyo tipo no coincide con el del carácter de tipo, se produce un error en tiempo de compilación.

En el ejemplo siguiente se muestra el uso de los caracteres de tipo:

' The follow line will cause an error: standard modules have no type.  
Module Test1#  
End Module  
  
Module Test2  
  
 ' This function takes a Long parameter and returns a String.  
 Function Func$(Param&)  
  
 ' The following line causes an error because the type character  
 ' conflicts with the declared type of Func and Param.  
 Func# = CStr(Param@)  
  
 ' The following line is valid.  
 Func$ = CStr(Param&)  
 End Function  
End Module

El carácter de tipo ! presenta un problema especial, ya que se puede usar como carácter de tipo y como separador en este lenguaje. Para evitar la ambigüedad, el carácter ! se considera carácter de tipo siempre que el carácter que le siga no pueda iniciar un identificador. Si puede, el carácter ! se considera separador y no carácter de tipo.

TypeCharacter ::=  
 IntegerTypeCharacter |  
 LongTypeCharacter |  
 DecimalTypeCharacter |  
 SingleTypeCharacter |  
 DoubleTypeCharacter |  
 StringTypeCharacter

IntegerTypeCharacter ::= %

LongTypeCharacter ::= &

DecimalTypeCharacter ::= @

SingleTypeCharacter ::= !

DoubleTypeCharacter ::= #

StringTypeCharacter ::= $

## Palabras clave

Una palabra clave es una palabra con un significado especial en una construcción del lenguaje. Todas las palabras clave se consideran reservadas por el lenguaje y no se pueden usar como identificadores a menos que se trate de identificadores de escape.

Nota   EndIf, GoSub, Let, Variant y Wend se conservan como palabras clave aunque ya no se usan en Visual Basic.

Keyword ::= < member of keyword table >

|  |  |  |  |
| --- | --- | --- | --- |
| AddHandler | AddressOf | Alias | And |
| AndAlso | As | Boolean | ByRef |
| Byte | ByVal | Call | Case |
| Catch | CBool | CByte | CChar |
| CDate | CDbl | CDec | Char |
| CInt | Class | CLng | CObj |
| Const | Continue | CSByte | CShort |
| CSng | CStr | CType | CUInt |
| CULng | CUShort | Date | Decimal |
| Declare | Default | Delegate | Dim |
| DirectCast | Do | Double | Each |
| Else | ElseIf | End | EndIf |
| Enum | Erase | Error | Event |
| Exit | False | Finally | For |
| Friend | Function | Get | GetType |
| GetXmlNamespace | Global | GoSub | GoTo |
| Handles | If | Implements | Imports |
| In | Inherits | Integer | Interface |
| Is | IsNot | Let | Lib |
| Like | Long | Loop | Me |
| Mod | Module | MustInherit | MustOverride |
| MyBase | MyClass | Namespace | Narrowing |
| New | Next | Not | Nothing |
| NotInheritable | NotOverridable | Object | Of |
| On | Operator | Option | Optional |
| Or | OrElse | Overloads | Overridable |
| Overrides | ParamArray | Partial | Private |
| Property | Protected | Public | RaiseEvent |
| ReadOnly | ReDim | REM | RemoveHandler |
| Resume | Return | SByte | Select |
| Set | Shadows | Shared | Short |
| Single | Static | Step | Stop |
| String | Structure | Sub | SyncLock |
| Then | Throw | To | True |
| Try | TryCast | TypeOf | UInteger |
| ULong | UShort | Using | Variant |
| Wend | When | While | Widening |
| With | WithEvents | WriteOnly | Xor |

## Literales

Un literal es una representación textual de un valor determinado de un tipo. Los tipos literales incluyen booleano, entero, punto flotante, cadena, carácter y fecha.

Literal ::=  
 BooleanLiteral |  
 IntegerLiteral |  
 FloatingPointLiteral |  
 StringLiteral |  
 CharacterLiteral |  
 DateLiteral |  
 Nothing

### Literales booleanos

True y False son literales del tipo Boolean que se asignan a los estados verdadero y falso, respectivamente.

BooleanLiteral ::= True | False

### Literales enteros

Los literales enteros pueden ser decimales (base 10), hexadecimales (base 16) u octales (base 8). Los literales enteros decimales son cadenas de dígitos decimales (0-9). Un literal hexadecimal es &H seguido de una cadena de dígitos hexadecimales (0-9, A-F). Un literal octal es &O seguido de una cadena de dígitos octales (0-7). Los literales decimales representan directamente el valor decimal del literal entero, en tanto que los literales octales y hexadecimales representan su valor binario (es decir, &H8000S es –32768, no un error de desbordamiento).

El tipo de un literal se determina a partir de su valor o del carácter de tipo siguiente. Si no se especifica ningún carácter de tipo, los valores del rango del tipo Integer tienen el tipo Integer; los valores fuera del rango de Integer tienen el tipo Long. Si el tipo de un literal entero no es suficiente para incluir el literal, se produce un error en tiempo de compilación.

Anotación

No existe un carácter de tipo para Byte, ya que el carácter natural sería B, que es un carácter válido en un literal hexadecimal.

IntegerLiteral ::= IntegralLiteralValue [ IntegralTypeCharacter ]

IntegralLiteralValue ::= IntLiteral | HexLiteral | OctalLiteral

IntegralTypeCharacter ::=  
 ShortCharacter |  
 UnsignedShortCharacter |  
 IntegerCharacter |  
 UnsignedIntegerCharacter |  
 LongCharacter |  
 UnsignedLongCharacter |  
 IntegerTypeCharacter |  
 LongTypeCharacter

ShortCharacter ::= S

UnsignedShortCharacter ::= US

IntegerCharacter ::= I

UnsignedIntegerCharacter ::= UI

LongCharacter ::= L

UnsignedLongCharacter ::= UL

IntLiteral ::= Digit+

HexLiteral ::= & H HexDigit+

OctalLiteral ::= & O OctalDigit+

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

HexDigit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F

OctalDigit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

### Literales de punto flotante

Un literal de punto flotante es un literal entero seguido de un elemento separador decimal opcional (el carácter de punto ASCII) y la mantisa, así como de un exponente de base 10 opcional. El literal de punto flotante es de tipo Double de forma predeterminada. Si se especifica el carácter de tipo Single, Double o Decimal, el literal pertenecerá a este tipo. Si el tipo de un literal de punto flotante no es suficiente para incluir el literal, se produce un error en tiempo de compilación.

Anotación:

Conviene indicar que el tipo de datos Decimal puede codificar ceros finales en un valor. Actualmente la especificación no señala si el compilador debe admitir los ceros finales de un literal Decimal.

FloatingPointLiteral ::=  
 FloatingPointLiteralValue [ FloatingPointTypeCharacter ] |  
 IntLiteral FloatingPointTypeCharacter

FloatingPointTypeCharacter ::=  
 SingleCharacter |  
 DoubleCharacter |  
 DecimalCharacter |  
 SingleTypeCharacter |  
 DoubleTypeCharacter |  
 DecimalTypeCharacter

SingleCharacter ::= F

DoubleCharacter ::= R

DecimalCharacter ::= D

FloatingPointLiteralValue ::=  
 IntLiteral . IntLiteral [ Exponent ] |  
 . IntLiteral [ Exponent ] |  
 IntLiteral Exponent

Exponent ::= E [ Sign ] IntLiteral

Sign ::= + | -

### Literales de cadena

Un literal de cadena es una secuencia de cero o más caracteres Unicode que comienza y finaliza con un carácter de comillas dobles de ASCII o un carácter de comillas dobles izquierdas o derechas de Unicode. Dentro de una cadena, una secuencia de dos caracteres de comillas dobles es una secuencia de escape que representa una comilla doble en la cadena. La constante de cadena es de tipo String.

Module Test  
 Sub Main()  
  
 ' This prints out: ".  
 Console.WriteLine("""")  
  
 ' This prints out: a"b.  
 Console.WriteLine("a""b")  
  
 ' This causes a compile error due to mismatched double-quotes.  
 Console.WriteLine("a"b")  
 End Sub  
End Module

Se permite que el compilador reemplace una expresión de cadena constante por un literal de cadena. Cada literal de cadena no genera necesariamente una nueva instancia de cadena. Cuando en el mismo programa aparecen dos o más literales de cadena equivalentes según el operador de igualdad de cadenas utilizando la semántica de comparación binaria, estos literales pueden hacer referencia a la misma instancia de cadena. Por ejemplo, el siguiente programa puede devolver True porque los dos literales pueden hacer referencia a la misma instancia de cadena.

Module Test  
 Sub Main()  
 Dim a As Object = "he" & "llo"  
 Dim b As Object = "hello"  
 Console.WriteLine(a Is b)  
 End Sub  
End Module

StringLiteral ::=  
 DoubleQuoteCharacter [ StringCharacter+ ] DoubleQuoteCharacter

DoubleQuoteCharacter ::=  
 " |  
 < Unicode left double-quote character (0x201C) > |  
 < Unicode right double-quote character (0x201D) >

StringCharacter ::=  
 < Character except for DoubleQuoteCharacter > |  
 DoubleQuoteCharacter DoubleQuoteCharacter

### Literales de carácter

Los literales de carácter representan un único carácter Unicode del tipo Char. Dos caracteres de comillas dobles constituyen una secuencia de escape que representa el carácter de comillas dobles.

Module Test  
 Sub Main()  
  
 ' This prints out: a.  
 Console.WriteLine("a"c)  
  
 ' This prints out: ".  
 Console.WriteLine(""""c)  
 End Sub  
End Module

CharacterLiteral ::= DoubleQuoteCharacter StringCharacter DoubleQuoteCharacter C

### Literales de fecha

Los literales de fecha representan un momento temporal determinado, expresado como un valor del tipo Date. El literal puede especificar una fecha y una hora, solamente una fecha o solamente una hora. Si se omite el valor de fecha, se supone la fecha del 1 de enero del año 1 en el calendario gregoriano. Si se omite el valor de hora, se supone 12:00:00 a.m.

Para evitar problemas a la hora de interpretar el año en un valor de fecha, el año no puede usar dos dígitos. Cuando se expresa una fecha del primer siglo A.D./C.E., se deben especificar ceros iniciales.

Los valores de hora se pueden especificar con los formatos de 24 o 12 horas; si se omite AM o PM en los valores de hora, se supone que se usa el formato de 24 horas. Si un valor de hora omite los minutos, se usa de forma predeterminada el literal 0. Si un valor de hora omite los segundos, se usa de forma predeterminada el literal 0. Si se omiten los minutos y los segundos, se debe especificar AM o PM. Si el valor de fecha especificado se sitúa fuera del intervalo del tipo Date, se produce un error en tiempo de compilación.

En el siguiente ejemplo se muestran varios literales de fecha.

Dim d As Date  
  
d = # 8/23/1970 3:45:39AM #  
d = # 8/23/1970 # ' Date value: 8/23/1970 12:00:00AM.  
d = # 3:45:39AM # ' Date value: 1/1/1 3:45:39AM.  
d = # 3:45:39 # ' Date value: 1/1/1 3:45:39AM.  
d = # 13:45:39 # ' Date value: 1/1/1 1:45:39PM.  
d = # 1AM # ' Date value: 1/1/1 1:00:00AM.  
d = # 13:45:39PM # ' This date value is not valid.

DateLiteral ::= # [ Whitespace+ ] DateOrTime [ Whitespace+ ] #

DateOrTime ::=  
 DateValue Whitespace+ TimeValue |  
 DateValue |  
 TimeValue

DateValue ::=  
 MonthValue / DayValue / YearValue |  
 MonthValue – DayValue - YearValue

TimeValue ::=  
 HourValue : MinuteValue [ : SecondValue ] [ WhiteSpace+ ] [ AMPM ] |  
 HourValue [ WhiteSpace+ ] AMPM

MonthValue ::= IntLiteral

DayValue ::= IntLiteral

YearValue ::= IntLiteral

HourValue ::= IntLiteral

MinuteValue ::= IntLiteral

SecondValue ::= IntLiteral

AMPM ::= AM | PM

ElseIf ::= ElseIf | Else If

### Nothing

Nothing es un literal especial; no cuenta con ningún tipo y se puede convertir a todos los tipos del sistema de tipos, incluidos los parámetros de tipo. Cuando se convierte a un tipo determinado, equivale al valor predeterminado de dicho tipo.

Nothing ::= Nothing

## Separadores

Los siguientes caracteres ASCII son separadores:

Separator ::= ( | ) | { | } | ! | # | , | . | : | ?

## Caracteres de operador

Los siguientes caracteres o secuencias de caracteres ASCII indican operadores:

Operator ::=  
 & | \* | + | - | / | \ | ^ | < | = | >

# Directivas de preprocesamiento

Una vez realizado el análisis léxico de un archivo, se producen varios tipos de preprocesamiento del código fuente. La más importante, la compilación condicional, determina qué código fuente se procesa mediante la gramática sintáctica; los otros dos tipos de directivas (directivas de código fuente externo y directivas de región) proporcionan metainformación acerca del código fuente pero no tienen ningún efecto en la compilación.

## Compilación condicional

La compilación condicional controla si las secuencias de líneas lógicas se convierten en código real. En el comienzo de la compilación condicional, todas las líneas lógicas están habilitadas; sin embargo, al incluirlas en instrucciones de compilación condicional, estas líneas se pueden deshabilitar de forma selectiva dentro del archivo, lo que daría lugar a su omisión durante el resto del proceso de compilación.

Por ejemplo, el programa:

#Const A = True  
#Const B = False  
  
Class C  
  
#If A Then  
 Sub F()  
 End Sub  
#Else  
 Sub G()  
 End Sub  
#End If  
  
#If B Then  
 Sub H()  
 End Sub  
#Else  
 Sub I()  
 End Sub  
#End If  
  
End Class

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

Class C  
 Sub F()  
 End Sub  
  
 Sub I()  
 End Sub  
End Class

Las expresiones constantes que se permiten en las directivas de compilación condicional son un subconjunto de las expresiones constantes generales.

El preprocesador admite espacios en blanco y continuaciones de línea explícitas antes y después de cada token.

Start ::= [ CCStatement+ ]

CCStatement ::=  
 CCConstantDeclaration |  
 CCIfGroup |  
 LogicalLine

CCExpression ::=  
 LiteralExpression |  
 CCParenthesizedExpression |  
 CCSimpleNameExpression |  
 CCCastExpression |  
 CCOperatorExpression |  
 CCConditionalExpression

CCParenthesizedExpression ::= ( CCExpression )

CCSimpleNameExpression ::= Identifier

CCCastExpression ::=   
 DirectCast ( CCExpression , TypeName ) |  
 TryCast ( CCExpression , TypeName ) |  
 CType ( CCExpression , TypeName ) |  
 CastTarget ( CCExpression )

CCOperatorExpression ::=  
 CCUnaryOperator CCExpression  
 CCExpression CCBinaryOperator CCExpression

CCUnaryOperator ::= + | - | Not

CCBinaryOperator ::= + | - | \* | / | \ | Mod | ^ | = | < > | < | > |  
 < = | > = | & | And | Or | Xor | AndAlso | OrElse | < < | > >

CCConditionalExpression ::=   
 If ( CCExpression , CCExpression , CCExpression ) |  
 If ( CCExpression , CCExpression )

### Directivas de constantes condicionales

Las instrucciones de constantes condicionales definen constantes que existen en un espacio de declaración de compilación condicional independiente cuyo ámbito es el archivo de código fuente. El espacio de declaración es especial en el sentido de que no es necesaria ninguna declaración explícita de las constantes de compilación condicional (las constantes condicionales se pueden definir de forma implícita en una directiva de compilación condicional).

Antes de recibir la asignación de un valor, la constante de compilación condicional tiene el valor Nothing. Cuando a una constante de compilación condicional se le asigna un valor, que debe ser una expresión constante, el tipo de la constante se convierte en el tipo del valor que se le asigna. Las constantes de compilación condicional se pueden redefinir varias veces en un archivo de código fuente.

Por ejemplo, el siguiente código solo imprime la cadena about to print value y el valor Test.

Module M1  
 Sub PrintValue(Test As Integer)  
  
#Const DebugCode = True  
  
#If DebugCode Then  
 Console.WriteLine("about to print value")  
#End If  
  
#Const DebugCode = False  
  
 Console.WriteLine(Test)  
  
#If DebugCode Then  
 Console.WriteLine("printed value")  
#End If  
  
 End Sub  
End Module

El entorno de compilación también puede definir constantes condicionales en un espacio de declaración de compilación condicional.

CCConstantDeclaration ::= # Const Identifier = CCExpression LineTerminator

### Directivas de compilación condicional

Las directivas de compilación condicional controlan la compilación condicional y solo pueden hacer referencia a las expresiones constantes y a las constantes de compilación condicional. Cada una de las expresiones constantes de un único grupo de compilación condicional se evalúa y se convierte al tipo Boolean por orden textual de primera a última hasta que una de las expresiones condicionales se evalúa como True. Si una expresión no se puede convertir a Boolean, se produce un error en tiempo de compilación. A la hora de evaluar las expresiones constantes de compilación condicional, se usa una semántica permisiva y comparaciones binarias de cadenas, independientemente de las directivas de Option o de la configuración del entorno de compilación.

Se deshabilitan todas las líneas delimitadas por el grupo, incluidas las directivas de compilación condicional anidadas, excepto las líneas situadas entre la instrucción que contiene la expresión True y la siguiente instrucción condicional del grupo, o las líneas situadas entre la instrucción Else y la instrucción End If si aparece Else en el grupo y todas las demás expresiones se evalúan como False.

En este ejemplo, no se procesa la llamada a WriteToLog en la directiva de compilación condicional Trace porque la directiva de compilación condicional Debug circundante se evalúa como False.

#Const Debug = False ' Debugging off  
#Const Trace = True ' Tracing on  
  
Class PurchaseTransaction  
 Sub Commit()  
  
#If Debug Then  
 CheckConsistency()  
#If Trace Then  
 WriteToLog(Me.ToString())  
#End If  
#End If  
 ...  
 End Sub  
End Class

CCIfGroup ::=  
 # If CCExpression [ Then ] LineTerminator  
 [ CCStatement+ ]  
 [ CCElseIfGroup+ ]  
 [ CCElseGroup ]  
 # End If LineTerminator

CCElseIfGroup ::=  
 # ElseIf CCExpression [ Then ] LineTerminator  
 [ CCStatement+ ]

CCElseGroup ::=  
 # Else LineTerminator  
 [ CCStatement+ ]

## Directivas de código fuente externo

Un archivo de código fuente puede incluir directivas de código fuente externo, que indican una asignación entre líneas de código fuente y texto externo al código fuente. Las directivas de código fuente externo no tienen efecto en la compilación y no se pueden anidar. Por ejemplo:

Module Test  
 Sub Main()  
  
#ExternalSource("c:\wwwroot\inetpub\test.aspx", 30)  
 Console.WriteLine("In test.aspx")  
#End ExternalSource  
  
 End Sub  
End Module

Start ::= [ ExternalSourceStatement+ ]

ExternalSourceStatement ::= ExternalSourceGroup | LogicalLine

ExternalSourceGroup ::=  
 # ExternalSource ( StringLiteral , IntLiteral ) LineTerminator  
 [ LogicalLine+ ]  
 # End ExternalSource LineTerminator

## Directivas de región

Las directivas de región agrupan líneas de código fuente, pero no tienen ningún otro efecto en la compilación. El grupo completo se puede contraer y ocultar, o expandir y ver, en el entorno de desarrollo integrado (IDE). Las regiones se pueden anidar. Las directivas de región son especiales porque no pueden comenzar ni terminar dentro del cuerpo de un método y deben respetar la estructura de bloques del programa. Por ejemplo:

Module Test#Region "Startup code – do not edit"  
 Sub Main()  
 End Sub  
#End Region  
  
End Module

' Error due to Region directives breaking the block structure  
Class C  
#Region "Fred"  
End Class  
#End Region

Start ::= [ RegionStatement+ ]

RegionStatement ::= RegionGroup | LogicalLine

RegionGroup ::=  
 # Region StringLiteral LineTerminator  
 [ RegionStatement+ ]  
 # End Region LineTerminator

## Directivas de suma de comprobación externa

Los archivos de código fuente pueden incluir una directiva de suma de comprobación externa que indica la suma de comprobación que se debe emitir para un archivo al que se hace referencia en una directiva de código fuente externo. En los demás aspectos, las directivas de código fuente externo no tienen ningún efecto en la compilación.

Las directivas de suma de comprobación externa contienen el nombre del archivo externo, un identificador único global (GUID) asociado con el archivo y la suma de comprobación del archivo. El GUID se especifica como una constante de cadena con el formato "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}", donde x es un dígito hexadecimal. La suma de comprobación se especifica como una constante de cadena con el formato "xxxx…", donde x es un dígito hexadecimal. El número de dígitos de una suma de comprobación debe ser un número par.

Un archivo externo puede tener varias directivas de suma de comprobación externa asociadas, siempre que todos los valores de GUID y suma de comprobación coincidan de forma exacta. Si el nombre del archivo externo coincide con el nombre de un archivo que se compila, la suma de comprobación se omite a favor del cálculo de la suma de comprobación del compilador.

Por ejemplo:

#ExternalChecksum("c:\wwwroot\inetpub\test.aspx", \_  
 "{12345678-1234-1234-1234-123456789abc}", \_  
 "1a2b3c4e5f617239a49b9a9c0391849d34950f923fab9484")  
  
Module Test  
 Sub Main()  
  
#ExternalSource("c:\wwwroot\inetpub\test.aspx", 30)  
 Console.WriteLine("In test.aspx")  
#End ExternalSource  
  
 End Sub  
End Module

Start ::= [ ExternalChecksumStatement+ ]

ExternalChecksumStatement ::=  
 # ExternalChecksum ( StringLiteral , StringLiteral , StringLiteral ) LineTerminator

# Conceptos generales

En este capítulo se tratan diferentes conceptos necesarios para comprender la semántica del lenguaje Microsoft Visual Basic. Muchos de los conceptos resultarán familiares para los programadores de Visual Basic o de C/C++, pero es posible que sus definiciones concretas varíen.

## Declaraciones

Los programas de Visual Basic constan de entidades con nombre. Estas entidades se incluyen por medio de declaraciones y representan el "significado" del programa.

En un nivel superior, los espacios de nombres son entidades que organizan otras entidades, como los espacios de nombres anidados y los tipos. Los tipos son entidades que describen valores y definen código ejecutable. Los tipos pueden contener tipos anidados y miembros de tipo. Los miembros de tipo son constantes, variables, métodos, operadores, propiedades, eventos, valores de enumeración y constructores.

Las entidades que pueden contener otras entidades definen un espacio de declaración. Las entidades se incluyen en un espacio de declaración por medio de declaraciones o de herencia; el espacio de declaración que las contiene se denomina contexto de declaración de las entidades. La declaración de una entidad en un espacio de declaración define a su vez un nuevo espacio de declaración que puede contener otras definiciones de entidad anidadas; así, las declaraciones de un programa conforman una jerarquía de espacios de declaración.

Excepto en el caso de miembros de tipo sobrecargados, las declaraciones no pueden incluir entidades con el mismo nombre del mismo tipo en el mismo contexto de declaración. Además, un espacio de declaración nunca puede contener diferentes tipos de entidades con el mismo nombre; por ejemplo, un espacio de declaración nunca puede contener una variable y un método con el mismo nombre.

Anotación

Puede que en otros lenguajes sea posible crear un espacio de declaración que contenga diferentes tipos de entidades con el mismo nombre (por ejemplo, si el lenguaje distingue entre mayúsculas y minúsculas y permite diferentes declaraciones en función del uso de mayúsculas y minúsculas). En este caso, se considera enlazada a este nombre la entidad más accesible; si las entidades más accesibles son de tipos diferentes, el nombre se considera ambiguo. Public es más accesible que Protected Friend, Protected Friend es más accesible que Protected o Friend, y Protected o Friend son más accesibles que Private.

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. En el siguiente ejemplo, las dos declaraciones de espacios de nombres contribuyen al mismo espacio de declaración, en este caso, declarando dos clases con los nombres completos Data.Customer y Data.Order:

Namespace Data  
 Class Customer  
 End Class  
End Namespace  
  
Namespace Data  
 Class Order  
 End Class  
End Namespace

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.

### Sobrecargas y firmas

La declaración de entidades con el mismo nombre y del mismo tipo en un espacio de declaración solamente se puede realizar mediante la sobrecarga. Únicamente se pueden sobrecargar métodos, operadores, constructores de instancia y propiedades.

Los miembros de tipo sobrecargados deben poseer firmas únicas. La signatura de un miembro de tipo se compone del número de parámetros de tipo y el número y los tipos de los parámetros del miembro. Los operadores de conversión también incluyen el tipo devuelto del operador en la firma.

Los siguientes elementos no forman parte de la firma de un miembro y por tanto no se pueden sobrecargar:

Modificadores de un miembro de tipo (por ejemplo, Shared o Private).

Modificadores de un parámetro (por ejemplo, ByVal o ByRef).

Nombres de los parámetros.

El tipo devuelto de un método u operador (excepto los operadores de conversión) o el tipo de elemento de una propiedad.

Restricciones en un parámetro de tipo.

El siguiente ejemplo muestra un conjunto de declaraciones de métodos sobrecargados con sus firmas. Esta declaración no sería válida, ya que varias declaraciones de método tienen la misma firma.

Interface ITest  
 Sub F1() ' Signature is ().  
 Sub F2(x As Integer) ' Signature is (Integer).  
 Sub F3(ByRef x As Integer) ' Signature is (Integer).  
 Sub F4(x As Integer, y As Integer) ' Signature is   
 (Integer, Integer).  
 Function F5(s As String) As Integer ' Signature is (String).  
 Function F6(x As Integer) As Integer ' Signature is (Integer).  
 Sub F7(a() As String) ' Signature is (String()).  
 Sub F8(ParamArray a() As String) ' Signature is (String()).  
 Sub F9(Of T)() ' Signature is !1().  
 Sub F10(Of T, U)(x As T, y As U) ' Signature is !2(!1, !2)  
 Sub F11(Of U, T)(x As T, y As U) ' Signature is !2(!2, !1)  
 Sub F12(Of T)(x As T) ' Signature is !1(!1)  
 Sub F13(Of T As IDisposable)(x As T) ' Signature is !1(!1)  
End Interface

Se puede definir un tipo genérico que puede contener miembros con firmas idénticas en función de los argumentos de tipo incluidos. Para intentar eliminar la ambigüedad entre estas sobrecargas se usan reglas de resolución de sobrecarga, aunque puede haber casos en los que resulte imposible evitar la ambigüedad. Por ejemplo:

Class C(Of T)  
 Sub F(x As Integer)  
 End Sub  
  
 Sub F(x As T)  
 End Sub  
  
 Sub G(Of U)(x As T, y As U)  
 End Sub  
  
 Sub G(Of U)(x As U, y As T)  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As New C(Of Integer)  
 x.F(10) ' Calls C(Of T).F(Integer)  
 x.G(Of Integer)(10, 10) ' Error: Can't choose between overloads  
 End Sub  
End Module

## Ámbito

El ámbito de un nombre de entidad es el conjunto de todos los espacios de declaración donde es posible referirse a dicho nombre sin una calificación. En general, el ámbito de un nombre de entidad es todo su contexto de declaración; sin embargo, la declaración de una entidad puede contener declaraciones anidadas de entidades con el mismo nombre. En este caso, la entidad anidada sombrea, u oculta, la entidad externa, y el acceso a esta entidad sombreada solamente es posible por medio de la calificación.

El sombreado por anidado se produce en espacios de nombres o tipos anidados dentro de espacios de nombres, en tipos anidados dentro de otros tipos y en los cuerpos de los miembros de tipo. El sombreado por el anidado de las declaraciones siempre se produce de forma implícita; no es necesaria ninguna sintaxis explícita.

En el siguiente ejemplo, dentro del método F, la variable local i sombrea la variable de instancia i, pero dentro del método G, i se sigue refiriendo a la variable de instancia.

Class Test  
 Private i As Integer = 0  
  
 Sub F()  
 Dim i As Integer = 1  
 End Sub  
  
 Sub G()  
 i = 1  
 End Sub  
End Class

Cuando un nombre en un ámbito interno oculta otro nombre del ámbito externo, sombrea todas sus apariciones sobrecargadas. En el siguiente ejemplo, 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") es errónea.

Class Outer  
 Shared Sub F(i As Integer)  
 End Sub  
  
 Shared Sub F(s As String)  
 End Sub  
  
 Class Inner  
 Shared Sub F(l As Long)  
 End Sub  
  
 Sub G()  
 F(1) ' Invokes Outer.Inner.F.  
 F("Hello") ' Error.  
 End Sub  
 End Class  
End Class

## Herencia

La relación de herencia es aquella en la que un tipo (el tipo derivado) deriva de otro (el tipo base), de forma que el espacio de declaración del tipo derivado contiene de forma implícita los miembros de tipo no constructores y los tipos anidados accesibles de su tipo base. En el ejemplo siguiente, la clase A es la clase base de B y B deriva de A.

Class A  
End Class  
  
Class B  
 Inherits A  
End Class

Dado que A no especifica explícitamente una clase base, su clase base es implícitamente Object.

Estos son algunos aspectos importantes de la herencia:

La herencia es transitiva. Si el tipo C se deriva del tipo B y el tipo B deriva del tipo A, entonces el tipo C hereda los miembros de tipo declarados en los tipos B y A.

Un tipo derivado extiende, pero no puede limitar, su tipo base. Un tipo derivado puede agregar nuevos miembros de tipo y sombrear miembros de tipo heredados, pero no puede quitar la definición de un miembro de tipo heredado.

Puesto que una instancia de un tipo contiene todos los miembros de tipo de su tipo base, siempre existe una conversión del tipo derivado a su tipo base.

Todos los tipos deben tener un tipo base, excepto el tipo Object. Por tanto, Object es el tipo base definitivo de todos los tipos y todos los tipos se pueden convertir en él.

No se permite la circularidad en la derivación. Es decir, si un tipo B deriva de un tipo A, es incorrecto que el tipo A derive directa o indirectamente del tipo B.

Un tipo no puede derivar directa ni indirectamente de un tipo anidado en él.

En el siguiente ejemplo se produce un error en tiempo de compilación ya que las clases dependen circularmente entre sí.

Class A  
 Inherits B  
End Class  
  
Class B  
 Inherits C  
End Class  
  
Class C  
 Inherits A  
End Class

En el ejemplo siguiente también se produce un error en tiempo de compilación ya que B deriva indirectamente de su clase anidada C a través de la clase A.

Class A  
 Inherits B.C  
End Class  
  
Class B  
 Inherits A  
  
 Public Class C  
 End Class   
End Class

En el siguiente ejemplo no se produce ningún error ya que la clase A no deriva de la clase B.

Class A  
 Class B  
 Inherits A  
 End Class   
End Class

### Clases MustInherit y NotInheritable

La clase MustInherit es un tipo incompleto que solamente puede actuar como tipo base. No se pueden crear instancias de la clase MustInherit, por lo que no es correcto usar el operador New en ella. Se pueden declarar variables de las clases MustInherit; a estas variables solamente se les puede asignar Nothing o un valor de una clase derivada de la clase MustInherit.

Cuando una clase normal deriva de la clase MustInherit, la clase normal debe invalidar todos los miembros MustOverride heredados. Por ejemplo:

MustInherit Class A  
 Public MustOverride Sub F()  
End Class  
  
MustInherit Class B  
 Inherits A  
  
 Public Sub G()  
 End Sub  
End Class   
  
Class C  
 Inherits B  
  
 Public Overrides Sub F()  
 End Sub   
End Class

La clase MustInherit A presenta un método MustOverride F. La clase B presenta un método adicional G, pero no proporciona ninguna implementación de F. Por tanto, la clase B también se debe declarar como MustInherit. La clase C invalida F y proporciona una implementación real. Ya que no existen miembros MustOverride pendientes en la clase C, no es necesario que se marque como MustInherit.

Una clase NotInheritable es una clase de la que no pueden derivar otras clases. Las clases NotInheritable se usan principalmente para evitar la derivación no deseada.

En este ejemplo, la clase B es errónea porque intenta derivarse de la clase NotInheritable A. Una clase no se puede marcar como MustInherit y como NotInheritable.

NotInheritable Class A  
End Class  
  
Class B  
 ' Error, a class cannot derive from a NotInheritable class.  
 Inherits A  
End Class

### Interfaces y herencia múltiple

A diferencia de otros tipos, que solamente derivan de un único tipo base, una interfaz puede derivar de varias interfaces base. Así, una interfaz puede heredar un miembro de tipo con el mismo nombre de interfaces base diferentes. En este caso, el nombre heredado varias veces no está disponible en la interfaz derivada y la referencia a uno de estos miembros de tipo a través de la interfaz derivada produce un error en tiempo de compilación, independientemente de las firmas o de la sobrecarga. La referencia a los miembros de tipo en conflicto debe hacerse a través de un nombre de interfaz base.

En el siguiente ejemplo, las dos primeras instrucciones producen errores en tiempo de compilación ya que el miembro de herencia múltiple Count no está disponible en la interfaz IListCounter:

Interface IList  
 Property Count() As Integer  
End Interface  
  
Interface ICounter  
 Sub Count(i As Integer)  
End Interface  
  
Interface IListCounter  
 Inherits IList  
 Inherits ICounter   
End Interface   
  
Module Test  
 Sub F(x As IListCounter)  
 x.Count(1) ' Error, Count is not available.  
 x.Count = 1 ' Error, Count is not available.  
 CType(x, IList).Count = 1 ' Ok, invokes IList.Count.  
 CType(x, ICounter).Count(1) ' Ok, invokes ICounter.Count.  
 End Sub   
End Module

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.

Si un miembro de tipo se hereda de la misma interfaz base a través de varias rutas de acceso, se trata como si solamente se hubiese heredado una vez. Es decir, la interfaz derivada solo contiene una instancia de cada miembro de tipo heredado de una interfaz base determinada. Por ejemplo:

Interface IBase  
 Sub F(i As Integer)  
End Interface  
  
Interface ILeft  
 Inherits IBase  
End Interface  
  
Interface IRight  
 Inherits IBase  
End Interface  
  
Interface IDerived  
 Inherits ILeft, IRight  
End Interface  
  
Class Derived  
 Implements IDerived  
  
 ' Only have to implement F once.  
 Sub F(i As Integer) Implements IDerived.F  
 End Sub  
End Class

Si un nombre de miembro de tipo se sombrea en una ruta de acceso de la jerarquía de herencia, se sombrea en todas las rutas de acceso. En el siguiente ejemplo, el miembro IBase.F se ve sombreado por el miembro ILeft.F, pero no por el miembro IRight:

Interface IBase  
 Sub F(i As Integer)  
End Interface   
  
Interface ILeft  
 Inherits IBase  
  
 Shadows Sub F(i As Integer)  
End Interface   
  
Interface IRight  
 Inherits IBase  
  
 Sub G()  
End Interface   
  
Interface IDerived  
 Inherits ILeft, IRight   
End Interface   
  
Class Test  
 Sub H(d As IDerived)  
 d.F(1) ' Invokes ILeft.F.  
 CType(d, IBase).F(1) ' Invokes IBase.F.  
 CType(d, ILeft).F(1) ' Invokes ILeft.F.  
 CType(d, IRight).F(1) ' Invokes IBase.F.  
 End Sub   
End Class

La invocación d.F(1) selecciona ILeft.F, aunque IBase.F no parece quedar sombreado en la ruta de acceso que conduce a IRight. Debido a que la ruta de acceso desde IDerived que pasa por ILeft hasta IBase sombrea a IBase.F, el miembro también se sombrea en la ruta de acceso desde IDerived que pasa por IRight hasta IBase.

### Sombreado

Para sombrear el nombre de un miembro de tipo heredado, los tipos derivados lo vuelven a declarar. El sombreado de un nombre no quita los miembros de tipo heredados con dicho nombre, simplemente hace que no estén disponibles en la clase derivada. La declaración de sombreado puede ser cualquier tipo de entidad.

Las entidades que se pueden sobrecargar pueden elegir entre dos formas de sombreado. El sombreado por nombre se especifica con la palabra clave Shadows. La entidad que sombrea por nombre oculta todo lo que lleve este nombre en la clase base, incluidas todas las sobrecargas. El sombreado por nombre y signatura se especifica con la palabra clave Overloads. La entidad que sombrea por nombre y firma oculta todo lo que lleva este nombre y que tiene la misma firma que la entidad. Por ejemplo:

Class Base  
 Sub F()  
 End Sub  
  
 Sub F(i As Integer)  
 End Sub  
  
 Sub G()  
 End Sub  
  
 Sub G(i As Integer)  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 ' Only hides F(Integer).  
 Overloads Sub F(i As Integer)  
 End Sub  
  
 ' Hides G() and G(Integer).  
 Shadows Sub G(i As Integer)  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As New Derived()  
  
 x.F() ' Calls Base.F().  
 x.G() ' Error: Missing parameter.  
 End Sub  
End Module

El sombreado de un método con un argumento ParamArray por nombre y signatura solamente oculta la signatura individual, no todas las posibles signaturas expandidas. Esto se cumple aunque la firma del método de sombreado coincida con la firma no expandida del método sombreado. El siguiente ejemplo:

Class Base  
 Sub F(ParamArray x() As Integer)  
 Console.WriteLine("Base")  
 End Sub  
End Class  
  
Class Derived   
 Inherits Base  
  
 Overloads Sub F(x() As Integer)  
 Console.WriteLine("Derived")  
 End Sub  
End Class  
  
Module Test  
 Sub Main  
 Dim d As New Derived()  
 d.F(10)  
 End Sub  
End Module

imprime Base, aunque Derived.F tiene la misma signatura que la forma no expandida de Base.F.

Por el contrario, un método con un argumento ParamArray solamente sombrea los métodos con la misma signatura, no todas las posibles signaturas expandidas. El siguiente ejemplo:

Class Base  
 Sub F(x As Integer)  
 Console.WriteLine("Base")  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Overloads Sub F(ParamArray x() As Integer)  
 Console.WriteLine("Derived")  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim d As New Derived()  
 d.F(10)  
 End Sub  
End Module

imprime Base, aunque Derived.F tiene una forma expandida con la misma signatura que Base.F.

Los métodos o propiedades de sombreado que no especifican Shadows ni Overloads suponen Overloads si el método o la propiedad se declaran como Overrides; en caso contrario, suponen Shadows. Si un miembro de un conjunto de entidades sobrecargadas especifica la palabra clave Shadows u Overloads, todos deben especificarla. Las palabras clave Shadows y Overloads no se pueden especificar a la vez. Shadows y Overloads no se pueden especificar en un módulo estándar; los miembros de un módulo estándar sombrean de forma implícita los miembros heredados de Object.

Se puede sombrear el nombre de un miembro de tipo que se ha heredado varias veces a través de la herencia de interfaz (y que, por tanto, no está disponible); de esta forma, el nombre estará disponible en la interfaz derivada.

Por ejemplo:

Interface ILeft  
 Sub F()  
End Interface  
  
Interface IRight  
 Sub F()  
End Interface  
  
Interface ILeftRight  
 Inherits ILeft, IRight  
  
 Shadows Sub F()  
End Interface  
  
Module Test  
 Sub G(i As ILeftRight)  
 i.F() ' Calls ILeftRight.F.  
 CType(i, ILeft).F() ' Calls ILeft.F.  
 CType(i, IRight).F() ' Calls IRight.F.  
 End Sub  
End Module

Debido a que se permite que los métodos sombreen métodos heredados, es posible que una clase contenga varios métodos Overridable con la misma signatura. Esto no presenta ningún problema de ambigüedad puesto que solo está visible el método más derivado. En el siguiente ejemplo, las clases C y D contienen dos métodos Overridable con la misma signatura:

Class A  
 Public Overridable Sub F()  
 Console.WriteLine("A.F")  
 End Sub   
End Class   
  
Class B  
 Inherits A  
  
 Public Overrides Sub F()  
 Console.WriteLine("B.F")  
 End Sub   
End Class   
  
Class C  
 Inherits B  
  
 Public Shadows Overridable Sub F()  
 Console.WriteLine("C.F")  
 End Sub   
End Class   
  
Class D  
 Inherits C  
  
 Public Overrides Sub F()  
 Console.WriteLine("D.F")  
 End Sub   
End Class   
  
Module Test  
 Sub Main()  
 Dim d As New D()  
 Dim a As A = d  
 Dim b As B = d  
 Dim c As C = d  
 a.F()  
 b.F()  
 c.F()  
 d.F()  
 End Sub   
End Module

Aquí existen dos métodos Overridable: uno que presenta la clase A y otro que presenta la clase C. El método que presenta la clase C oculta el método que se hereda de la clase A. De este modo, la declaración Overrides de la clase D invalida el método que presenta la clase C y la clase D no puede invalidar el método que presenta la clase A. El ejemplo produce el resultado:

B.F  
B.F  
D.F  
D.F

Es posible invocar el método Overridable oculto mediante el acceso a una instancia de la clase D a través de un tipo menos derivado en el que el método no esté oculto.

No se puede sombrear un método MustOverride, ya que en la mayoría de los casos la clase sería inservible. Por ejemplo:

MustInherit Class Base  
 Public MustOverride Sub F()  
End Class  
  
MustInherit Class Derived  
 Inherits Base  
  
 Public Shadows Sub F()  
 End Sub  
End Class  
  
Class MoreDerived  
 Inherits Derived  
  
 ' Error: MustOverride method Base.F is not overridden.  
End Class

En este caso, la clase MoreDerived debe invalidar el método MustOverride Base.F, pero dado que la clase Derived sombrea Base.F, esto no es posible. No existen ningún modo de declarar un descendiente válido de Derived.

A diferencia del sombreado de un nombre de un ámbito externo, cuando se sombrea un nombre accesible de un ámbito heredado se genera una advertencia, como en el siguiente ejemplo.

Class Base  
 Public Sub F()  
 End Sub  
  
 Private Sub G()  
 End Sub   
End Class  
  
Class Derived  
 Inherits Base  
  
 Public Sub F() ' Warning: shadowing an inherited name.  
 End Sub  
  
 Public Sub G() ' No warning, Base.G is not accessible here.  
 End Sub  
End Class

La declaración del método F en la clase Derived genera una advertencia. El sombreado de un nombre heredado es específicamente válido y no genera ningún 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 la clase Base presentó 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 supuesto que las clases derivadas dejasen de ser válidas.

La advertencia causada por el sombreado de un nombre heredado puede eliminarse mediante el uso de los modificadores Shadows u Overloads:

Class Base  
 Public Sub F()  
 End Sub   
End Class   
  
Class Derived  
 Inherits Base  
  
 Public Shadows Sub F() 'OK.  
 End Sub  
End Class

El modificador Shadows indica la intención de sombrear el miembro heredado. Si no existe ningún nombre de miembro de tipo que se pueda sombrear, la especificación de los modificadores Shadows u Overloads no supone ningún error.

Una declaración de un miembro nuevo solamente sombrea un miembro heredado dentro del ámbito del nuevo miembro, como en el siguiente ejemplo:

Class Base  
 Public Shared Sub F()  
 End Sub   
End Class   
  
Class Derived  
 Inherits Base  
  
 Private Shared Shadows Sub F() ' Shadows Base.F in class Derived only.  
 End Sub   
End Class   
  
Class MoreDerived  
 Inherits Derived  
  
 Shared Sub G()  
 F() ' Invokes Base.F.  
 End Sub   
End Class

En el ejemplo anterior, la declaración del método F en la clase Derived sombrea el método F que se heredó de la clase Base; sin embargo, como el nuevo método F de la clase Derived tiene acceso de tipo Private, su ámbito no se extiende a la clase MoreDerived. De este modo, la llamada F() de MoreDerived.G es válida e invocará a Base.F. En el caso de los miembros de tipo sobrecargados, su conjunto completo se trata como si todos tuviesen el acceso más permisivo con fines de sombreado.

Class Base  
 Public Sub F()  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Private Shadows Sub F()  
 End Sub  
  
 Public Shadows Sub F(i As Integer)  
 End Sub  
End Class  
  
Class MoreDerived  
 Inherits Derived  
  
 Public Sub G()  
 F() ' Error. No accessible member with this signature.  
 End Sub  
End Class

En este ejemplo, aunque la declaración de F() en Derived se declara con acceso de tipo Private, el método F(Integer) sobrecargado se declara con acceso de tipo Public. Por tanto, para los fines de sombreado, el nombre F de Derived se trata como si fuese de tipo Public, por lo que ambos métodos sombrean a F en Base.

## Implementación

La relación de implementación se produce cuando un tipo declara que implementa una interfaz e implementa todos los miembros de tipo de la interfaz. Un tipo que implementa una interfaz determinada se puede convertir a dicha interfaz. No se pueden crear instancias de las interfaces, pero sí se pueden declarar variables de interfaces; a estas variables solamente se les puede asignar un valor de una clase que implemente la interfaz. Por ejemplo:

Interface ITestable  
 Function Test(value As Byte) As Boolean  
End Interface  
  
Class TestableClass  
 Implements ITestable  
  
 Function Test(value As Byte) As Boolean Implements ITestable.Test  
 Return value > 128  
 End Function  
End Class  
  
Module Test  
 Sub F()  
 Dim x As ITestable = New TestableClass  
 Dim b As Boolean  
  
 b = x.Test(34)  
 End Sub  
End Module

Un tipo que implementa una interfaz con miembros de tipo heredados varias veces debe seguir implementando estos métodos, aunque no se pueda tener acceso directamente a ellos desde la interfaz derivada que se implementa. Por ejemplo:

Interface ILeft  
 Sub Test()  
End Interface  
  
Interface IRight  
 Sub Test()  
End Interface  
  
Interface ILeftRight  
 Inherits ILeft, IRight  
End Interface  
  
Class LeftRight  
 Implements ILeftRight  
  
 ' Has to reference ILeft explicitly.  
 Sub TestLeft() Implements ILeft.Test  
 End Sub  
  
 ' Has to reference IRight explicitly.  
 Sub TestRight() Implements IRight.Test  
 End Sub  
  
 ' Error: Test is not available in ILeftRight.  
 Sub TestLeftRight() Implements ILeftRight.Test  
 End Sub  
End Class

Incluso las clases MustInherit deben proporcionar implementaciones de todos los miembros de las interfaces implementadas; sin embargo, pueden diferir la implementación de estos métodos si los declaran como MustOverride. Por ejemplo:

Interface ITest  
 Sub Test1()  
 Sub Test2()  
End Interface  
  
MustInherit Class TestBase  
 Implements ITest  
  
 ' Provides an implementation.  
 Sub Test1() Implements ITest.Test1  
 End Sub  
  
 ' Defers implementation.  
 MustOverride Sub Test2() Implements ITest.Test2  
End Class  
  
Class TestDerived  
 Inherits TestBase  
  
 ' Have to implement MustOverride method.  
 Overrides Sub Test2()  
 End Sub  
End Class

Un tipo puede volver a implementar una interfaz que implementa su tipo base. Para volver a implementar la interfaz, el tipo debe indicar de forma explícita que la implementa. Un tipo que vuelve a implementar una interfaz puede volver a implementar solamente algunos, y no todos, los miembros de la interfaz; los miembros que no se vuelven a implementar siguen usando la implementación del tipo base. Por ejemplo:

Class TestBase  
 Implements ITest  
  
 Sub Test1() Implements ITest.Test1  
 Console.WriteLine("TestBase.Test1")  
 End Sub  
  
 Sub Test2() Implements ITest.Test2  
 Console.WriteLine("TestBase.Test2")  
 End Sub  
End Class  
  
Class TestDerived  
 Inherits TestBase  
 Implements ITest ' Required to re-implement  
  
 Sub DerivedTest1() Implements ITest.Test1  
 Console.WriteLine("TestDerived.DerivedTest1")  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim Test As ITest = New TestDerived()  
 Test.Test1()  
 Test.Test2()  
 End Sub  
End Module

Este ejemplo imprime:

TestDerived.DerivedTest1  
TestBase.Test2

Cuando un tipo derivado implementa una interfaz cuyas interfaces base se implementan en los tipos base del tipo derivado, el tipo derivado solamente puede implementar los miembros de tipo de la interfaz que no implementen ya los tipos base. Por ejemplo:

Interface IBase  
 Sub Base()  
End Interface  
  
Interface IDerived  
 Inherits IBase  
  
 Sub Derived()  
End Interface  
  
Class Base  
 Implements IBase  
  
 Public Sub Base() Implements IBase.Base  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
 Implements IDerived  
  
 ' Required: IDerived.Derived not implemented by Base.  
 Public Sub Derived() Implements IDerived.Derived  
 End Sub  
End Class

Un método de interfaz también se puede implementar con un método invalidable de un tipo base. En este caso, un tipo derivado también puede invalidar el método invalidable y modificar la implementación de la interfaz. Por ejemplo:

Class Base  
 Implements ITest  
  
 Public Sub Test1() Implements ITest.Test1  
 Console.WriteLine("TestBase.Test1")  
 End Sub  
  
 Public Overridable Sub Test2() Implements ITest.Test2  
 Console.WriteLine("TestBase.Test2")  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 ' Overrides base implementation.  
 Public Overrides Sub Test2()  
 Console.WriteLine("TestDerived.Test2")  
 End Sub  
End Class

### Método de implementación

Los tipos implementan los miembros de tipo de una interfaz implementada mediante la incorporación de un método con una cláusula Implements. Los dos miembros de tipo deben tener el mismo número de parámetros, todos los tipos y modificadores de los parámetros deben coincidir, incluido el valor predeterminado de los parámetros opcionales, el tipo devuelto debe coincidir y todas las restricciones de los parámetros del método también deben coincidir. Por ejemplo:

Interface ITest  
 Sub F(ByRef x As Integer)  
 Sub G(Optional y As Integer = 20)  
 Sub H(Paramarray z() As Integer)  
End Interface  
  
Class Test  
 Implements ITest  
  
 ' Error: ByRef/ByVal mismatch.  
 Sub F(x As Integer) Implements ITest.F  
 End Sub  
  
 ' Error: Defaults do not match.  
 Sub G(Optional y As Integer = 10) Implements ITest.G  
 End Sub  
  
 ' Error: Paramarray does not match.  
 Sub H(z() As Integer) Implements ITest.H  
 End Sub  
End Class

Un único método puede implementar varios miembros de tipo de la interfaz si todos cumplen los criterios anteriores. Por ejemplo:

Interface ITest  
 Sub F(i As Integer)  
 Sub G(i As Integer)  
End Interface  
  
Class Test

Implements ITest  
  
 Sub F(i As Integer) Implements ITest.F, ITest.G  
 End Sub  
End Class

Al implementar un método en una interfaz genérica, el método de implementación debe indicar los argumentos de tipo que corresponden a los parámetros de tipo de la interfaz. Por ejemplo:

Interface I1(Of U, V)   
 Sub M(x As U, y As List(Of V))   
End Interface  
  
Class C1(Of W, X)  
 Implements I1(Of W, X)  
  
 ' W corresponds to U and X corresponds to V  
 Public Sub M(x As W, y As List(Of X)) Implements I1(Of W, X).M  
 End Sub   
End Class  
  
Class C2  
 Implements I1(Of String, Integer)  
  
 ' String corresponds to U and Integer corresponds to V  
 Public Sub M(x As String, y As List(Of Integer)) \_  
 Implements I1(Of String, Integer).M  
 End Sub  
End Class

Tenga en cuenta que es posible que una interfaz genérica no se pueda implementar en algún conjunto de argumentos de tipo.

Interface I1(Of T, U)  
 Sub S1(x As T)  
 Sub S1(y As U)  
End Interface  
  
Class C1  
 ' Unable to implement because I1.S1 has two identical signatures  
 Implements I1(Of Integer, Integer)  
End Class

## Polimorfismo

El polimorfismo proporciona la capacidad de variar la implementación de un método o una propiedad. Gracias al polimorfismo, el mismo método o propiedad puede realizar diferentes acciones en función del tipo en tiempo de ejecución de la instancia que lo invoca. Los métodos o las propiedades polimórficos se denominan invalidables. Por el contrario, la implementación de un método o una propiedad no invalidables es invariable; es decir, es la misma tanto si el método o la propiedad se invocan en una instancia de la clase en la que se declararon o en una instancia de una clase derivada. Cuando se invoca a un método o una propiedad no invalidables, el factor determinante es el tipo en tiempo de compilación de la instancia. Por ejemplo:

Class Base  
 Public Overridable Property X() As Integer  
 Get  
 End Get  
  
 Set  
 End Set  
 End Property  
End Class  
  
Class Derived  
 Inherits Base  
  
 Public Overrides Property X() As Integer  
 Get  
 End Get  
  
 Set  
 End Set  
 End Property  
End Class  
  
Module Test  
 Sub F()  
 Dim Z As Base  
  
 Z = New Base()  
 Z.X = 10 ' Calls Base.X  
 Z = New Derived()  
 Z.X = 10 ' Calls Derived.X  
 End Sub  
End Module

Un método invalidable también debe ser MustOverride, lo que significa que no proporciona ningún cuerpo de método y que se debe invalidar. Los métodos MustOverride solamente se permiten en las clases MustInherit.

En el ejemplo siguiente, la clase Shape define la noción abstracta de un objeto de forma geométrica que puede dibujarse a sí mismo:

MustInherit Public Class Shape  
 Public MustOverride Sub Paint(g As Graphics, r As Rectangle)  
End Class   
  
Public Class Ellipse  
 Inherits Shape  
  
 Public Overrides Sub Paint(g As Graphics, r As Rectangle)  
 g.drawEllipse(r)  
 End Sub   
End Class   
  
Public Class Box  
 Inherits Shape  
  
 Public Overrides Sub Paint(g As Graphics, r As Rectangle)  
 g.drawRect(r)  
 End Sub   
End Class

El método Paint es de tipo MustOverride porque no existe una implementación predeterminada significativa. Las clases Ellipse y Box son implementaciones Shape concretas. Ya que estas clases no son de tipo MustInherit, son necesarias para invalidar el método Paint y proporcionar una implementación real.

La referencia a un método MustOverride en un acceso base constituye un error, como se muestra en el siguiente ejemplo:

MustInherit Class A  
 Public MustOverride Sub F()  
End Class  
  
Class B  
 Inherits A  
  
 Public Overrides Sub F()  
 MyBase.F() ' Error, MyBase.F is MustOverride.  
 End Sub   
End Class

En la llamada MyBase.F() se genera un error debido a que hace referencia a un método MustOverride.

### Métodos de invalidación

Un tipo puede invalidar un método invalidable heredado si declara un método con el mismo nombre, la misma signatura y el mismo tipo de valor devuelto (si existe), y marca la declaración con el modificador Overrides. Los métodos invalidables presentan requisitos adicionales, enumerados a continuación. En tanto que una declaración de método Overridable presenta un nuevo método, una declaración de método Overrides reemplaza la implementación heredada del método.

Un método de invalidación se puede declarar como NotOverridable, lo que impide la invalidación posterior del método en tipos derivados. De hecho, los métodos NotOverridable se convierten en no invalidables en las clases derivadas subsiguientes.

Considere el ejemplo siguiente:

Class A  
 Public Overridable Sub F()  
 Console.WriteLine("A.F")  
 End Sub  
  
 Public Overridable Sub G()  
 Console.WriteLine("A.G")  
 End Sub  
End Class  
  
Class B  
 Inherits A  
  
 Public Overrides NotOverridable Sub F()  
 Console.WriteLine("B.F")  
 End Sub  
  
 Public Overrides Sub G()  
 Console.WriteLine("B.G")  
 End Sub  
End Class  
  
Class C  
 Inherits B  
  
 Public Overrides Sub G()  
 Console.WriteLine("C.G")  
 End Sub  
End Class

En el ejemplo, la clase B proporciona dos métodos Overrides: un método F que tiene el modificador NotOverridable y un método G que no lo tiene. El uso del modificador NotOverridable impide que una clase C siga invalidando el método F.

Los métodos de invalidación también se pueden declarar como MustOverride, incluso si el método al que invalidan no se declara como MustOverride. Esto requiere que la clase que los contiene se declare como MustInherit y que las clases derivadas subsiguientes que no se declaren como MustInherit deban invalidar el método. Por ejemplo:

Class A  
 Public Overridable Sub F()  
 Console.WriteLine("A.F")  
 End Sub  
End Class  
  
MustInherit Class B  
 Inherits A  
  
 Public Overrides MustOverride Sub F()  
End Class

En el ejemplo, la clase B invalida A.F con un método MustOverride. Esto significa que las clases que deriven de B tendrán que invalidar F, a menos que también se declaren como MustInherit.

Se produce un error en tiempo de compilación a menos que se cumplan todas las condiciones siguientes de un método de invalidación:

1. El contexto de declaración contiene un único método heredado accesible con la misma firma y el mismo tipo de valor devuelto (si existe) que el método de invalidación.
2. El método heredado que se invalida es invalidable. Es decir, el método heredado que se invalida no está declarado como Shared ni NotOverridable.
3. El dominio de accesibilidad del método que se declara es el mismo que el del método heredado que se invalida. Existe una excepción: el método Protected Friend se debe invalidar con un método Protected si el otro método se encuentra en un ensamblado al que el método de invalidación no tiene acceso de tipo Friend.
4. Los parámetros del método de invalidación coinciden con los parámetros del método invalidado en lo que respecta al uso de los modificadores ByVal, ByRef, ParamArray, y Optional, incluidos los valores proporcionados para los parámetros opcionales.
5. Los parámetros de tipo del método de invalidación coinciden con los parámetros de tipo del método invalidado en lo que respecta a las restricciones de tipo.

Al invalidar un método en un tipo genérico base, el método de invalidación debe indicar los argumentos de tipo que corresponden a los parámetros del tipo base. Por ejemplo:

Class Base(Of U, V)   
 Public Overridable Sub M(x As U, y As List(Of V))   
 End Sub  
End Class  
  
Class Derived(Of W, X)  
 Inherits Base(Of W, X)  
  
 ' W corresponds to U and X corresponds to V  
 Public Overrides Sub M(x As W, y As List(Of X))   
 End Sub   
End Class  
  
Class MoreDerived  
 Inherits Derived(Of String, Integer)  
  
 ' String corresponds to U and Integer corresponds to V  
 Public Overrides Sub M(x As String, y As List(Of Integer))  
 End Sub  
End Class

Es posible que un método invalidable de una clase genérica no se pueda invalidar en algunos conjuntos de argumentos de tipo. Si el método se declara como MustOverride, quizá no se puedan crear algunas cadenas de herencia. Por ejemplo:

MustInherit Class Base(Of T, U)  
 Public MustOverride Sub S1(x As T)  
 Public MustOverride Sub S1(y As U)  
End Class  
  
Class Derived  
 Inherits Base(Of Integer, Integer)  
  
 ' Error: Can't override both S1's at once  
 Public Overrides Sub S1(x As Integer)  
 End Sub  
End Class

Una declaración de invalidación puede obtener acceso al método base invalidado mediante un acceso base, como se muestra en el siguiente ejemplo:

Class Base  
 Private x As Integer  
  
 Public Overridable Sub PrintVariables()  
 Console.WriteLine("x = " & x)  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Private y As Integer  
  
 Public Overrides Sub PrintVariables()  
 MyBase.PrintVariables()  
 Console.WriteLine("y = " & y)  
 End Sub  
End Class

En el ejemplo, la invocación de MyBase.PrintVariables() en la clase Derived invoca la declaración del método PrintVariables en la clase Base. Un acceso base deshabilita el mecanismo de invocación invalidable y simplemente trata el método base como método no invalidable. Si la invocación de Derived se hubiese escrito como CType(Me, Base).PrintVariables(), invocaría de forma recursiva el método PrintVariables declarado en Derived y no el declarado en Base.

Un método solamente puede invalidar otro método si incluye un modificador Overrides. En los demás casos, un método con la misma firma que un método heredado simplemente sombrea el método heredado, como sucede en el ejemplo siguiente:

Class Base  
 Public Overridable Sub F()  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Public Overridable Sub F() ' Warning, shadowing inherited F().  
 End Sub  
End Class

En el ejemplo, el método F de la clase Derived no incluye ningún modificador Overrides, por lo que no invalida el método F de la clase Base. En realidad, el método F de la clase Derived sombrea el método de la clase Base y se emite una advertencia porque la declaración no incluye ningún modificador Shadows ni Overloads.

En el ejemplo siguiente, el método F de la clase Derived sombrea el método invalidable F heredado de la clase Base:

Class Base  
 Public Overridable Sub F()  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Private Shadows Sub F() ' Shadows Base.F within Derived.  
 End Sub  
End Class  
  
Class MoreDerived  
 Inherits Derived  
  
 Public Overrides Sub F() ' Ok, overrides Base.F.  
 End Sub  
End Class

Dado que el nuevo método F de la clase Derived cuenta con acceso de tipo Private, su ámbito solo incluye el cuerpo de clase de Derived y no se extiende a la clase MoreDerived. La declaración del método F en la clase MoreDerived puede, por tanto, invalidar el método F heredado de la clase Base.

Cuando se invoca un método Overridable, se llama a la implementación más derivada del método de instancia en función del tipo de la instancia, con independencia de si la llamada se realiza al método de la clase base o de la clase derivada. La implementación más derivada de un método Overridable M con respecto a una clase R se determina de la siguiente manera:

Si R contiene la declaración Overridable introductora de M, esta es la implementación más derivada de M.

En caso contrario, si R contiene una invalidación de M, esta es la implementación más derivada de M.

En último caso, la implementación más derivada de M es la misma que la de la clase base directa de R.

## Accesibilidad

Una declaración especifica la accesibilidad de la entidad que declara. La accesibilidad de una entidad no cambia el ámbito de su nombre. El dominio de accesibilidad de una declaración es el conjunto de todos los espacios de declaración donde se puede tener acceso a la entidad declarada.

Los cinco tipos de acceso son Public, Protected, Friend, Protected Friend y Private. Public es el tipo de acceso más permisivo y los otros cuatro tipos son subconjuntos de Public. El tipo de acceso menos permisivo es Private y los otros cuatro tipos de acceso son superconjuntos de Private.

El tipo de acceso de una declaración se especifica por medio de un modificador de acceso opcional, que puede ser Public, Protected, Friend, Privateo la combinación de Protected y Friend. Si no se especifica ningún modificador de acceso, el tipo de acceso predeterminado depende del contexto de declaración; los tipos de acceso permitidos también dependen de este contexto.

Las entidades declaradas con el modificador Public cuentan con acceso de tipo Public. No existe ninguna restricción en el uso de entidades de tipo Public.

Las entidades declaradas con el modificador Protected tienen un acceso de tipo Protected. El acceso de tipo Protected solamente se puede especificar en los miembros de las clases (tanto los miembros de tipo normal como las clases anidadas) o en miembros de tipo Overridable de estructuras y módulos estándar (que deben, por definición, heredarse de System.Object o System.ValueType). Las clases derivadas tienen acceso a los miembros de tipo Protected, siempre que el miembro no sea un miembro de instancia o bien siempre que el acceso se realice a través de una instancia de la clase derivada. El acceso de tipo Protected no es un superconjunto del acceso de tipo Friend.

Las entidades declaradas con el modificador Friend cuentan con acceso de tipo Friend. Solamente se puede tener acceso a las entidades con acceso de tipo Friend dentro del programa que contiene la declaración de entidad o de cualquier ensamblado que cuente con acceso de tipo Friend a través del atributo System.Runtime.CompilerServices.InternalsVisibleToAttribute.

Las entidades declaradas con los modificadores Protected Friend cuentan con el acceso resultado de la unión de Protected y Friend.

Las entidades declaradas con el modificador Private cuentan con acceso de tipo Private. El acceso a las entidades Private solamente se puede tener desde su contexto de declaración, incluidas las entidades anidadas.

La accesibilidad de una declaración no depende de la accesibilidad del contexto de declaración. Por ejemplo, un tipo declarado con acceso Private puede contener un miembro de tipo con acceso Public.

En el código siguiente se muestran varios dominios de accesibilidad:

Public Class A  
 Public Shared X As Integer  
 Friend Shared Y As Integer  
 Private Shared Z As Integer  
End Class  
  
Friend Class B  
 Public Shared X As Integer  
 Friend Shared Y As Integer  
 Private Shared Z As Integer  
  
 Public Class C  
 Public Shared X As Integer  
 Friend Shared Y As Integer  
 Private Shared Z As Integer  
 End Class  
  
 Private Class D  
 Public Shared X As Integer  
 Friend Shared Y As Integer  
 Private Shared Z As Integer  
 End Class  
End Class

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 programa contenedor.

El dominio de accesibilidad de A.Z es A.

El dominio de accesibilidad de B.Z, B.D, B.D.X y B.D.Y es B, incluidos B.C y B.D.

El dominio de accesibilidad de B.C.Z es B.C.

El dominio de accesibilidad de B.D.Z es 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.

El acceso a los miembros de instancia declarados como Protected se debe realizar a través de una instancia del tipo derivado, de forma que los tipos no relacionados no pueden obtener acceso a los miembros protegidos de los demás. Por ejemplo:

Class User  
 Protected Password As String  
End Class  
  
Class Employee  
 Inherits User  
End Class  
  
Class Guest  
 Inherits User  
  
 Public Function GetPassword(u As User) As String  
 ' Error: protected access has to go through derived type.  
 Return U.Password  
 End Function  
End Class

En el ejemplo anterior, la clase Guest solo tiene acceso al campo Password protegido si se califica con una instancia de Guest. Esto impide que Guest obtenga acceso al campo Password de un objeto Employee simplemente mediante su conversión a User.

Para el acceso de miembros declarados como Protected en tipos genéricos, el contexto de declaración incluye parámetros de tipo. Esto significa que un tipo derivado con un conjunto de argumentos de tipo no tiene acceso a los miembros declarados como Protected de un tipo derivado con un conjunto diferente de argumentos de tipo. Por ejemplo:

Class Base(Of T)  
 Protected x As T  
End Class  
  
Class Derived(Of T)  
 Inherits Base(Of T)  
  
 Public Sub F(y As Derived(Of String))  
 ' Error: Derived(Of T) cannot access Derived(Of String)'s   
 ' protected members  
 y.x = "a"  
 End Sub  
End Class

Anotación

El lenguaje C# (y posiblemente otros lenguajes) permite que un tipo genérico tenga acceso a los miembros declarados como Protected independientemente de los argumentos de tipo que se proporcionen. Es necesario recordar esta característica a la hora de diseñar clases genéricas que contengan miembros declarados como Protected.

AccessModifier ::= Public | Protected | Friend | Private | Protected Friend

### Tipos constituyentes

Los tipos constituyentes de una declaración son los tipos a los que hace referencia la declaración. Por ejemplo, son tipos constituyentes el tipo de una constante, el tipo de valor devuelto de un método y los tipos de parámetro de un constructor. El dominio de accesibilidad del tipo constituyente de una declaración debe ser el mismo o un superconjunto del dominio de accesibilidad de la propia declaración. Por ejemplo:

Public Class X  
 Private Class Y  
 End Class  
  
 ' Error: Exposing private class Y outside of X.  
 Public Function Z() As Y  
 End Function  
  
 ' Valid: Not exposing outside of X.  
 Private Function A() As Y  
 End Function  
End Class  
  
Friend Class B  
 Private Class C  
 End Class  
  
 ' Error: Exposing private class Y outside of B.  
 Public Function D() As C  
 End Function  
End Class

## Nombres de tipos y de espacios de nombres

Muchas construcciones de lenguaje requieren que se indiquen un espacio de nombres o un tipo; estos elementos se pueden especificar mediante la forma calificada del nombre del espacio de nombres o del tipo. El nombre calificado consta de una serie de identificadores separados por puntos; el identificador situado a la derecha de un punto se resuelve en el espacio de declaración que especifica el identificador situado a la izquierda del punto.

El nombre completo de un espacio de nombres o un tipo es un nombre calificado que contiene el nombre de todos los espacios de nombres y tipos que lo contienen. Es decir, el nombre completo de un espacio de nombres o un tipo es N.T donde T es el nombre de la entidad y N el nombre completo de su entidad contenedora.

En el ejemplo siguiente se muestran varias declaraciones de espacio de nombres y tipos con sus nombres completos asociados en comentarios incluidos en la misma línea.

Class A ' A.  
End Class  
  
Namespace X ' X.  
 Class B ' X.B.  
 Class C ' X.B.C.  
 End Class  
 End Class  
  
 Namespace Y ' X.Y.  
 Class D ' X.Y.D.  
 End Class  
 End Namespace   
End Namespace   
  
Namespace X.Y ' X.Y.  
 Class E ' X.Y.E.  
 End Class  
End Namespace

Observe que el espacio de nombres X.Y se ha declarado en dos ubicaciones diferentes en el código fuente, pero estas dos declaraciones parciales constituyen un solo espacio de nombres llamado X.Y que incluye tanto la clase D como la clase E.

En algunos casos, los nombres calificados pueden comenzar con la palabra clave Global. Esta palabra clave representa el espacio de nombres más externo sin nombre y resulta útil en los casos en que una declaración sombrea un espacio de nombres envolvente. En esa situación, la palabra clave Global permite “escapar” del espacio de nombres más externo. Por ejemplo:

Namespace NS1  
 Class System  
 End Class  
  
 Module Test  
 Sub Main()  
 ' Error: Class System does not contain Int32  
 Dim x As System.Int32  
  
  
 ' Legal, binds to System in outermost namespace  
 Dim y As Global.System.Int32  
 End Sub  
 End Module  
End Namespace

En el ejemplo anterior, la primera llamada al método no es válida porque el identificador System enlaza con la clase System, no con el espacio de nombres System. La única manera de obtener acceso al espacio de nombres System consiste en usar Global para escapar del espacio de nombres más externo. Global no se puede usar en una instrucción Imports ni en una declaración Namespace.

Dado que otros lenguajes pueden incluir tipos y espacios de nombres que coincidan con palabras claves de Visual Basic, este lenguaje reconoce las palabras clave como parte de un nombre calificado siempre que sigan a un punto. Las palabras clave usadas de este modo se consideran identificadores. Por ejemplo, el identificador calificado X.Default.Class es un identificador calificado válido, en tanto que Default.Class no lo es.

### Resolución de nombres calificados para los espacios de nombres y tipos

Dado un nombre de espacio de nombres o de tipo calificado con el formato N.R(Of A), donde R es el identificador situado más a la derecha del nombre calificado y A una lista de argumentos de tipo opcional, los pasos siguientes describen cómo se determina el espacio de nombres o el tipo al que se refiere el nombre calificado:

1. Resuelva N mediante las reglas para la resolución de nombres calificados o no calificados.
2. Si se produce un error en la resolución de N o se resuelve en un parámetro de tipo, se produce un error en tiempo de compilación.
3. En caso contrario, si R coincide con el nombre de un espacio de nombres en N y no se proporcionaron argumentos de tipo, o R coincide con un tipo accesible en N con el mismo número de parámetros de tipo que de argumentos de tipo (si existen), el nombre calificado se refiere a dicho espacio de nombres o tipo.
4. Si N contiene uno o varios módulos estándar y R coincide con el nombre de un tipo accesible con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) de un único módulo estándar, el nombre calificado se refiere a dicho tipo. Si R coincide con el nombre de tipos accesibles con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) en varios módulos estándar, se produce un error en tiempo de compilación.
5. En caso contrario, se producirá un error en tiempo de compilación.

Nota   Este proceso de resolución implica que los miembros de tipo no sombrean los espacios de nombres ni los tipos al resolver sus nombres.

### Resolución de nombres no calificados para los espacios de nombres y tipos

Dado un nombre no calificado R(Of A), donde A es una lista de argumentos de tipo opcional, los siguientes pasos describen cómo se determina el espacio de nombres o el tipo al que se refiere el nombre no calificado:

1. En caso contrario, si R coincide con el nombre de un parámetro de tipo del método actual y no se han proporcionado argumentos de tipo, el nombre no calificado se refiere a dicho parámetro de tipo.
2. En cada tipo anidado que contiene la referencia de nombre, comenzando por el más interno y yendo hacia el más externo:
3. Si R coincide con el nombre de un parámetro de tipo del tipo actual y no se han proporcionado argumentos de tipo, el nombre no calificado se refiere a dicho parámetro de tipo.
4. En caso contrario, si R coincide con el nombre de un tipo anidado accesible con el mismo número de parámetros de tipo que de argumentos de tipo (si existen), el nombre no calificado se refiere a dicho tipo.
5. En cada espacio de nombres anidado que contiene la referencia de nombre, comenzando por el más interno y yendo hacia el más externo:
6. Si R coincide con el nombre de un espacio de nombres anidado en el espacio de nombres actual y no se ha proporcionado ninguna lista de argumentos, el nombre no calificado se refiere a dicho espacio de nombres anidado.
7. En caso contrario, si R coincide con el nombre de un tipo accesible con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) del espacio de nombres actual, el nombre no calificado se refiere a dicho tipo.
8. Por lo demás, si el espacio de nombres contiene uno o varios módulos estándar accesibles y R coincide con el nombre de un tipo anidado accesible con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) de un único módulo estándar, el nombre no calificado se refiere a dicho tipo anidado. Si R coincide con el nombre de tipos anidados accesibles con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) en varios módulos estándar, se produce un error en tiempo de compilación.
9. Si el archivo de origen tiene uno o más alias de importación y R coincide con el nombre de uno de ellos, el nombre no calificado se refiere a dicho alias de importación. Si se proporciona una lista de argumentos de tipo, se produce un error en tiempo de compilación.
10. Si el archivo de origen que contiene la referencia de nombre tiene una o más importaciones:
11. Si R coincide con el nombre de un tipo accesible con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) de una única importación, el nombre no calificado se refiere a dicho tipo. Si R coincide con el nombre un tipo accesible con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) en varias importaciones y no todos son del mismo tipo, se produce un error en tiempo de compilación.
12. De lo contrario, si no hay lista de argumentos de tipo y R coincide con el nombre de un espacio de nombres con tipos accesibles exactamente en una importación, el nombre no calificado hace referencia a ese espacio de nombres. Si no se proporcionó ninguna lista de argumentos de tipo, R coincide con el nombre de un espacio de nombres con tipos accesibles en más de una importación y no todos son el mismo espacio de nombres, se produce un error en tiempo de compilación.
13. Por lo demás, si las importaciones contienen uno o varios módulos estándar accesibles y R coincide con el nombre de un tipo anidado accesible con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) de un único módulo estándar, el nombre no calificado se refiere a dicho tipo. Si R coincide con el nombre de tipos anidados accesibles con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) en varios módulos estándar, se produce un error en tiempo de compilación.
14. Si el entorno de compilación define uno o más alias de importación y R coincide con el nombre de uno de ellos, el nombre no calificado se refiere a dicho alias de importación. Si se proporciona una lista de argumentos de tipo, se produce un error en tiempo de compilación.
15. Si el entorno de compilación define una o más importaciones:
16. Si R coincide con el nombre de un tipo accesible con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) de una única importación, el nombre no calificado se refiere a dicho tipo. Si R coincide con el nombre de un tipo accesible con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) en varias importaciones, se produce un error en tiempo de compilación.
17. De lo contrario, si no hay lista de argumentos de tipo y R coincide con el nombre de un espacio de nombres con tipos accesibles exactamente en una importación, el nombre no calificado hace referencia a ese espacio de nombres. Si no se proporcionó una lista de argumentos de tipo y R coincide con el nombre de un espacio de nombres con tipos accesibles en más de una importación, se produce un error en tiempo de compilación.
18. Por lo demás, si las importaciones contienen uno o varios módulos estándar accesibles y R coincide con el nombre de un tipo anidado accesible con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) de un único módulo estándar, el nombre no calificado se refiere a dicho tipo. Si R coincide con el nombre de tipos anidados accesibles con el mismo número de parámetros de tipo que de argumentos de tipo (si existen) en varios módulos estándar, se produce un error en tiempo de compilación.
19. En caso contrario, se producirá un error en tiempo de compilación.

Nota   Este proceso de resolución implica que los miembros de tipo no sombrean los espacios de nombres ni los tipos al resolver sus nombres.

Normalmente un nombre solo aparece una vez en un espacio de nombres concreto. Sin embargo, como los espacios de nombres se pueden declarar entre múltiples ensamblados de .NET, es posible que se dé una situación en la que dos ensamblados definen un tipo con el mismo nombre completo. En ese caso, se prefiere un tipo declarado en el grupo de archivos de origen actual a un tipo declarado en un ensamblado de .NET externo. De otro modo, el nombre es ambiguo y no hay forma de romper esa ambigüedad.

## Variables

Las variables representan ubicaciones de almacenamiento. Toda variable tiene un tipo, que determina los valores que pueden almacenarse en ella. Visual Basic es un lenguaje con seguridad de tipos, por lo que todas las variables de un programa disponen de un tipo y el lenguaje garantiza que los valores almacenados en variables siempre son del tipo apropiado. Las variables se inicializan siempre en el valor predeterminado de su tipo antes de que se realicen referencias a ellas. No es posible tener acceso a memoria no inicializada.

## Tipos y métodos genéricos

Los tipos (excepto los módulos estándar y los tipos enumerados) y los métodos pueden declarar parámetros de tipo, es decir, tipos que no se proporcionarán hasta que se declare una instancia del tipo o se invoque el método. Los tipos y métodos con parámetros de tipo se conocen también como tipos genéricos y métodos genéricos, respectivamente, porque el tipo o el método se deben escribir de forma genérica, sin conocimiento específico de los tipos que se proporcionarán en el código que usa dicho tipo o método.

Anotación

En este momento, aunque los métodos y los delegados pueden ser genéricos, las propiedades, los eventos y los operadores no lo pueden ser. Pueden, sin embargo, usar parámetros de tipo de la clase contenedora.

Desde la perspectiva del tipo o el método genérico, un parámetro de tipo es un tipo marcador que se llenará con un tipo real al usar el tipo o el método. Los argumentos de tipo se reemplazan por los parámetros de tipo en el tipo o el método en el punto en el que se usan estos últimos. Por ejemplo, una clase de pila genérica se puede implementar como:

Public Class Stack(Of ItemType)  
 Protected Items(0 To 99) As ItemType  
 Protected CurrentIndex As Integer = 0  
  
 Public Sub Push(data As ItemType)  
 If CurrentIndex = 100 Then  
 Throw New ArgumentException("Stack is full.")  
 End If  
  
 Items(CurrentIndex) = Data  
 CurrentIndex += 1  
 End Sub  
  
 Public Function Pop() As ItemType  
 If CurrentIndex = 0 Then  
 Throw New ArgumentException("Stack is empty.")  
 End If  
  
 CurrentIndex -= 1  
 Return Items(CurrentIndex + 1)   
 End Function  
End Class

Las declaraciones que usan la clase Stack(Of ItemType) deben proporcionar un argumento de tipo para el parámetro de tipo ItemType. Este tipo se llena al usar ItemType dentro de la clase:

Option Strict On  
  
Module Test  
 Sub Main()  
 Dim s1 As New Stack(Of Integer)()  
 Dim s2 As New Stack(Of Double)()  
  
 s1.Push(10.10) ' Error: Stack(Of Integer).Push takes an Integer  
 s2.Push(10.10) ' OK: Stack(Of Double).Push takes a Double  
 Console.WriteLine(s2.Pop().GetType().ToString()) ' Prints: Double  
 End Sub  
End Module

### Parámetros de tipo

Los parámetros de tipo se pueden proporcionar en declaraciones de tipo o de método. Cada parámetro de tipo es un identificador que denota un marcador de posición para un argumento de tipo proporcionado para crear un tipo o un método construido. Por el contrario, un argumento de tipo es el tipo real que se sustituye por el parámetro de tipo al usar un tipo o un método genérico.

Cada parámetro de tipo de una declaración de tipos o método define un nombre en el espacio de declaración de dicho tipo o método. Así, no puede tener el mismo nombre que otro parámetro de tipo, que un miembro de tipo, que un parámetro de método o que una variable local. El ámbito de un parámetro de tipo de un tipo o un método es el tipo o el método completo. El ámbito de los parámetros de tipo es la declaración de tipos completa, por lo que los tipos anidados pueden usar parámetros de tipo externos. Esto también implica que los parámetros de tipo siempre se deben especificar al tener acceso a tipos anidados dentro de tipos genéricos:

Public Class Outer(Of T)  
 Public Class Inner  
 Public Sub F(x As T)  
 ...  
 End Sub  
 End Class  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As New Outer(Of Integer).Inner()  
 ...  
 End Sub  
End Module

A diferencia de otros miembros de una clase, los parámetros de tipo no se heredan. Solo se puede hacer referencia a los parámetros de tipo de un tipo por su nombre simple; es decir, no se pueden calificar con el nombre de tipo contenedor. Aunque el estilo de programación no es correcto, el parámetro de tipo en un tipo anidado puede ocultar un parámetro de miembro o de tipo declarado en el tipo externo:

Class Outer(Of T)  
 Class Inner(Of T)  
 Public t1 As T ' Refers to Inner's T  
 End Class  
End Class

Los tipos y los métodos se pueden sobrecargar en función del número de parámetros de tipo (o aridad) que declaran los tipos o métodos. Por ejemplo, las siguientes declaraciones son válidas:

Module C  
 Sub M()  
 End Sub  
  
 Sub M(Of T)()  
 End Sub  
  
 Sub M(Of T, U)()  
 End Sub  
End Module  
  
Structure C(Of T)  
 Dim x As T  
End Structure  
  
Class C(Of T, U)  
End Class

En el caso de los tipos, las sobrecargas siempre se comparan con el número de argumentos de tipo especificados. Esto resulta útil al usar tanto clases genéricas como no genéricas en el mismo programa:

Class Queue   
End Class   
   
Class Queue(Of T)  
End Class  
  
Class X  
 Dim q1 As Queue ' Non-generic queue  
 Dim q2 As Queue(Of Integer) ' Generic queue  
End Class

Las reglas para los métodos sobrecargados en los parámetros de tipo se tratan en la sección sobre resolución de sobrecargas de método.

Los parámetros de tipo se consideran tipos completos dentro de la declaración contenedora. 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 que otros tipos, según se describe a continuación:

Un parámetro de tipo no se puede utilizar directamente para declarar una interfaz o una clase base.

Las reglas para la búsqueda de miembros en parámetros de tipo dependen de las restricciones, si existen, que se aplican al parámetro de tipo.

Las conversiones disponibles para un parámetro de tipo dependen de las restricciones, si existen, que se aplican a los parámetros de tipo.

En ausencia de una restricción Structure, un valor con un tipo representado mediante un parámetro de tipo se puede comparar con Nothing mediante Is e IsNot.

Un parámetro de tipo solo se puede usar en una expresión New si el parámetro de tipo está restringido por una restricción New o Structure.

Un parámetro de tipo no se puede usar dentro de una excepción de atributo incluida en una expresión GetType.

Los parámetros de tipo se pueden usar como argumentos de tipo de otros parámetros y tipos genéricos. En el siguiente ejemplo se muestra un tipo genérico que extiende la clase Stack(Of ItemType):

Class MyStack(Of ItemType)  
 Inherits Stack(Of ItemType)  
  
 Public ReadOnly Property Size() As Integer  
 Get  
 Return CurrentIndex  
 End Get  
 End Property  
End Class

Si una declaración proporciona un argumento de tipo a MyStack, este mismo argumento de tipo se aplicará también a Stack.

Al igual que un tipo, los parámetros de tipo son una construcción en tiempo de compilación. En tiempo de ejecución, cada parámetro de tipo se enlaza a un tipo en tiempo de ejecución que se especifica al proporcionar un argumento de tipo a la declaración genérica. Por lo tanto, el tipo de una variable declarado con un parámetro de tipo será, en tiempo de ejecución, un tipo no genérico o un tipo construido específico. 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.

TypeParameterList ::=  
 OpenParenthesis Of TypeParameters CloseParenthesis

TypeParameters ::=  
 TypeParameter |  
 TypeParameters Comma TypeParameter

TypeParameter ::=  
 [ VarianceModifier ] Identifier [ TypeParameterConstraints ]

VarianceModifier ::=  
 In | Out

### Restricciones de tipo

Los argumentos de tipo pueden corresponder a cualquier tipo del sistema de tipos, por lo que un tipo o un método genérico no puede realizar ninguna suposición sobre el parámetro de tipo. Así, los miembros de un parámetro de tipo se consideran miembros del tipo Object, ya que todos los tipos derivan de Object.

En el caso de una colección como Stack(Of ItemType), este hecho puede que no suponga una restricción especialmente importante, pero es posible que haya casos en los que un tipo genérico desee suponer los tipos que se proporcionarán como argumentos de tipo. En los parámetros de tipo se pueden incluir restricciones de tipo que limitan los tipos que se pueden proporcionar como parámetro de tipo y que permiten que los tipos o métodos genéricos asuman más información sobre los parámetros de tipo.

Public Class DisposableStack(Of ItemType As IDisposable)  
 Implements IDisposable  
  
 Private \_items(0 To 99) As ItemType  
 Private \_currentIndex As Integer = 0  
  
 Public Sub Push(data As ItemType)  
 ...  
 End Sub  
  
 Public Function Pop() As ItemType  
 ...  
 End Function  
  
 Private Sub Dispose() Implements IDisposable.Dispose  
 For Each item As IDisposable In \_items  
 If item IsNot Nothing Then  
 item.Dispose()  
 End If  
 Next item  
 End Sub  
End Class

En este ejemplo, DisposableStack(Of ItemType) limita su parámetro de tipo a los tipos que implementan la interfaz System.IDisposable. Como resultado, puede implementar un método Dispose que desecha los objetos que permanecen en la cola.

Las restricciones de tipo deben pertenecer a las restricciones especiales Class, Structure o New, o bien deben ser del tipo T donde:

T es una clase, una interfaz o un parámetro de tipo.

T no es NotInheritable.

T no es uno de los siguientes tipos especiales ni un tipo heredado de ellos: System.Array, System.Delegate, System.MulticastDelegate, System.Enum o System.ValueType.

T no es Object. Puesto que todos los tipos derivan de Object, dicha restricción no tendría efecto si se permitiera.

T debe ser al menos tan accesible como el tipo o método genérico que se declara.

En un único parámetro de tipo se pueden especificar varias restricciones de tipo, si se incluyen entre llaves ({}). Solo una restricción de tipo de un parámetro de tipo dado puede ser una clase. Es un error combinar una restricción Structure especial con una limitación de clase con nombre o la restricción Class especial.

Class ControlFactory(Of T As {Control, New})  
 ...  
End Class

Las restricciones de tipo pueden usar los tipos contenedores o los parámetros de tipo de estos. En el siguiente ejemplo, la restricción requiere que el argumento de tipo proporcionado implemente una interfaz genérica usándose a sí mismo como argumento de tipo:

Class Sorter(Of V As IComparable(Of V))  
 ...  
End Class

La restricción de tipo especial Class limita el argumento de tipo proporcionado a cualquier tipo de referencia.

Anotación

La restricción de tipo especial Class se puede satisfacer mediante una interfaz. Y una estructura puede implementar una interfaz. Por tanto, las restricciones (Of T As U, U As Class) se pueden cumplir con “T” una estructura (que no cumple la restricción especial Class) y “U” una interfaz que implementa (que sí cumple la restricción especial Class).

La restricción de tipo especial Structure limita el argumento de tipo proporcionado a cualquier tipo de valor excepto System.Nullable(Of T).

Anotación

Las restricciones de estructura no permiten System.Nullable(Of T), por lo que no es posible proporcionar System.Nullable(Of T) como argumento de tipo a sí mismo.

La restricción de tipo especial New requiere que el argumento de tipo proporcionado incluya un constructor accesible sin parámetros y no se pueda declarar como MustInherit. Por ejemplo:

Class Factory(Of T As New)  
 Function CreateInstance() As T  
 Return New T()  
 End Function  
End Class

Las restricciones de tipo de clase requieren que el argumento de tipo proporcionado sea de dicho tipo o herede de él. Las restricciones de tipo de interfaz requieren que el argumento de tipo proporcionado implemente dicha interfaz. Las restricciones de parámetro de tipo requieren que el argumento de tipo proporcionado derive de todos los enlaces dados del parámetro de tipo coincidente o los implemente. Por ejemplo:

Class List(Of T)  
 Sub AddRange(Of S As T)(collection As IEnumerable(Of S))  
 ...  
 End Sub  
End Class

En este ejemplo, el parámetro de tipo S de AddRange se limita al parámetro de tipo T de List. Esto implica que List(Of Control) limitaría el parámetro de tipo de AddRange a cualquier tipo que sea o herede de Control.

Una restricción de parámetro de tipo "Of S As T" se resuelve agregando transitivamente todas las restricciones de T a S, que no sean las restricciones especiales (Class, Structure, New). Es un error tener restricciones circulares (p. ej. Of S As T, T As S). Es un error tener una restricción de parámetro de tipo que tenga la restricción Structure. Tras agregar restricciones, es posible que se produzcan varias situaciones especiales:

Si existen varias restricciones de clase, se considera que la clase más derivada es la restricción. Si una o varias restricciones de clase no tienen relación de herencia, la restricción no se puede satisfacer y es un error.

Si un tipo de parámetro combina una restricción especial Structure con una restricción de clase con nombre o la restricción especial Class, es un error. Una restricción de clase puede ser NotInheritable, en cuyo caso no se aceptan tipos derivados de esa restricción y es un error.

El tipo puede ser uno de los siguientes tipos especiales o un tipo heredado de ellos: System.Array, System.Delegate, System.MulticastDelegate, System.Enum o System.ValueType. En ese caso, solo se acepta el tipo, o un tipo heredado de él. Un parámetro de tipo restringido a uno de estos tipos solo puede usar las conversiones que permita el operador DirectCast. Por ejemplo:

MustInherit Class Base(Of T)  
 MustOverride Sub S1(Of U As T)(x As U)  
End Class  
  
Class Derived  
 Inherits Base(Of Integer)  
  
 ' The constraint of U must be Integer, which is normally not allowed.  
 Overrides Sub S1(Of U As Integer)(x As U)  
 Dim y As Integer = x ' OK  
 Dim z As Long = x ' Error: Can't convert  
 End Sub  
End Class

Además, los parámetros de tipo restringidos a un tipo de valor debido a una de las relajaciones anteriores no puede llamar a ningún método definido en dicho tipo de valor. Por ejemplo:

Class C1(Of T)  
 Overridable Sub F(Of G As T)(x As G)  
 End Sub  
End Class  
  
Class C2  
 Inherits C1(Of IntPtr)  
  
 Overrides Sub F(Of G As IntPtr)(ByVal x As G)  
 ' Error: Cannot access structure members  
 x.ToInt32()  
 End Sub  
End Class

Si la restricción, tras la sustitución, finaliza como un tipo de matriz, también se permiten los tipos de matriz covariantes. Por ejemplo:

Module Test  
 Class B  
 End Class  
  
 Class D  
 Inherits B  
 End Class  
  
 Function F(Of T, U As T)(x As U) As T  
 Return x  
 End Function  
  
 Sub Main()  
 Dim a(9) As B  
 Dim b(9) As D  
  
 a = F(Of B(), D())(b)  
 End Sub  
End Module

Se considera que un parámetro de tipo con una restricción de clase o interfaz tiene los mismos miembros que la restricción. Si un parámetro de tipo tiene varias restricciones, se considera que tiene la unión de todos los miembros de estas. Si existen miembros con el mismo nombre en varias restricciones, los miembros se ocultan según el siguiente orden: la restricción de clase oculta los miembros de las restricciones de interfaz, que ocultan los miembros de System.ValueType (si se especifica la restricción Structure), que oculta los miembros de Object. Si en varias restricciones de interfaz aparece un miembro con el mismo nombre, el miembro no estará disponible (como sucede en la herencia de varias interfaces) y el parámetro de tipo deberá convertirse a la interfaz deseada. Por ejemplo:

Class C1  
 Sub S1(x As Integer)  
 End Sub  
End Class  
  
Interface I1  
 Sub S1(x As Integer)  
End Interface  
  
Interface I2  
 Sub S1(y As Double)  
End Interface  
  
Module Test  
 Sub T1(Of T As {C1, I1, I2})()  
 Dim a As T  
 a.S1(10) ' Calls C1.S1, which is preferred  
 a.S1(10.10) ' Also calls C1.S1, class is still preferred  
 End Sub  
  
 Sub T2(Of T As {I1, I2})()  
 Dim a As T  
 a.S1(10) ' Error: Call is ambiguous between I1.S1, I2.S1  
 End Sub  
End Module

Si se proporcionan parámetros de tipo como argumentos de tipo, los parámetros deben satisfacer las restricciones de los parámetros de tipo con los que coinciden.

Class Base(Of T As Class)  
End Class  
  
Class Derived(Of V)  
 ' Error: V does not satisfy the constraints of T  
 Inherits Base(Of V)  
End Class

Se pueden utilizar valores de un parámetro de tipo restringido para obtener acceso a los miembros de instancia, incluidos los métodos de instancia, especificados en la restricción.

Interface IPrintable  
 Sub Print()  
End Interface  
  
Class Printer(Of V As IPrintable)  
 Sub PrintOne(v1 As V)  
 V1.Print()  
 End Sub  
End Class

TypeParameterConstraints ::=  
 As Constraint |  
 As OpenCurlyBrace ConstraintList CloseCurlyBrace

ConstraintList ::=  
 ConstraintList *Comma* Constraint |  
 Constraint

Constraint ::= TypeName | New | Structure | Class

### Varianza de los parámetros de tipo

Un parámetro de tipo en una declaración de tipos de delegado o interfaz puede, opcionalmente, especificar un modificador de varianza. Los parámetros de tipo con modificadores de varianza restringen la forma en que el parámetro de tipo se puede usar en el tipo de delegado o interfaz, pero permiten que un tipo de delegado o interfaz genérico se convierta en otro tipo genérico con argumentos de tipo compatibles con la variante. Por ejemplo:

Class Base  
End Class  
  
Class Derived  
 Inherits Base  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As IEnumerable(Of Derived) = ...  
  
 ' OK, as IEnumerable(Of Base) is variant compatible  
 ' with IEnumerable(Of Derived)  
 Dim y As IEnumerable(Of Base) = x  
 End Sub  
End Module

Las interfaces genéricas que incluyen parámetros de tipo con modificadores de varianza presentan varias restricciones:

No pueden contener una declaración de evento que especifique una lista de parámetros (aunque se permiten las declaraciones de evento personalizadas o las declaraciones de evento con un tipo de delegado).

No pueden contener una clase, estructura o tipo enumerado anidado.

Anotación

Estas restricciones se deben al hecho de que los tipos anidados en tipos genéricos copian de forma implícita los parámetros genéricos de su elemento primario. En el caso de las clases, estructuras o tipos enumerados anidados, estas clases de tipos no pueden incluir modificadores de varianza en sus parámetros de tipo. En el caso de una declaración de evento con una lista de parámetros, la clase de delegado anidada generada podría generar errores de confusión si un tipo que parece ser usado en una posición In (es decir, un tipo de parámetro) se usa de hecho en una posición Out (es decir, el tipo del evento).

Un parámetro de tipo que se declara con el modificador Out es covariante. Dicho de manera informal, un parámetro de tipo covariante solo se puede usar en una posición de salida (es decir, el valor que se devuelve del tipo de interfaz o delegado) y no se puede usar en una posición de entrada. Un tipo T se considera covariante válido si:

T es una clase, estructura o tipo enumerado.

T es un tipo de delegado o interfaz no genérico.

T es un tipo de matriz cuyo tipo de elemento es covariante válido.

T es un parámetro de tipo que no se ha declarado como parámetro de tipo Out.

T es un tipo de interfaz o delegado construido X(Of P1,…,Pn) con argumentos de tipo A1,…,An de manera que:

Si Pi se declaró como parámetro de tipo Out Ai es covariante válido.

Si Pi se declaró como parámetro de tipo In Ai es contravariante válido.

Los siguientes elementos deben ser covariantes válidos en un tipo de interfaz o delegado:

La interfaz base de una interfaz.

El tipo de valor devuelto de una función o el tipo de delegado.

El tipo de una propiedad si hay un descriptor de acceso Get.

El tipo de cualquier parámetro ByRef.

Por ejemplo:

Delegate Function D(Of Out T, U)(x As U) As T  
  
Interface I1(Of Out T)  
End Interface  
  
Interface I2(Of Out T)  
 Inherits I1(Of T)  
  
 ' OK, T is only used in an Out position  
 Function M1(x As I1(Of T)) As T  
  
 ' Error: T is used in an In position  
 Function M2(x As T) As T  
End Interface

Un parámetro de tipo que se declara con el modificador In es contravariante. Dicho de manera informal, un parámetro de tipo contravariante solo se puede usar en una posición de entrada (es decir, el valor que se pasa al tipo de delegado o interfaz) y no se puede usar en una posición de salida. Un tipo T se considera contravariante válido si:

T es una clase, estructura o tipo enumerado.

T es un tipo de delegado o interfaz no genérico.

T es un tipo de matriz cuyo tipo de elemento es contravariante válido.

T es un parámetro de tipo que no se declaró como In parámetro de tipo.

T es un tipo de interfaz o delegado construido X(Of P1,…,Pn) con argumentos de tipo A1,…,An de manera que:

Si Pi se declaró como parámetro de tipo Out, Ai es contravariante válido.

Si Pi se declaró como parámetro de tipo In, Ai es covariante válido.

Los siguientes elementos deben ser contravariantes válidos en un tipo de interfaz o delegado:

El tipo de un parámetro.

Una restricción de tipo en un parámetro de tipo de método.

El tipo de una propiedad si tiene un descriptor de acceso Set.

El tipo de un evento.

Por ejemplo:

Delegate Function D(Of T, In U)(x As U) As T  
  
Interface I1(Of In T)  
End Interface  
  
Interface I2(Of In T)  
 ' OK, T is only used in an In position  
 Sub M1(x As I1(Of T))  
  
 ' Error: T is used in an Out position  
 Function M2() As T  
End Interface

Si un tipo debe ser contravariante y covariante válido (como en el caso de una propiedad con un descriptor de acceso Get y Set o un parámetro ByRef), no se puede usar un parámetro de tipo variante.

Nota Out no es una palabra reservada.

Anotación

La covarianza y la contravarianza dan lugar a un "problema de ambigüedad del diamante". Observe el código siguiente:

Clase C  
 Implements IEnumerable(Of String)  
 Implements IEnumerable(Of Exception)  
  
 Public Function GetEnumerator1() As IEnumerator(Of String) \_  
 Implements IEnumerable(Of String).GetEnumerator  
 Console.WriteLine(“string”)  
 End Function  
  
 Public Function GetEnumerator2() As IEnumerator(Of Exception) \_  
 Implements IEnumerable(Of Execption).GetEnumerator  
 Console.WriteLine(“exception”)  
 End Function  
End Class

Dim c As IEnumerable(Of Object) = New C  
c.GetEnumerator()

La clase C se puede convertir en IEnumerable(Of Object) de dos formas, ambas mediante la conversión covariante de IEnumerable(Of String) y mediante la conversión covariante de IEnumerable(Of Exception). CLR no especifica a cuál de los dos métodos llamará c.GetEnumerator(). En general, siempre que se declara una clase para implementar una interfaz covariante con dos argumentos genéricos diferentes que tienen un supertipo común (p. ej., en este caso, String y Exception tienen el supertipo común Object), o se declara una clase para implementar una interfaz contravariante con dos argumentos genéricos diferentes que tienen un subtipo común, es probable que surja ambigüedad. El compilador da una advertencia en cuanto a dichas declaraciones.

# Atributos

El lenguaje Visual Basic permite a los programadores especificar modificadores en las declaraciones, que representan información sobre las entidades que se declaran. Por ejemplo, si en un método de clase se anexan los modificadores Public, Protected, Friend, Protected Friend o Private, se especifica su accesibilidad.

Además de los modificadores definidos en el lenguaje, Visual Basic también permite que los programadores creen nuevos modificadores, denominados atributos, y que los usen en la declaración de nuevas entidades. Estos nuevos modificadores, que se definen mediante la declaración de clases de atributos, se asignan a las entidades mediante bloques de atributos.

Nota   Los atributos se pueden recuperar en tiempo de ejecución mediante las API de reflexión de .NET Framework. Estas API no se incluyen en el ámbito de esta especificación.

Por ejemplo, un marco de trabajo podría definir un atributo Help que se pueda colocar en elementos del programa, como clases y métodos, para proporcionar una asignación entre dichos elementos y su documentación, tal como se muestra en el siguiente ejemplo:

<AttributeUsage(AttributeTargets.All)> \_  
Public Class HelpAttribute  
 Inherits Attribute  
  
 Public Sub New(urlValue As String)  
 Me.UrlValue = urlValue  
 End Sub  
  
 Public Topic As String  
 Private UrlValue As String  
  
 Public ReadOnly Property Url() As String  
 Get  
 Return UrlValue  
 End Get  
 End Property  
End Class

El ejemplo define una clase de atributos denominada HelpAttribute, abreviada como Help, que posee un parámetro de posición (UrlValue) y un argumento con nombre (Topic).

En el siguiente ejemplo se muestran varios usos del atributo:

<Help("http://www.example.com/.../Class1.htm")> \_  
Public Class Class1  
 <Help("http://www.example.com/.../Class1.htm", Topic:="F")> \_  
 Public Sub F()  
 End Sub  
End Class

En el siguiente ejemplo se comprueba si Class1 incluye un atributo Help, y se escriben los valores Topic y Url asociados si el atributo está presente.

Module Test  
 Sub Main()  
 Dim type As Type = GetType(Class1)  
 Dim arr() As Object = \_  
 type.GetCustomAttributes(GetType(HelpAttribute), True)  
  
 If arr.Length = 0 Then  
 Console.WriteLine("Class1 has no Help attribute.")  
 Else  
 Dim ha As HelpAttribute = CType(arr(0), HelpAttribute)  
 Console.WriteLine("Url = " & ha.Url & ", Topic = " & ha.Topic)  
 End If  
 End Sub  
End Module

## Clases de atributos

Una clase de atributos es una clase no genérica que deriva de System.Attribute y no se declara como MustInherit. La clase de atributos puede incluir un atributo System.AttributeUsage que declare en qué elementos es válido el atributo, si se puede usar varias veces en una declaración y si se hereda. En el siguiente ejemplo se define una clase de atributos denominada SimpleAttribute que se puede colocar en declaraciones de clase y en declaraciones de interfaz:

<AttributeUsage(AttributeTargets.Class Or AttributeTargets.Interface)> \_  
Public Class SimpleAttribute  
 Inherits System.Attribute  
End Class

En el siguiente ejemplo se muestran algunos usos del atributo Simple. Si bien la clase de atributos se denomina SimpleAttribute, los usos de este atributo pueden omitir el sufijo Attribute y acortar el nombre a Simple:

<Simple> Class Class1  
End Class  
  
<Simple> Interface Interface1  
End Interface

Si el atributo no tiene System.AttributeUsage, se puede colocar en cualquier destino (equivalente a AttributeTargets.All). El atributo System.AttributeUsage incluye un inicializador de variable, AllowMultiple, que especifica si el atributo indicado se puede especificar varias veces en una declaración determinada. Si el inicializador AllowMultiple de un atributo es True, se trata de una clase de atributos de uso múltiple y se puede especificar más de una vez en una declaración. Si el inicializador AllowMultiple de un atributo es False o no se especifica en un atributo, se trata de una clase de atributos de uso único y se puede especificar una vez como máximo en una declaración.

En el ejemplo siguiente se define una clase de atributos de uso múltiple de nombre AuthorAttribute:

<AttributeUsage(AttributeTargets.Class, AllowMultiple:=True)> \_  
Public Class AuthorAttribute  
 Inherits System.Attribute  
  
 Private \_Value As String  
  
 Public Sub New(value As String)  
 Me.\_Value = value  
 End Sub  
  
 Public ReadOnly Property Value() As String  
 Get  
 Return \_Value  
 End Get  
 End Property  
End Class

En el ejemplo se muestra una declaración de clase con dos usos del atributo Author:

<Author("Maria Hammond"), Author("Ramesh Meyyappan")> \_  
Class Class1  
End Class

El atributo System.AttributeUsage incluye una variable de instancia pública, Inherited, que especifica si el atributo, cuando se incluye en un tipo base, también se hereda en los tipos que derivan de este tipo base. Si no se inicializa la variable de instancia pública Inherited, se usa el valor predeterminado False. Las propiedades y los eventos no heredan atributos, aunque sí lo hacen los métodos que se definen en ellos. Las interfaces no heredan atributos.

Si un atributo de uso único se hereda y a la vez se especifica en un tipo derivado, el atributo especificado en el tipo derivado invalida el atributo heredado. Si un atributo de uso múltiple se hereda y a la vez se especifica en un tipo derivado, ambos atributos se especifican en el tipo derivado. Por ejemplo:

<AttributeUsage(AttributeTargets.Class, AllowMultiple:=True, \_  
 Inherited:=True) > \_  
Class MultiUseAttribute  
 Inherits System.Attribute  
  
 Public Sub New(value As Boolean)  
 End Sub  
End Class  
  
<AttributeUsage(AttributeTargets.Class, Inherited:=True)> \_  
Class SingleUseAttribute  
 Inherits Attribute  
  
 Public Sub New(value As Boolean)  
 End Sub  
End Class  
  
<SingleUse(True), MultiUse(True)> Class Base  
End Class  
  
' Derived has three attributes defined on it: SingleUse(False),  
' MultiUse(True) and MultiUse(False)  
<SingleUse(False), MultiUse(False)> \_  
Class Derived  
 Inherits Base  
End Class

Los parámetros posicionales del atributo se definen mediante parámetros de los constructores públicos de la clase de atributos. Los parámetros posicionales se deben declarar como ByVal y no pueden especificar ByRef. Las propiedades y las variables de instancia públicas se definen mediante variables de instancia o propiedades de lectura y escritura públicas de la clase de atributos. Los tipos que se pueden usar en los parámetros posicionales y en las propiedades y variables de instancia públicas se limitan a los tipos de atributo. Un tipo es un tipo de atributo si es:

Cualquier tipo primitivo excepto Date y Decimal.

El tipo Object.

El tipo System.Type.

Un tipo enumerado, con la condición de que este y los tipos en los que esté anidado (si los hubiera) tengan acceso definido como Public.

Una matriz unidimensional de uno de los tipos anteriores de la lista.

## Bloques de atributos

Los atributos se especifican en bloques de atributos. Cada bloque de atributos se delimita con corchetes angulares ("<>"); se pueden especificar varios atributos con una lista separada por comas en uno o varios bloques de atributos. El orden en que se especifican los atributos no resulta significativo. Por ejemplo, los bloques de atributos <A, B>, <B, A>, <A> <B> y <B> <A> son equivalentes.

Los atributos no se pueden especificar en una clase de declaración que no admitan y los atributos de uso único no se pueden especificar varias veces en un bloque de atributos. El ejemplo siguiente produce errores ya que intenta usar HelpString en la interfaz Interface1 e intenta usarlo más de una vez en la declaración de Class1.

<AttributeUsage(AttributeTargets.Class)> \_  
Public Class HelpStringAttribute  
 Inherits System.Attribute  
  
 Private InternalValue As String  
  
 Public Sub New(value As String)  
 Me.InternalValue = value  
 End Sub  
  
 Public ReadOnly Property Value() As String  
 Get  
 Return InternalValue  
 End Get  
 End Property  
End Class  
  
' Error: HelpString only applies to classes.  
<HelpString("Description of Interface1")> \_  
Interface Interface1  
 Sub Sub1()  
End Interface  
  
' Error: HelpString is single-use.  
<HelpString("Description of Class1"), \_  
 HelpString("Another description of Class1")> \_  
Public Class Class1  
End Class

Los atributos constan de un modificador de atributo opcional, un nombre de atributo, una lista opcional de argumentos posicionales e inicializadores de variables y propiedades. Si no hay parámetros ni inicializadores, pueden omitirse los paréntesis. Si el atributo incluye un modificador, debe incluirse en un bloque de atributos en la parte superior del archivo de código fuente.

Si un archivo de código fuente incluye en su parte superior un bloque de atributos que especifica los atributos del ensamblado o módulo que contendrá el archivo de código fuente, cada atributo del bloque deberá incluir los modificadores Assembly o Module y un signo de dos puntos como prefijo.

Attributes ::=  
 AttributeBlock |  
 Attributes AttributeBlock

AttributeBlock ::= [ LineTerminator ] < AttributeList [ LineTerminator ] > [ LineTerminator ]

AttributeList ::=  
 Attribute |  
 AttributeList *Comma* Attribute

Attribute ::=  
 [ AttributeModifier : ] SimpleTypeName [ OpenParenthesis [ AttributeArguments ] CloseParenthesis ]

AttributeModifier ::= Assembly | Module

### Nombres de atributo

El nombre de un atributo especifica una clase de atributos. Por convención, las clases de atributos se denominan con el sufijo Attribute. Los usos de un atributo pueden incluir u omitir este sufijo. Por tanto, el nombre de una clase de atributos que corresponde a un identificador de atributo es el propio identificador o la concatenación del identificador calificado y Attribute. Cuando el compilador resuelve un nombre de atributo, anexa Attribute al nombre y realiza la búsqueda. Si da un error, el compilador realiza la búsqueda sin el sufijo. Por ejemplo, los usos de una clase de atributos SimpleAttribute pueden omitir el sufijo Attribute, de manera que el nombre se acortaría a Simple:

<Simple> Class Class1  
End Class  
  
<Simple> Interface Interface1  
End Interface

El ejemplo anterior es semánticamente equivalente al que se muestra a continuación:

<SimpleAttribute> Class Class1  
End Class  
  
<SimpleAttribute> Interface Interface1  
End Interface

En general, se prefieren los atributos que incluyen el sufijo Attribute en su nombre. En el ejemplo siguiente se muestran dos clases de atributos, T y TAttribute.

<AttributeUsage(AttributeTargets.All)> \_  
Public Class T  
 Inherits System.Attribute  
End Class  
  
<AttributeUsage(AttributeTargets.All)> \_  
Public Class TAttribute  
 Inherits System.Attribute  
End Class  
  
' Refers to TAttribute.  
<T> Class Class1   
End Class  
  
' Refers to TAttribute.  
<TAttribute> Class Class2   
End Class

Tanto el bloque de atributos <T> como el bloque de atributos <TAttribute> refieren a la clase de atributos denominada TAttribute. No se puede usar T como atributo si no se quita la declaración de la clase TAttribute.

### Argumentos de atributos

Los argumentos de un atributo pueden aparecer de dos formas: como argumentos posicionales y como inicializadores de variables o propiedades de instancia. Los argumentos posicionales del atributo deben preceder a los inicializadores de variables o de propiedades de instancia. Los argumentos posicionales se componen de un expresión constante, una expresión de creación de matriz unidimensional o una expresión GetType. Los inicializadores de variables o propiedades de instancia se componen de un identificador, que puede coincidir con palabras clave, seguido de un carácter de dos puntos y un signo igual, y que termina con una expresión constante o una expresión GetType.

Dado un atributo con la clase de atributos T, la lista de argumentos posicionales P y la lista de inicializadores de variables y propiedades de instancia N, estos pasos determinan si los argumentos son válidos:

Siga los pasos de procesamiento en tiempo de compilación para compilar una expresión con la forma New T(P). Esta operación da como resultado un error en tiempo de compilación o determina un constructor en T que es más aplicable a la lista de argumentos.

Si el constructor determinado en el paso 1 incluye parámetros que no son tipos de atributo o resulta inaccesible en el sitio de declaración, se produce un error en tiempo de compilación.

En cada inicializador de variables o propiedades de instancia Arg de N:

Supongamos que Name es el identificador del inicializador de variables o propiedades de instancia Arg.

Name debe identificar una propiedad sin parámetros o una variable de instancia Shared, grabable y Public en T cuyo tipo sea un tipo de atributo. Si T no cuenta con dicha propiedad o variable de instancia, se produce un error en tiempo de compilación.

Por ejemplo:

<AttributeUsage(AttributeTargets.All)> \_  
Public Class GeneralAttribute  
 Inherits Attribute  
  
 Public Sub New(x As Integer)  
 End Sub  
  
 Public Sub New(x As Double)  
 End Sub  
  
 Public y As Type  
  
 Public Property z As Integer  
 Get  
 End Get  
  
 Set  
 End Set  
 End Property  
End Class  
  
' Calls the first constructor.  
<General(10, z:=30, y:=GetType(Integer))> \_  
Class C1  
End Class  
  
' Calls the second constructor.  
<General(10.5, z:=10)> \_  
Class C2  
End Class

Los parámetros de tipo no se puede utilizar en ninguna parte de los argumentos de un atributo. Sin embargo, sí pueden usarse los tipos construidos:

<AttributeUsage(AttributeTargets.All)> \_  
Class A   
 Inherits System.Attribute   
   
 Public Sub New(t As Type)  
 End Sub   
End Class  
  
Class List(Of T)   
 ' Error: attribute argument cannot use type parameter  
 <A(GetType(T))> Dim t1 As T   
  
 ' OK: closed type  
 <A(GetType(List(Of Integer)))> Dim y As Integer  
End Class

AttributeArguments ::=  
 AttributePositionalArgumentList |  
 AttributePositionalArgumentList Comma VariablePropertyInitializerList |  
 VariablePropertyInitializerList

AttributePositionalArgumentList ::=  
 [ AttributeArgumentExpression ] |  
 AttributePositionalArgumentList Comma [ AttributeArgumentExpression ]

VariablePropertyInitializerList ::=  
 VariablePropertyInitializer |  
 VariablePropertyInitializerList Comma VariablePropertyInitializer

VariablePropertyInitializer ::=  
 IdentifierOrKeyword ColonEquals AttributeArgumentExpression

AttributeArgumentExpression ::=  
 ConstantExpression |  
 GetTypeExpression |  
 ArrayExpression

# Archivos de código fuente y espacios de nombres

Los programas de Visual Basic constan de uno o varios archivos de código fuente. Cuando se compila un programa, se procesan todos los archivos de código fuente, por lo que estos pueden depender entre sí, posiblemente en un sentido circular, sin necesidad de ninguna declaración adelantada. El orden textual de las declaraciones en el texto del programa generalmente carece de importancia.

Los archivos de código fuente constan de un conjunto opcional de instrucciones de opción, instrucciones de importación y atributos, a los que sigue un cuerpo de espacio de nombres. Los atributos, cada uno de los cuales debe incluir el modificador Assembly o Module, se aplican al módulo o ensamblado .NET producido por la compilación. El cuerpo de los archivos de código fuente actúa como una declaración de espacio de nombres implícita para el espacio de nombres global, en el sentido de que todas las declaración del nivel superior de un archivo de código fuente se incluyen en el espacio de nombres global. Por ejemplo:

FileA.vb:

Class A  
End Class

FileB.vb:

Class B  
End Class

Los dos archivos de código fuente contribuyen al espacio de nombres global, en este caso, declarando dos clases con los nombres completos A y B. Como los dos archivos de código fuente contribuyen al mismo espacio de declaración, si cada uno contiene una declaración de miembro con el mismo nombre se producirá un error.

Nota El entorno de compilación puede invalidar las declaraciones de espacio de nombres en las que se incluye un archivo de código fuente de forma implícita.

Salvo donde se indique, las instrucciones de un programa de Visual Basic pueden finalizar con un terminador de línea o con un signo de dos puntos.

Start ::=  
 [ OptionStatement+ ]  
 [ ImportsStatement+ ]  
 [ AttributesStatement+ ]  
 [ NamespaceMemberDeclaration+ ]

StatementTerminator ::= LineTerminator | :

AttributesStatement ::= Attributes StatementTerminator

## Inicio y terminación de un programa

El inicio de programa se produce cuando el entorno de ejecución ejecuta un método designado, que se conoce como punto de entrada del programa. Este método de punto de entrada, que siempre debe recibir el nombre Main, debe estar compartido, no puede estar incluido en un tipo genérico, no puede tener el modificador "async" y debe incluir una de las siguientes signaturas:

Sub Main()  
Sub Main(args() As String)  
Function Main() As Integer  
Function Main(args() As String) As Integer

La accesibilidad del método de punto de entrada es irrelevante. Si un programa contiene varios puntos de entrada adecuados, el entorno de compilación debe designar uno de ellos como punto de entrada. En caso contrario, se producirá un error en tiempo de compilación. El entorno de compilación también puede crear un método de punto de entrada si no existe ninguno.

Cuando comienza un programa, si el punto de entrada incluye un parámetro, el argumento que se proporciona en el entorno de ejecución contiene los argumentos de línea de comandos para el programa representados como cadenas. Si el punto de entrada tiene un tipo de valor devuelto Integer, el valor que devuelve la función vuelve al entorno de ejecución como resultado del programa.

En todos los demás aspectos, los métodos de punto de entrada actúan de la misma forma que otros métodos. Cuando la ejecución abandona la invocación del método de punto de entrada realizada por el entorno de ejecución, termina el programa.

## Opciones de compilación

Los archivos de código fuente pueden especificar opciones de compilación en su código mediante instrucciones de opción. Las instrucciones Option solo se aplican al archivo de código fuente donde aparecen y en un archivo de código fuente solo puede aparecer una instrucción Option de cada tipo. Por ejemplo:

Option Strict On  
Option Compare Text  
Option Strict Off ' Not allowed, Option Strict is already specified.  
Option Compare Text ' Not allowed, Option Compare is already specified.

Existen cuatro opciones de compilación: semántica de tipo estricta, semántica de declaración explícita, semántica de comparación y semántica de inferencia de tipo de variable local. Si un archivo de código fuente no incluye una instrucción Option determinada, el entorno de compilación determina el conjunto concreto de semántica que se usará. Existe también una quinta opción de compilación, comprobaciones de desbordamiento con enteros, que solamente se puede especificar mediante el entorno de compilación.

OptionStatement ::=  
 OptionExplicitStatement |  
 OptionStrictStatement |  
 OptionCompareStatement |  
 OptionInferStatement

### Instrucción Option Explicit

La instrucción Option Explicit determina si las variables locales se pueden declarar de forma implícita. Las palabras clave On u Off pueden seguir a la instrucción; si no se especifica ninguna de ellas, el valor predeterminado es On. Si no se especifica ninguna instrucción en un archivo, el entorno de compilación determina cuál se usará.

Nota   Explicit y Off no son palabras reservadas.

Option Explicit Off  
  
Module Test  
 Sub Main()  
 x = 5 ' Valid because Option Explicit is off.  
 End Sub  
End Module

En este ejemplo, la variable local x se declara de forma implícita mediante una asignación a ella. El tipo de x es Object.

OptionExplicitStatement ::= Option Explicit [ OnOff ] StatementTerminator

OnOff ::= On | Off

### Instrucción Option Strict

La instrucción Option Strict determina si las conversiones y las operaciones de Object se rigen por una semántica de tipo estricta o permisiva, y si los tipos se declaran de forma implícita como Object si no se especifica ninguna cláusula As. Las palabras clave On u Off pueden seguir a la instrucción; si no se especifica ninguna de ellas, el valor predeterminado es On. Si no se especifica ninguna instrucción en un archivo, el entorno de compilación determina cuál se usará.

Nota   Strict y Off no son palabras reservadas.

Option Strict On  
  
Module Test  
 Sub Main()  
 Dim x ' Error, no type specified.  
 Dim o As Object  
 Dim b As Byte = o ' Error, narrowing conversion.  
  
 o.F() ' Error, late binding disallowed.  
 o = o + 1 ' Error, addition is not defined on Object.  
 End Sub  
End Module

En la semántica estricta, no se permiten las siguientes acciones:

Conversiones de restricción sin un operador de conversión explícito.

Enlace en tiempo de ejecución.

Operaciones en el tipo Object distintas de TypeOf...Is, Is e IsNot.

Omisión de la cláusula As en una declaración que no incluye un tipo inferido.

OptionStrictStatement ::= Option Strict [ OnOff ] StatementTerminator

### Instrucción Option Compare

La instrucción Option Compare determina la semántica de las comparaciones de cadenas. Las comparaciones de cadenas se llevan a cabo mediante comparaciones binarias (en las que se compara el valor Unicode binario de cada carácter) o comparaciones de texto (en las que se compara el significado léxico de cada carácter partiendo de la referencia cultural actual). Si no se especifica ninguna instrucción en un archivo, el entorno de compilación controla el tipo de comparación que se usará.

Nota   Compare, Binary y Text no son palabras reservadas.

Option Compare Text  
  
Module Test  
 Sub Main()  
 Console.WriteLine("a" = "A") ' Prints True.  
 End Sub  
End Module

En este caso, la comparación de cadenas se realiza mediante una comparación de texto que omite las diferencias entre mayúsculas y minúsculas. Si se hubiese especificado Option Compare Binary, el resultado sería False.

OptionCompareStatement ::= Option Compare CompareOption StatementTerminator

CompareOption ::= Binary | Text

### Comprobaciones de desbordamiento con enteros

Las operaciones con enteros se pueden comprobar o no en tiempo de ejecución para detectar condiciones de desbordamiento. Si se comprueban las condiciones de desbordamiento y se produce un desbordamiento en una operación con enteros, se produce una excepción System.OverflowException. Si no se comprueban las condiciones de desbordamiento, los desbordamientos en una operación con enteros no producen ninguna excepción . El entorno de compilación determina si esta opción está activada o desactivada.

### Instrucción Option Infer

La instrucción Option Infer determina si las declaraciones de variable local que no incluyen la cláusula As cuentan con un tipo inferido o usan Object. Las palabras clave On u Off pueden seguir a la instrucción; si no se especifica ninguna de ellas, el valor predeterminado es On. Si no se especifica ninguna instrucción en un archivo, el entorno de compilación determina cuál se usará.

Nota   Infer y Off no son palabras reservadas.

Option Infer On  
  
Module Test  
 Sub Main()  
 ' The type of x is Integer  
 Dim x = 10  
  
 ' The type of y is String  
 Dim y = "abc"  
 End Sub  
End Module

OptionInferStatement ::= Option Infer [ OnOff ] StatementTerminator

## Instrucción Imports

Las instrucciones Imports importan los nombres de entidades en un archivo de código fuente, lo que permite hacer referencia a estos nombres sin ninguna calificación, o importan un espacio de nombres para usarlo en expresiones XML.

En las declaraciones de miembros de un archivo de código fuente que contiene una instrucción Imports, los tipos incluidos en el espacio de nombres dado pueden referirse directamente, tal como se muestra en el siguiente ejemplo:

Imports N1.N2  
  
Namespace N1.N2  
 Class A  
 End Class  
End Namespace   
  
Namespace N3  
 Class B  
 Inherits A  
 End Class  
End Namespace

En el anterior ejemplo, dentro del archivo de código fuente, los miembros de tipo del espacio de nombres N1.N2 están directamente disponibles y, por tanto, la clase N3.B se deriva de la clase N1.N2.A.

Las instrucciones Imports deben aparecer detrás de las instrucciones Option, pero antes de cualquier declaración de tipos. El entorno de compilación también puede definir instrucciones Imports implícitas.

Las instrucciones Imports hacen que los nombres estén disponibles en un archivo de código fuente, pero no declaran nada en el espacio de declaración del espacio de nombres global. El ámbito de los nombres que se importan en una instrucción Imports se extiende a lo largo de las declaraciones de miembro del espacio de nombres incluidas en el archivo de código fuente. El ámbito de una instrucción Imports no incluye específicamente otras instrucciones Imports ni otros archivos de código fuente. Las instrucciones Imports no pueden hacerse referencia entre ellas.

En este ejemplo, la última instrucción Imports es errónea porque no se ve afectada por el primer alias de importación.

Imports R1 = N1 ' OK.  
Imports R2 = N1.N2 ' OK.  
Imports R3 = R1.N2 ' Error: Can't refer to R1.  
  
Namespace N1.N2  
End Namespace

Nota   Los nombres de espacios de nombres o tipos que aparecen en las instrucciones Imports siempre se tratan como si fuesen completos. Es decir, el identificador del extremo izquierdo de un nombre de espacio de nombres o tipo siempre se resuelve en el espacio de nombres global y el resto de la resolución continúa según las reglas normales de resolución. Este es el único lugar del lenguaje donde se aplica esta regla, que garantiza que un nombre no se puede ocultar por completo de la calificación. Sin la regla, si un nombre del espacio de nombres global se ocultase en un archivo de código fuente determinado, sería imposible especificar ningún nombre de dicho espacio de nombres mediante calificación.

En este ejemplo, la instrucción Imports siempre hace referencia al espacio de nombres System global y no a la clase del archivo de código fuente.

Imports System ' Imports the namespace, not the class.  
  
Class System  
End Class

ImportsStatement ::= Imports ImportsClauses StatementTerminator

ImportsClauses ::=  
 ImportsClause |  
 ImportsClauses Comma ImportsClause

ImportsClause ::=  
 AliasImportsClause |  
 MembersImportsClause |  
 XMLNamespaceImportsClause

### Alias de importación

Los alias de importación definen alias para un espacio de nombres o un tipo.

Imports A = N1.N2.A  
  
Namespace N1.N2  
 Class A  
 End Class  
End Namespace  
  
Namespace N3  
 Class B  
 Inherits A  
 End Class  
End Namespace

En este ejemplo, dentro del archivo de código fuente, 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:

Imports R = N1.N2  
  
Namespace N3  
 Class B  
 Inherits R.A  
 End Class  
End Namespace

El identificador de un alias de importación debe ser único dentro del espacio de declaración del espacio de nombres global (no solo en la declaración del espacio de nombres global del archivo de código fuente donde se define el alias de importación), aunque no declare ningún nombre en el espacio de declaración del espacio de nombres global.

Anotación

Las declaraciones de un módulo no incluyen nombres en el espacio de declaración contenedor. Por lo tanto, una declaración de un módulo puede tener el mismo nombre que un alias de importación, aunque el nombre de la declaración resultará accesible en el espacio de declaración contenedor.

' Error: Alias A conflicts with typename A  
Imports A = N3.A  
  
Class A  
End Class  
  
Namespace N3  
 Class A  
 End Class  
End Namespace

En este ejemplo, el espacio de nombres global ya contiene un miembro A, por lo que un alias de importación no puede usar este identificador. De la misma forma, resultaría un error que dos o más alias de importación del mismo archivo de código fuente declarasen alias con el mismo nombre.

Los alias de importación pueden crear alias para cualquier espacio de nombres o tipo. 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.

Imports R1 = N1  
Imports R2 = N1.N2  
  
Namespace N1.N2  
 Class A  
 End Class  
End Namespace  
  
Namespace N3  
 Class B  
 Private a As N1.N2.A  
 Private b As R1.N2.A  
 Private c As R2.A  
 End Class  
End Namespace

En este ejemplo, 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.

La importación especifica el nombre exacto del espacio de nombres o tipo para los que crea un alias. Este debe ser el nombre completo exacto de dicho espacio de nombres o tipo: no usa las reglas normales para resolución de nombres calificados (que, por ejemplo, permiten el acceso a los miembros de una clase base a través de una clase derivada).

Si un alias de importación apunta a un tipo o espacio de nombres que estas reglas no pueden resolver, la instrucción de importación se omite (y el compilador da una advertencia).

Asimismo, no se puede hacer referencia a un tipo genérico abierto: todos los tipos genéricos deben contar con argumentos de tipo válidos y las reglas anteriores deben poder resolver todos los argumentos de tipo. Cualquier enlace incorrecto de un tipo genérico es un error.

Imports A = G ' error: since G is an open generic type  
Imports B = G(Of Integer) ' okay  
Imports C = Derived.Nested ' warning: Derived.Nested isn't itself a type  
Imports D = G(Of Derived.Nested) ' error: Derived.Nested isn't found  
  
Class G(Of T) : End Class  
  
Class Base  
 Class Nested : End Class  
End Class  
  
Class Derived : Inherits Base  
End Class  
  
Module Module1  
 Sub Main()  
 Dim x As C ' error: "C" wasn’t succesfully defined  
 Dim y As Derived.Nested ' okay  
 End Sub  
End Module

Las declaraciones del archivo de código fuente pueden sombrear el nombre del alias de importación.

Imports R = N1.N2  
  
Namespace N1.N2  
 Class A  
 End Class  
End Namespace  
  
Namespace N3  
 Class R  
 End Class  
  
 Class B  
 Inherits R.A ' Error, R has no member A  
 End Class  
End Namespace

En el ejemplo anterior, la referencia a R.A en la declaración de B produce un error porque R hace referencia a N3.R, no a N1.N2.

Un alias de importación ofrece un alias dentro de un determinado archivo de código fuente, pero no contribuye con ningún miembro nuevo al espacio de declaración subyacente. En otras palabras, los alias de importación no son transitivos sino que afectan únicamente al archivo de código fuente donde aparecen.

File1.vb:

Imports R = N1.N2  
  
Namespace N1.N2  
 Class A  
 End Class  
End Namespace

File2.vb:

Class B  
 Inherits R.A ' Error, R unknown.  
End Class

En el ejemplo anterior, el ámbito del alias de importación que define R solo se extiende a las declaraciones del archivo de código fuente que lo contiene y, por lo tanto, R es desconocido en el segundo archivo de código fuente.

AliasImportsClause ::=   
 Identifier Equals TypeName

### Importaciones de espacios de nombres

La importación del espacio de nombres importa todos los miembros de un espacio de nombres o un tipo; de esta forma, los identificadores de los miembros del espacio de nombres o tipo se pueden usar sin calificación. En el caso de los tipos, la importación del espacio de nombres solo permite el acceso a los miembros compartidos del tipo sin necesidad de calificar el nombre de clase. Concretamente, permite que los miembros de los tipos enumerados se usen sin calificación. Por ejemplo:

Imports Colors  
  
Enum Colors  
 Red  
 Green  
 Blue  
End Enum  
  
Module M1  
 Sub Main()  
 Dim c As Colors = Red  
 End Sub  
End Module

A diferencia de un alias de importación, una importación de espacio de nombres no impone ninguna restricción a los nombres que importa y puede importar espacios de nombres y tipos cuyos identificadores ya estén declarados en el espacio de nombres global. Los nombres que se importan mediante una importación normal se sombrean con los alias de importación y las declaraciones del archivo de código fuente.

En el ejemplo siguiente, A se refiere a N3.A y no a N1.N2.A en las declaraciones de miembro del espacio de nombres N3.

Imports N1.N2  
  
Namespace N1.N2  
 Class A  
 End Class  
End Namespace  
  
Namespace N3  
 Class A  
 End Class  
  
 Class B  
 Inherits A  
 End Class  
End Namespace

Si varios espacios de nombres importados contienen miembros con el mismo nombre (y este nombre no se sombrea con un alias de importación o una declaración), la referencia a dicho nombre resulta ambigua u produce un error en tiempo de compilación.

Imports N1  
Imports N2  
  
Namespace N1  
 Class A  
 End Class  
End Namespace   
  
Namespace N2  
 Class A  
 End Class  
End Namespace   
  
Namespace N3  
 Class B  
 Inherits A ' Error, A is ambiguous.  
 End Class  
End Namespace

En el ejemplo anterior, tanto N1 como N2 contienen un miembro A. N3 importa ambos, por lo que una referencia a A en N3 produce un error en tiempo de compilación. En este caso, el conflicto se puede resolver mediante la calificación de referencias a A, o mediante la introducción de un alias de importación que elija un A determinado, tal como se muestra en el siguiente ejemplo:

Imports N1  
Imports N2  
Imports A = N1.A  
  
Namespace N3  
 Class B  
 Inherits A ' A means N1.A.  
 End Class  
End Namespace

Solamente se pueden importar espacios de nombres, clases, estructuras, tipos enumerados y módulos estándar.

MembersImportsClause ::=  
 TypeName

### Importaciones de espacios de nombres XML

Las importaciones de espacios de nombres XML definen un espacio de nombres o el espacio de nombres predeterminado para las expresiones XML no calificadas incluidas en la unidad de compilación. Por ejemplo:

Imports <xmlns:db="http://example.org/database">  
  
Module Test  
 Sub Main()  
 ' db namespace is "http://example.org/database"  
 Dim x = <db:customer><db:Name>Bob</></>  
  
 Console.WriteLine(x.<db:Name>)  
 End Sub  
End Module

Un espacio de nombres XML, incluido el espacio de nombres predeterminado, solamente se puede definir una vez para un conjunto determinado de importaciones. Por ejemplo:

Imports <xmlns:db="http://example.org/database-one">  
' Error: namespace db is already defined  
Imports <xmlns:db="http://example.org/database-two">

XMLNamespaceImportsClause ::=  
 < XMLNamespaceAttributeName [ XMLWhitespace ] *Equals* [ XMLWhitespace ] XMLNamespaceValue >

XMLNamespaceValue ::=  
 DoubleQuoteCharacter [ XMLAttributeDoubleQuoteValueCharacter+ ] DoubleQuoteCharacter |  
 SingleQuoteCharacter [ XMLAttributeSingleQuoteValueCharacter+ ] SingleQuoteCharacter

## Espacios de nombres

Los programas de Visual Basic se organizan utilizando espacios de nombres. Los espacios de nombres organizan internamente un programa y organizan la manera en que los elementos del programa se exponen a otros programas.

A diferencia de otras entidades, los espacios de nombres son de extremo abierto y se pueden declarar varias veces en el mismo programa y entre varios programas; cada declaración incluye sus miembros en el mismo espacio de nombres. En el siguiente ejemplo, las dos declaraciones de espacios de nombres contribuyen al mismo espacio de declaración, declarando dos clases con los nombres completos N1.N2.A y N1.N2.B.

Namespace N1.N2  
 Class A  
 End Class  
End Namespace  
  
Namespace N1.N2  
 Class B  
 End Class  
End Namespace

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.

Existe un espacio de nombres global que no tiene nombre y a cuyos espacios de nombres y tipos anidados siempre se puede tener acceso sin calificación. El ámbito de un miembro de espacio de nombres declarado en el espacio de nombres global es el texto de todo el programa. En caso contrario, el ámbito de un tipo o un espacio de nombres declarado en un espacio de nombres cuyo nombre completo es N, es el texto del programa de cada espacio de nombres cuyo nombre completo de espacio de nombres correspondiente comience con N o sea el propio N. (Un compilador puede decidir incluir las declaraciones en un espacio de nombres concreto de forma predeterminada. Esto no afecta al hecho de que existe un espacio de nombres global sin nombre.)

En este ejemplo, la clase B puede ver a la clase A porque el espacio de nombres N1.N2.N3 de B está anidado conceptualmente en el espacio de nombres N1.N2.

Namespace N1.N2  
 Class A  
 End Class  
End Namespace  
  
Namespace N1.N2.N3  
 Class B  
 Inherits A  
 End Class  
End Namespace

### Declaraciones de espacio de nombres

Existen tres formas de expresiones de declaración de espacio de nombres.

La primera forma empieza por la palabra clave Namespace seguida de un nombre de espacio de nombres relativo. Si el nombre del espacio de nombres relativo está calificado, la declaración de espacio de nombres se considera léxicamente anidada dentro de las declaraciones de espacio de nombres correspondientes a cada nombre del nombre calificado. Por ejemplo, los dos espacios de nombres siguientes son semánticamente equivalentes:

Namespace N1.N2  
 Class A  
 End Class  
  
 Class B  
 End Class  
End Namespace   
  
Namespace N1  
 Namespace N2  
 Class A  
 End Class  
  
 Class B  
 End Class  
 End Namespace  
End Namespace

La segunda forma empieza por las palabras clave Namespace Global. Se trata como si todas las declaraciones de miembros se situaran léxicamente en el espacio de nombres global sin nombre, independientemente de cualquier valor predeterminado proporcionado por el entorno de compilación. Esta forma de declaración de espacio de nombres no se puede anidar léxicamente dentro de cualquier otra declaración de espacio de nombres.

La tercera forma empieza por las palabras clave Namespace Global seguidas de un identificador calificado N. Se trata como si fuera una declaración de espacio de nombres de la primera forma, "Namespace N", situada léxicamente en el espacio de nombres global sin nombre, independientemente de cualquier valor predeterminado proporcionado por el entorno de compilación. Esta forma de declaración de espacio de nombres no se puede anidar léxicamente dentro de cualquier otra declaración de espacio de nombres.

Namespace Global ' Puts N1.A in the global namespace  
 Namespace N1  
 Class A  
 End Class  
 End Namespace  
End Namespace  
  
Namespace Global.N1 ' Equivalent to the above  
 Class A  
 End Class  
End Namespace  
  
Namespace N1 ' May or may not be equivalent to the above,  
 Class A ' depending on defaults provided by the  
 End Class ' compilation environment  
End Namespace

Cuando se trabaja con los miembros de un espacio de nombres, no importa dónde se declara un miembro determinado. Si dos programas definen una entidad con el mismo nombre en el mismo espacio de nombres y se intenta resolver el nombre en el espacio de nombres, se produce un error de ambigüedad.

Los espacios de nombres se declaran como Public por definición y la declaración de un espacio de nombres no puede incluir modificadores de acceso.

NamespaceDeclaration ::=  
 Namespace NamespaceName StatementTerminator  
 [ NamespaceMemberDeclaration+ ]  
 End Namespace StatementTerminator

NamespaceName ::=   
 RelativeNamespaceName |  
 Global |  
 Global . RelativeNamespaceName

*RelativeNamespaceName ::=*  
 Identifier |  
 *Relative*NamespaceName Period IdentifierOrKeyword

### Miembros de espacio de nombres

Los miembros de un espacio de nombres solo pueden ser las declaraciones de espacio de nombres y las declaraciones de tipo. Las declaraciones de tipos pueden tener acceso Public o Friend. El acceso predeterminado de los tipos es Friend.

NamespaceMemberDeclaration ::=  
 NamespaceDeclaration |  
 TypeDeclaration

TypeDeclaration ::=  
 ModuleDeclaration |  
 NonModuleDeclaration

NonModuleDeclaration ::=  
 EnumDeclaration |  
 StructureDeclaration |  
 InterfaceDeclaration |  
 ClassDeclaration |  
 DelegateDeclaration

# Tipos

Las dos categorías fundamentales de tipos de Visual Basic son los tipos de valor y los tipos de referencia. Las estructuras, las enumeraciones y los tipos primitivos (excepto las cadenas) son tipos de valor. Las clases, las cadenas y los módulos estándar, las interfaces, las matrices y los delegados son tipos de referencia.

Cada tipo tiene un valor predeterminado, que es el valor que se asigna a las variables de dicho tipo durante la inicialización.

TypeName ::=  
 ArrayTypeName |  
 NonArrayTypeName

NonArrayTypeName ::=  
 SimpleTypeName |  
 NullableTypeName

SimpleTypeName ::=  
 QualifiedTypeName |  
 BuiltInTypeName

QualifiedTypeName ::=  
 Identifier [ TypeArguments ] |  
 Global Period IdentifierOrKeyword [ TypeArguments ] |  
 QualifiedTypeName Period IdentifierOrKeyword [ TypeArguments ]

TypeArguments ::=  
 OpenParenthesis Of TypeArgumentList CloseParenthesis

TypeArgumentList ::=  
 TypeName |  
 TypeArgumentList Comma TypeName

BuiltInTypeName ::= Object | PrimitiveTypeName

TypeModifier ::= AccessModifier | Shadows

IdentifierModifiers ::= [ *NullableNameModifier* ] [ ArrayNameModifier ]

## Tipos de valor y tipos de referencia

Aunque los tipos de valor y los tipos de referencia pueden ser similares en cuanto al uso y la sintaxis de declaración, su semántica es diferente.

Los tipos de referencia se guardan en el montón en tiempo de ejecución y solo se tiene acceso a ellos mediante una referencia a ese almacén. Debido a que a los tipos de referencia siempre se tiene acceso a través de referencias, su duración la administra .NET Framework. Se lleva a cabo el seguimiento de las referencias pendientes a una instancia concreta y la instancia se destruye cuando no quedan más referencias. Una variable de tipo de referencia contiene una referencia a un valor de ese tipo, un valor de un tipo más derivado o un valor nulo. Un valor nulo (null) no hace referencia a nada y no es posible hacer nada con él salvo asignarlo. La asignación a una variable de un tipo de referencia crea una copia de la referencia en lugar de una copia del valor al que se hace referencia. Para una variable de un tipo de referencia, el valor predeterminado es null.

Los tipos de valor se guardan directamente en la pila, bien dentro de una matriz bien dentro de otro tipo; a su almacenamiento solo se tiene acceso de manera directa. Como los tipos de valor se guardan directamente en las variables, su duración está determinada por la duración de la variable que los contiene. Cuando la ubicación que contiene una instancia de un tipo de valor se destruye, la instancia también se destruye. El acceso a los tipos de valor siempre es directo; no es posible crear una referencia a un tipo de valor. Prohibiendo estas referencias se impide hacer referencia a la instancia de una clase de valor que ha sido destruida. Dado que los tipos de valor siempre son NotInheritable, una variable de un tipo de valor siempre contiene un valor de dicho tipo. Por esta razón, el valor de un tipo de valor no puede ser null ni puede hacer referencia a un objeto de un tipo más derivado. La asignación a una variable de un tipo de valor crea una copia del valor que se asigna. Para una variable de un tipo de valor, el valor predeterminado es el resultado de inicializar cada miembro de variable del tipo en su valor predeterminado.

En el siguiente ejemplo se muestra la diferencia entre los tipos de referencia y los tipos de valor:

Class Class1  
 Public Value As Integer = 0  
End Class  
  
Module Test  
 Sub Main()  
 Dim val1 As Integer = 0  
 Dim val2 As Integer = val1  
 val2 = 123  
 Dim ref1 As Class1 = New Class1()  
 Dim ref2 As Class1 = ref1  
 ref2.Value = 123  
 Console.WriteLine("Values: " & val1 & ", " & val2)  
 Console.WriteLine("Refs: " & ref1.Value & ", " & ref2.Value)  
 End Sub  
End Module

El resultado del programa es:

Values: 0, 123  
Refs: 123, 123

La asignación de la variable local val2 no tiene efecto en la variable local val1 puesto que ambas son un tipo de valor (el tipo Integer) y cada variable local de un tipo de valor tiene su almacenamiento propio. Por el contrario, la asignación ref2.Value = 123; afecta al objeto al que tanto ref1 como ref2 hacen referencia.

Un aspecto que se debe destacar sobre el sistema de tipos de .NET Framework es que aunque las estructuras, enumeraciones y tipos primitivos (excepto String) son tipos de valor, todos heredan de tipos de referencia. Las estructuras y los tipos primitivos heredan del tipo de referencia System.ValueType, que, a su vez, hereda de Object. Los tipos enumerados heredan del tipo de referencia System.Enum, el cual hereda del tipo System.ValueType.

### Tipos de valor que aceptan valores NULL

Con los tipos de valor, es posible agregar un modificador ? a un nombre de tipo para representar la versión que acepta valores NULL de ese tipo. Un tipo de valor que acepta el valor null puede contener los mismos valores que la versión que no los acepta, además del valor null. De este modo, con un tipo de valor que acepta valores null, asignar Nothing a una variable del tipo establece el valor de la variable en el valor null, no en el valor cero del tipo de valor. Por ejemplo:

Dim x As Integer = Nothing  
Dim y As Integer? = Nothing  
  
' Prints zero  
Console.WriteLine(x)  
' Prints nothing (because the value of y is the null value)  
Console.WriteLine(y)

También puede declararse una variable para que sea de un tipo de valor null añadiendo un modificador de tipo que acepte valores null al nombre de la variable. Por motivos de claridad, no es válido tener un modificador de tipo que acepte valores null en el nombre de la variable y en el nombre del tipo en la misma declaración. Ya que los tipos que aceptan valores null se implementan usando el tipo System.Nullable(Of T), el tipo T? es sinónimo del tipo System.Nullable(Of T) y ambos nombres se pueden emplear indistintamente. No se puede colocar el modificador ? en un tipo que ya acepta valores null; por consiguiente, no es posible declarar el tipo Integer?? ni System.Nullable(Of Integer)?.

Un tipo de valor que acepta valores null T? tiene los miembros de System.Nullable(Of T) además de los operadores o conversiones elevados del tipo subyacente T al tipo T?. La elevación copia los operadores y las conversiones del tipo subyacente, en la mayoría de los casos para sustituir tipos de valor que aceptan valores null por tipos de valor que no los aceptan. Esto permite que muchas de las operaciones y conversiones que se aplican a T se apliquen también a T?.

NullableTypeName ::= NonArrayTypeName ?

NullableNameModifier ::= ?

## Implementación de interfaces

Las declaraciones de clases y estructuras pueden declarar que se implementan como un conjunto de tipos de interfaz a través de una o de varias cláusulas Implements. Todos los tipos especificados en la cláusula Implements deben ser interfaces y el tipo debe implementar todos los miembros especificados de las interfaces. Por ejemplo:

Interface ICloneable  
 Function Clone() As Object  
End Interface   
  
Interface IComparable  
 Function CompareTo(other As Object) As Integer  
End Interface   
  
Structure ListEntry  
 Implements ICloneable, IComparable  
  
 ...  
  
 Public Function Clone() As Object Implements ICloneable.Clone  
 ...  
 End Function   
  
 Public Function CompareTo(other As Object) As Integer \_  
 Implements IComparable.CompareTo  
 ...  
 End Function   
End Structure

Un tipo que implementa una interfaz también implementa implícitamente todas las interfaces base de dicha interfaz. Esto es cierto aunque el tipo no muestre explícitamente todas las interfaces base en la cláusula Implements. En este ejemplo, la estructura TextBox implementa tanto IControl como ITextBox.

Interface IControl  
 Sub Paint()  
End Interface   
  
Interface ITextBox  
 Inherits IControl  
  
 Sub SetText(text As String)  
End Interface   
  
Structure TextBox  
 Implements ITextBox  
  
 ...  
  
 Public Sub Paint() Implements ITextBox.Paint  
 ...  
 End Sub  
  
 Public Sub SetText(text As String) Implements ITextBox.SetText  
 ...  
 End Sub  
End Structure

Declarar que un tipo implementa una interfaz en y de sí mismo no declara nada en el espacio de declaración del tipo. De este modo, es válido implementar dos interfaces que tienen un método con el mismo nombre.

Los tipos no pueden implementar un parámetro de tipo por sí solos, aunque se implique a los parámetros de tipo que están en el ámbito.

Class C1(Of V)  
 Implements V ' Error, can't implement type parameter directly  
 Implements IEnumerable(Of V) ' OK, not directly implementing  
  
 ...  
End Class

Las interfaces genéricas se pueden implementar muchas veces mediante argumentos de tipo distintos. Sin embargo, un tipo genérico no puede implementar una interfaz genérica utilizando un parámetro de tipo si el parámetro proporcionado (con independencia de las restricciones de tipo) pudiera superponerse a otra implementación de esa interfaz. Por ejemplo:

Interface I1(Of T)  
End Interface  
  
Class C1  
 Implements I1(Of Integer)  
 Implements I1(Of Double) ' OK, no overlap  
End Class  
  
Class C2(Of T)  
 Implements I1(Of Integer)  
 Implements I1(Of T) ' Error, T could be Integer  
End Class

TypeImplementsClause ::= Implements *Type*Implements StatementTerminator

TypeImplements ::=  
 NonArrayTypeName |  
 *Type*Implements Comma NonArrayTypeName

## Tipos primitivos

Los tipos primitivos se identifican mediante palabras clave, las cuales son alias de tipos predefinidos en el espacio de nombres System. Un tipo primitivo es totalmente indiferenciable del tipo del que es alias: escribir la palabra reservada Byte es exactamente igual que escribir System.Byte. Los tipos primitivos se conocen también como *tipos intrínsecos*.

Como un tipo primitivo es el alias de un tipo regular, todos los tipos primitivos tienen miembros. Por ejemplo, Integer tiene los miembros declarados en System.Int32. Los literales pueden tratarse como instancias de sus tipos correspondientes.

Los tipos primitivos se diferencian de otros tipos de estructura en que permiten determinadas operaciones adicionales:

La mayoría de los tipos primitivos permiten la creación de valores mediante la escritura de literales. Por ejemplo, 123I es un literal del tipo Integer.

Es posible declarar constantes de tipos primitivos.

Cuando todos los operandos de una expresión son constantes de tipos primitivos, el compilador puede evaluar la expresión en tiempo de compilación. Dicha expresión se conoce por el nombre de expresión constante.

En Visual Basic se definen los siguientes tipos primitivos:

Los tipos de valor entero Byte (entero de 1 byte sin signo), SByte (entero de 1 byte con signo), UShort (entero de 2 bytes sin signo), Short (entero de 2 bytes con signo), UInteger (entero de 4 bytes sin signo), Integer (entero de 4 bytes con signo), ULong (entero de 8 bytes sin signo) y Long (entero de 8 bytes con signo). Estos tipos se asignan a System.Byte, System.SByte, System.UInt16, System.Int16, System.UInt32, System.Int32, System.UInt64 y System.Int64, respectivamente. El valor predeterminado de un tipo entero es equivalente al literal 0.

Los tipos de valor de punto flotante Single (punto flotante de 4 bytes) y Double (punto flotante de 8 bytes). Estos tipos se asignan a System.Single y System.Double, respectivamente. El valor predeterminado de un punto flotante es equivalente al literal 0.

El tipo Decimal (valor decimal de 16 bytes), que se asigna a System.Decimal. El valor predeterminado de un decimal es equivalente al literal 0D.

El tipo de valor Boolean, que representa un valor de veracidad, suele ser resultado de una relación u operación lógica. El literal es del tipo System.Boolean. El valor predeterminado de un tipo Boolean es equivalente al literal False.

El tipo de valor Date, que representa una fecha o una hora y se asigna a System.DateTime. El valor predeterminado de un tipo Date es equivalente al literal # 01/01/0001 12:00:00AM #.

El tipo de valor Char, que representa un único carácter Unicode y se asigna a System.Char. El valor predeterminado de un tipo Char es equivalente a la expresión constante ChrW(0).

El tipo de referencia String, que representa una secuencia de caracteres Unicode y se asigna a System.String. El valor predeterminado de un tipo String es un valor null.

PrimitiveTypeName ::= NumericTypeName | Boolean | Date | Char | String

NumericTypeName ::= IntegralTypeName | FloatingPointTypeName | Decimal

IntegralTypeName ::= Byte | SByte | UShort | Short | UInteger | Integer | ULong | Long

FloatingPointTypeName ::= Single | Double

## Enumeraciones

Las enumeraciones son tipos de valor que heredan de System.Enum y representan simbólicamente un conjunto de valores de uno de los tipos enteros primitivos. El valor predeterminado de un tipo de enumeración E es el valor que produce la expresión CType(0, E).

El tipo subyacente de una enumeración debe ser un tipo entero que pueda representar todos los valores de enumerador definidos en la enumeración. Si se especifica un tipo subyacente, debe ser Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long o uno de sus tipos correspondientes del espacio de nombres System. Si no se especifica ningún tipo subyacente de forma explícita, el valor predeterminado será Integer.

En el ejemplo siguiente se declara una enumeración con un tipo subyacente Long.

Enum Color As Long  
 Red  
 Green  
 Blue  
End Enum

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 Integer, o para preservar esta opción para el futuro.

EnumDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Enum Identifier [ As NonArrayTypeName ] StatementTerminator  
 EnumMemberDeclaration+  
 End Enum StatementTerminator

### Miembros de enumeraciones

Los miembros de una enumeración son los valores enumerados declarados en la enumeración y los miembros heredados de la clase System.Enum.

El ámbito de un miembro de una enumeración es el cuerpo de la declaración de la enumeración. Esto quiere decir que, fuera de una declaración de enumeración, un miembro de enumeración siempre se debe calificar, salvo que el tipo se importe específicamente en un espacio de nombres a través de una importación de espacio de nombres.

El orden de declaración en las declaraciones de miembros de enumeración es importante cuando se omiten los valores de expresión constante. Todos los miembros de la enumeración tienen implícitamente acceso Public y no se puede utilizar ningún modificador de acceso en las declaraciones de miembros de enumeración.

EnumMemberDeclaration ::= [ Attributes ] Identifier [ Equals ConstantExpression ] StatementTerminator

### Valores de enumeración

Los valores de enumeración de una lista de miembros de enumeración se declaran como constantes con el tipo de la enumeración subyacente y pueden aparecer donde se requieran constantes. Una definición de miembro de enumeración con = da al miembro asociado el valor indicado en la expresión constante. La expresión constante debe evaluarse como un tipo entero convertible implícitamente en el tipo subyacente y debe estar en el intervalo de valores que puede representar el tipo subyacente. En el ejemplo siguiente se produce un error porque los valores constantes 1.5, 2.3 y 3.3 no son implícitamente convertibles en el tipo entero subyacente Long con semántica estricta.

Option Strict On  
  
Enum Color As Long  
 Red = 1.5  
 Green = 2.3  
 Blue = 3.3  
End Enum

Varios miembros de enumeración pueden compartir el mismo valor asociado, como se muestra seguidamente:

Enum Color  
 Red  
 Green  
 Blue  
 Max = Blue  
End Enum

En el ejemplo se muestra una enumeración con dos miembros de enumeración (Blue y Max) que tienen el mismo valor asociado.

Si la definición de valor del primer enumerador de la enumeración no tiene inicializador, el valor de la constante correspondiente es 0. Una definición de valor de enumeración sin inicializador da al enumerador el valor obtenido de aumentar el valor del valor de enumeración previo en 1. Este valor aumentado debe estar incluido en el intervalo de valores que puede representar el tipo subyacente.

Enum Color  
 Red  
 Green = 10  
 Blue  
End Enum   
  
Module Test  
 Sub Main()  
 Console.WriteLine(StringFromColor(Color.Red))  
 Console.WriteLine(StringFromColor(Color.Green))  
 Console.WriteLine(StringFromColor(Color.Blue))  
 End Sub  
  
 Function StringFromColor(c As Color) As String  
 Select Case c  
 Case Color.Red  
 Return String.Format("Red = " & CInt(c))  
  
 Case Color.Green  
 Return String.Format("Green = " & CInt(c))  
  
 Case Color.Blue  
 Return String.Format("Blue = " & CInt(c))  
  
 Case Else  
 Return "Invalid color"  
 End Select  
 End Function  
End Module

En el ejemplo anterior se imprimen los valores de enumeración y sus valores asociados. El resultado es:

Red = 0  
Green = 10  
Blue = 11

Las razones de los valores son las siguientes:

Al valor de enumeración Red se le asigna automáticamente el valor 0 (puesto que no tiene inicializador y es el primer miembro de valor de la enumeración).

Al valor de enumeración Green se le asigna explícitamente el valor 10.

Al valor de enumeración Blue se le asigna automáticamente el valor más uno del valor de enumeración que le precede en el código.

La expresión constante no puede usar directa ni indirectamente el valor de su propio valor de enumeración asociado, es decir, no se permite la circularidad en la expresión constante. El ejemplo siguiente no es válido porque las declaraciones de A y B son circulares.

Enum Circular  
 A = B  
 B  
End Enum

A depende de B explícitamente y B depende de A implícitamente.

## Clases

Una clase es una estructura de datos que puede contener miembros de datos (constantes, variables y eventos), miembros de función (métodos, propiedades, indizadores, operadores y constructores) y tipos anidados. Las clases son tipos de referencia. En el ejemplo siguiente se muestra una clase que contiene cada tipo de miembro.

Class AClass  
 Public Sub New()  
 Console.WriteLine("Constructor")  
 End Sub  
  
 Public Sub New(value As Integer)  
 MyVariable = value  
 Console.WriteLine("Constructor")  
 End Sub  
  
 Public Const MyConst As Integer = 12  
 Public MyVariable As Integer = 34  
  
 Public Sub MyMethod()  
 Console.WriteLine("MyClass.MyMethod")  
 End Sub  
  
 Public Property MyProperty() As Integer  
 Get  
 Return MyVariable  
 End Get  
  
 Set (value As Integer)  
 MyVariable = value  
 End Set  
 End Property  
  
 Default Public Property Item(index As Integer) As Integer  
 Get  
 Return 0  
 End Get  
  
 Set (value As Integer)  
 Console.WriteLine("Item(" & index & ") = " & value)  
 End Set  
 End Property  
  
 Public Event MyEvent()  
  
 Friend Class MyNestedClass  
 End Class   
End Class

En el siguiente ejemplo se muestran usos de estos miembros.

Module Test  
  
 ' Event usage.  
 Dim WithEvents aInstance As AClass  
  
 Sub Main()  
 ' Constructor usage.  
 Dim a As AClass = New AClass()  
 Dim b As AClass = New AClass(123)  
  
 ' Constant usage.  
 Console.WriteLine("MyConst = " & AClass.MyConst)  
  
 ' Variable usage.  
 a.MyVariable += 1  
 Console.WriteLine("a.MyVariable = " & a.MyVariable)  
  
 ' Method usage.  
 a.MyMethod()  
  
 ' Property usage.  
 a.MyProperty += 1  
 Console.WriteLine("a.MyProperty = " & a.MyProperty)  
 a(1) = 1  
  
 ' Event usage.  
 aInstance = a  
 End Sub   
  
 Sub MyHandler() Handles aInstance.MyEvent  
 Console.WriteLine("Test.MyHandler")  
 End Sub   
End Module

Existen dos modificadores específicos de clase, MustInherit y NotInheritable. No es válido especificar ambos.

ClassDeclaration ::=  
 [ Attributes ] [ ClassModifier+ ] Class Identifier [ TypeParameterList ] StatementTerminator  
 [ ClassBase ]  
 [ TypeImplementsClause+ ]  
 [ ClassMemberDeclaration+ ]  
 End Class StatementTerminator

ClassModifier ::= TypeModifier | MustInherit | NotInheritable | Partial

### Especificación de clase base

Una declaración de clase puede incluir una especificación de tipo base que define el tipo base directo de la clase. Si una declaración de clase no tiene tipo base explícito, el tipo base directo es implícitamente Object. Por ejemplo:

Class Base  
End Class  
  
Class Derived  
 Inherits Base  
End Class

Los tipos base no pueden ser parámetros de tipo por sí solos, aunque se implique a los parámetros de tipo que están en el ámbito.

Class C1(Of V)   
End Class  
  
Class C2(Of V)  
 Inherits V ' Error, type parameter used as base class   
End Class  
  
Class C3(Of V)  
 Inherits C1(Of V) ' OK: not directly inheriting from V.  
End Class

Las clases solo pueden derivar de Object y clases. No es válido que una clase derive de System.ValueType, System.Enum, System.Array, System.MulticastDelegate o System.Delegate. Una clase genérica no puede derivar de System.Attribute ni de una clase que se derive de ella.

Cada clase tiene exactamente una clase base directa y en la derivación está prohibida la circularidad. No es posible derivar de una clase NotInheritable y el dominio de accesibilidad de la clase base debe ser el mismo o un superconjunto del dominio de accesibilidad de la propia clase.

ClassBase ::= Inherits NonArrayTypeName StatementTerminator

### Miembros de clase

Los miembros de una clase se componen de los miembros que introducen las declaraciones de miembros de clase y los miembros heredados de su clase base directa.

Una declaración de miembro de clase puede tener un acceso Public, Protected, Friend, Protected Friend o Private. Cuando una declaración de miembro de clase no incluye un modificador de acceso, el valor predeterminado de acceso es Public, salvo que sea una declaración de variable, en cuyo caso el valor predeterminado de acceso es Private.

El ámbito de un miembro de clase es el cuerpo de clase en que se produce la declaración de miembro, además de la lista de restricciones de dicha clase (si es genérica y tiene restricciones). Si el miembro tiene acceso Friend, su ámbito se amplía al cuerpo de clase de cualquier clase derivada del mismo programa o cualquier ensamblado al que se le haya dado el acceso Friend, y si el miembro tiene acceso Public, Protected o Protected Friend, su ámbito se extiende al cuerpo de clase de cualquier clase derivada en cualquier programa.

ClassMemberDeclaration ::=  
 NonModuleDeclaration |  
 EventMemberDeclaration |  
 VariableMemberDeclaration |  
 ConstantMemberDeclaration |  
 MethodMemberDeclaration |  
 PropertyMemberDeclaration |  
 ConstructorMemberDeclaration |  
 OperatorDeclaration

## Estructuras

Las estructuras son tipos de valor que heredan de System.ValueType. Se parecen a las clases en que representan estructuras de datos que pueden contener miembros de datos y miembros de función. No obstante, a diferencia de las clases, las estructuras no requieren asignación del montón.

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 las estructuras, cada variable tiene su propia copia de los datos que no son Shared, de manera que no posible que las operaciones de una afecten a la otra, como se ilustra en el siguiente ejemplo:

Structure Point  
 Public x, y As Integer  
  
 Public Sub New(x As Integer, y As Integer)  
 Me.x = x  
 Me.y = y  
 End Sub  
End Structure

Dada la declaración anterior, el resultado del código siguiente es el valor 10:

Module Test  
 Sub Main()  
 Dim a As New Point(10, 10)  
 Dim b As Point = a  
  
 a.x = 100  
 Console.WriteLine(b.x)  
 End Sub  
End Module

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.

StructureDeclaration ::=  
 [ Attributes ] [ StructureModifier+ ] Structure Identifier [ TypeParameterList ]  
 StatementTerminator  
 [ TypeImplementsClause+ ]  
 [ StructMemberDeclaration+ ]  
 End Structure StatementTerminator

StructureModifier ::= TypeModifier | Partial

### Miembros de estructura

Los miembros de una estructura se componen de los miembros introducidos por sus declaraciones de miembros de estructura y los miembros heredados de System.ValueType. Cada estructura tiene implícitamente un constructor de instancias sin parámetros Public que produce el valor predeterminado de la estructura. Como resultado, no es posible que una declaración de tipos de estructura declare un constructor de instancias sin parámetros. Un tipo de estructura, no obstante, puede declarar constructores de instancia parametrizados, como en el ejemplo siguiente:

Structure Point  
 Private x, y As Integer  
  
 Public Sub New(x As Integer, y As Integer)  
 Me.x = x  
 Me.y = y  
 End Sub  
End Structure

Dada la declaración anterior, las instrucciones siguientes crean un Point con x e y inicializados en cero.

Dim p1 As Point = New Point()  
Dim p2 As Point = New Point(0, 0)

Como las estructuras contienen directamente sus valores de campo (en lugar de referencias a esos valores), no pueden contener campos que hagan referencia a sí mismos directa o indirectamente. Por ejemplo, el código siguiente no es válido:

Structure S1  
 Dim f1 As S2  
End Structure  
  
Structure S2  
 ' This would require S1 to contain itself.  
 Dim f1 As S1  
End Structure

Por lo general, una declaración de miembros de estructura solo tiene acceso Public, Friend o Private, pero cuando se invalidan miembros heredados de Object, también podrá usarse acceso Protected y Protected Friend. Cuando una declaración de miembro de estructura no incluye modificador de acceso, el valor predeterminado es acceso Public. El ámbito de un miembro declarado por una estructura es el cuerpo de la estructura en el que tiene lugar la declaración, además de las restricciones de dicha estructura (si fuera genérico y tuviera restricciones).

StructMemberDeclaration ::=  
 NonModuleDeclaration |  
 VariableMemberDeclaration |  
 ConstantMemberDeclaration |  
 EventMemberDeclaration |  
 MethodMemberDeclaration |  
 PropertyMemberDeclaration |  
 ConstructorMemberDeclaration |  
 OperatorDeclaration

## Módulos estándar

Un módulo estándar es un tipo cuyos miembros son Shared implícitamente y su ámbito es el espacio de declaración del espacio de nombres que contiene el módulo estándar, en lugar de solo la declaración del módulo estándar. Es posible que nunca se creen instancias de los módulos estándar. Es un error declarar una variable de un tipo de módulo estándar.

Un miembro de un módulo estándar tiene dos nombres completos, uno sin el nombre del módulo estándar y otro con el nombre del módulo estándar. Más de un módulo estándar de un espacio de nombres puede definir un miembro con un nombre concreto; las referencias no calificadas al nombre fuera de los módulos son ambiguas. Por ejemplo:

Namespace N1  
 Module M1  
 Sub S1()  
 End Sub  
  
 Sub S2()  
 End Sub  
 End Module  
  
 Module M2  
 Sub S2()  
 End Sub  
 End Module  
  
 Module M3  
 Sub Main()  
 S1() ' Valid: Calls N1.M1.S1.  
 N1.S1() ' Valid: Calls N1.M1.S1.  
 S2() ' Not valid: ambiguous.  
 N1.S2() ' Not valid: ambiguous.  
 N1.M2.S2() ' Valid: Calls N1.M2.S2.  
 End Sub  
 End Module  
End Namespace

Un módulo solo puede declararse en un espacio de nombres y no puede estar anidado en otro tipo. Los módulos estándar no pueden implementar interfaces, derivan de Object implícitamente y solo tienen constructores Shared.

ModuleDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Module Identifier StatementTerminator  
 [ ModuleMemberDeclaration+ ]  
 End Module StatementTerminator

### Miembros de módulo estándar

Los miembros de un módulo estándar son los miembros introducidos por sus declaraciones de miembro y los miembros heredados de Object. Los módulos estándar pueden tener cualquier tipo de miembro salvo constructores de instancias. Todos los miembros de tipo de módulo estándar son implícitamente Shared.

Por lo general, una declaración de miembro de módulo estándar solo puede tener acceso Public, Friend o Private, pero cuando se invalidan miembros heredados de Object, también se podrán especificar los modificadores de acceso Protected y Protected Friend. Cuando una declaración de miembro de módulo estándar no incluye un modificador de acceso, el valor predeterminado de acceso es Public, salvo que se trate de una variable, en cuyo caso el valor predeterminado de acceso es Private.

Como se ha mencionado antes, el ámbito de un miembro de módulo estándar es la declaración que contiene la declaración del módulo estándar. Los miembros que heredan de Object no se incluyen en este ámbito especial; esos miembros no tienen ámbito y siempre deben calificar con el nombre del módulo. Si el miembro tiene acceso Friend, su ámbito se extiende solo a los miembros del espacio de nombres declarados en el mismo programa o ensamblados a los que se les ha dado acceso Friend.

ModuleMemberDeclaration ::=  
 NonModuleDeclaration |  
 VariableMemberDeclaration |  
 ConstantMemberDeclaration |  
 EventMemberDeclaration |  
 MethodMemberDeclaration |  
 PropertyMemberDeclaration |  
 ConstructorMemberDeclaration

## Interfaces

Las interfaces son tipos de referencia que otros tipos implementan para garantizar que admiten determinados métodos. Una interfaz nunca se crea directamente y no tiene representación real: otros tipos deben convertirse en un tipo de interfaz. Una interfaz define un contrato. Una clase o estructura que implementa una interfaz debe adherirse a su contrato.

En el ejemplo siguiente se muestra una interfaz que contiene una propiedad predeterminada Item, un evento E, un método F y una propiedad P:

Interface IExample  
 Default Property Item(index As Integer) As String  
  
 Event E()  
  
 Sub F(value As Integer)  
  
 Property P() As String  
End Interface

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

Interface IControl  
 Sub Paint()  
End Interface   
  
Interface ITextBox  
 Inherits IControl  
  
 Sub SetText(text As String)  
End Interface   
  
Interface IListBox  
 Inherits IControl  
  
 Sub SetItems(items() As String)  
End Interface   
  
Interface IComboBox  
 Inherits ITextBox, IListBox   
End Interface

Las clases y las estructuras no pueden implementar varias interfaces. En el ejemplo siguiente, la clase EditBox deriva de la clase Control e implementa tanto IControl como IDataBound:

Interface IDataBound  
 Sub Bind(b As Binder)  
End Interface   
  
Public Class EditBox  
 Inherits Control  
 Implements IControl, IDataBound  
  
 Public Sub Paint() Implements IControl.Paint  
 ...  
 End Sub  
  
 Public Sub Bind(b As Binder) Implements IDataBound.Bind  
 ...  
 End Sub  
End Class

InterfaceDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Interface Identifier [ TypeParameterList ] StatementTerminator  
 [ InterfaceBase+ ]  
 [ InterfaceMemberDeclaration+ ]  
 End Interface StatementTerminator

### Herencia de interfaces

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. Si una declaración de interfaz no tiene base de interfaz explícita, entonces no hay interfaz base para el tipo: las interfaces no heredan de Object (aunque tienen una conversión widening a Object). En el ejemplo siguiente, las interfaces base de IComboBox son IControl, ITextBox e IListBox.

Interface IControl  
 Sub Paint()  
End Interface   
  
Interface ITextBox  
 Inherits IControl  
  
 Sub SetText(text As String)  
End Interface   
  
Interface IListBox  
 Inherits IControl  
  
 Sub SetItems(items() As String)  
End Interface   
  
Interface IComboBox  
 Inherits ITextBox, IListBox   
End Interface

Una interfaz hereda todos los miembros de sus interfaces base. Es decir, la interfaz IComboBox anterior hereda los miembros SetText y SetItems además de Paint.

Una clase o una estructura que implementa una interfaz también implementa implícitamente todas las interfaces base de dicha interfaz.

Si una interfaz aparece más de una vez en el cierre transitivo de las interfaces base, solo contribuye con sus miembros a la interfaz derivada una vez. Un tipo que implementa la interfaz derivada únicamente tiene que implementar una vez los métodos de la interfaz definida varias veces. En el ejemplo siguiente, Paint solo se tiene que implementar una vez, aunque la clase implemente IComboBox e IControl.

Class ComboBox  
 Implements IControl, IComboBox  
  
 Sub SetText(text As String) Implements IComboBox.SetText  
 End Sub  
  
 Sub SetItems(items() As String) Implements IComboBox.SetItems  
 End Sub  
  
 Sub Print() Implements IComboBox.Paint  
 End Sub  
End Class

Una cláusula Inherits no tiene efecto en otras cláusulas Inherits. En el ejemplo siguiente, IDerived debe calificar el nombre de INested con IBase.

Interface IBase  
 Interface INested  
 Sub Nested()  
 End Interface  
  
 Sub Base()  
End Interface  
  
Interface IDerived  
 Inherits IBase, INested ' Error: Must specify IBase.INested.  
End Interface

El dominio de accesibilidad de una interfaz base debe ser el mismo o un superconjunto del dominio de accesibilidad de la propia interfaz.

InterfaceBase ::= Inherits InterfaceBases StatementTerminator

InterfaceBases ::=  
 NonArrayTypeName |  
 InterfaceBases Comma NonArrayTypeName

### Miembros de interfaz

Los miembros de una interfaz se componen de los miembros que presentan las declaraciones de miembros y los miembros heredados de sus interfaces base. Aunque las interfaces no heredan miembros de Object, porque todas las clases o estructuras que implementan una interfaz heredan de Object, los miembros de Object, incluidos los métodos de extensión, se consideran miembros de una interfaz y se les puede llamar directamente en una interfaz sin necesidad de conversión en Object. Por ejemplo:

Interface I1  
End Interface  
  
Class C1  
 Implements I1  
End Class  
  
Module Test  
 Sub Main()  
 Dim i As I1 = New C1()  
 Dim h As Integer = i.GetHashCode()  
 End Sub  
End Module

Members of an interface with the same name as members of Object implicitly shadow Object members. Only nested types, methods, properties, and events may be members of an interface. Methods and properties may not have a body. Interface members are implicitly Public and may not specify an access modifier. The scope of a member declared in an interface is the interface body in which the declaration occurs, plus the constraint list of that interface (if it is generic and has constraints).InterfaceMemberDeclaration ::=  
 NonModuleDeclaration |  
 InterfaceEventMemberDeclaration |  
 InterfaceMethodMemberDeclaration |  
 InterfacePropertyMemberDeclaration

## Matrices

Una matriz es un tipo de referencia que contiene variables a las que se tiene acceso a través de los índices correspondientes de un modelo de uno a uno con el orden de las variables de la matriz. Todas las variables contenidas en una matriz, también conocidas como elementos de la matriz, deben ser del mismo tipo, y este tipo se denomina tipo de elemento de la matriz. Los elementos de una matriz comienzan a existir cuando se crea una instancia de la matriz y dejan de hacerlo cuando se destruye la instancia de la matriz. Cada elemento de una matriz se inicializa en el valor predeterminado de su tipo. El tipo System.Array es el tipo base de todos los tipos de matrices y no se pueden crear instancias de él. Cada tipo de matriz hereda los miembros que declara el tipo System.Array y se puede convertir en él (y Object). Un tipo de matriz unidimensional con elemento T también implementa las interfaces System.Collections.Generic.IList(Of T) e IReadOnlyList(Of T); si T es un tipo de referencia, entonces el tipo de matriz también implementa IList(Of U) e IReadOnlyList(Of U) para cualquier U que tenga una conversión de referencia widening de T.

Una matriz tiene un rango que determina el número de índices asociados a cada elemento de la matriz. El rango de una matriz determina el número de dimensiones de la matriz. Por ejemplo, una matriz con rango de uno se denomina matriz unidimensional y una matriz con rango mayor que uno se denomina matriz multidimensional.

En el ejemplo siguiente se crea una matriz unidimensional de valores enteros, se inicializan los elementos de la matriz y después se imprime cada uno:

Module Test  
 Sub Main()  
 Dim arr(5) As Integer  
 Dim i As Integer  
  
 For i = 0 To arr.Length - 1  
 arr(i) = i \* i  
 Next i  
  
 For i = 0 To arr.Length - 1  
 Console.WriteLine("arr(" & i & ") = " & arr(i))  
 Next i  
 End Sub  
End Module

El programa da el resultado siguiente:

arr(0) = 0  
arr(1) = 1  
arr(2) = 4  
arr(3) = 9  
arr(4) = 16  
arr(5) = 25

Cada dimensión de una matriz tiene una longitud asociada. Las longitudes de las dimensiones 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 cero a N – 1. Si una dimensión tiene longitud cero, no hay índices válidos para ella. 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 longitud cero, se dice que la matriz está vacía. El tipo de elemento de una matriz puede ser cualquiera.

Los tipos de matriz se especifican agregando un modificador a un nombre de tipo existente. El modificador consiste en un paréntesis de apertura, un conjunto con ninguna o alguna coma y un paréntesis de cierre. El tipo modificado es el tipo de elemento de la matriz y el número de dimensiones es el número de comas más uno. Si se especifica más de un modificador, el tipo de elemento de la matriz es una matriz. Los modificadores se leen de izquierda a derecha y el modificador del extremo izquierdo es la matriz más externa. En el ejemplo

Module Test  
 Dim arr As Integer(,)(,,)()  
End Module

el tipo de elemento de arr es una matriz bidimensional de matrices tridimensionales de matrices unidimensionales de Integer.

También puede declararse una variable para que sea de un tipo de matriz poniendo un modificador de tipo de matriz o un modificador de inicialización de tamaño de matriz en el nombre de la variable. En ese caso, el tipo de elemento de matriz es el tipo dado en la declaración y las dimensiones de la matriz las determina el modificador del nombre de variable. Por motivos de claridad, no es válido tener un modificador de tipo de matriz en el nombre de variable y en el nombre de tipo en la misma declaración.

En el ejemplo siguiente se muestran diversas declaraciones de variables locales que usan tipos de matriz con Integer como tipo de elemento:

Module Test  
 Sub Main()  
 Dim a1() As Integer ' Declares 1-dimensional array of integers.  
 Dim a2(,) As Integer ' Declares 2-dimensional array of integers.  
 Dim a3(,,) As Integer ' Declares 3-dimensional array of integers.  
  
 Dim a4 As Integer() ' Declares 1-dimensional array of integers.  
 Dim a5 As Integer(,) ' Declares 2-dimensional array of integers.  
 Dim a6 As Integer(,,) ' Declares 3-dimensional array of integers.  
  
 ' Declare 1-dimensional array of 2-dimensional arrays of integers   
 Dim a7()(,) As Integer  
 ' Declare 2-dimensional array of 1-dimensional arrays of integers.  
 Dim a8(,)() As Integer  
  
 Dim a9() As Integer() ' Not allowed.  
 End Sub  
End Module

Un modificador de nombre de tipo de matriz se extiende a todos los grupos de paréntesis que le siguen. Esto quiere decir que en los casos en los que se permite un conjunto de argumentos incluidos entre paréntesis después de un nombre de tipo, no es posible especificar los argumentos de un nombre de tipo de matriz. Por ejemplo:

Module Test  
 Sub Main()  
 ' This calls the Integer constructor.  
 Dim x As New Integer(3)  
  
 ' This declares a variable of Integer().  
 Dim y As Integer()  
  
 ' This gives an error.  
 ' Array sizes can not be specified in a type name.  
 Dim z As Integer()(3)  
 End Sub  
End Module

En el último caso, (3) se interpreta como parte del nombre de tipo, en lugar de como un conjunto de argumentos de constructor.

ArrayTypeName ::= NonArrayTypeName ArrayTypeModifiers

ArrayTypeModifiers ::= ArrayTypeModifier+

ArrayTypeModifier ::= OpenParenthesis [ RankList ] CloseParenthesis

RankList ::=  
 Comma |  
 RankList Comma

ArrayNameModifier ::=  
 ArrayTypeModifiers |  
 ArraySizeInitializationModifier

## Delegados

Un delegado es un tipo de referencia que hace referencia a un método Shared de un tipo o a un método de instancia de un objeto. El equivalente más cercano de un delegado en otros idiomas es un puntero a una función pero, mientras que este solo puede hacer referencia a funciones Shared, un delegado puede hacer referencia tanto a métodos Shared como a métodos de instancia. En este caso, el delegado no solo almacena una referencia al punto de entrada del método, también almacena una referencia a la instancia del objeto con que se invoca el método.

La declaración de delegados no puede tener una cláusula Handles, una cláusula Implements, un cuerpo de método ni una construcción End. La lista de parámetros de la declaración de delegados no puede tener parámetros Optional ni ParamArray. El dominio de accesibilidad del tipo devuelto y los tipos de parámetro debe ser el mismo o un superconjunto del dominio de accesibilidad del propio delegado.

Los miembros de un delegado son los miembros heredados de la clase System.Delegate. Un delegado también define los métodos siguientes:

Un constructor que toma dos parámetros, uno de tipo Object y otro de tipo System.IntPtr.

Un método Invoke que tiene la misma signatura que el delegado.

Un método BeginInvoke cuya signatura es la signatura del delegado, con tres diferencias. Primera, el tipo devuelto se cambia por System.IAsyncResult. Segunda, tiene dos parámetros adicionales, anexados al final de la lista de parámetros: el primero de tipo System.AsyncCallback y el segundo de tipo Object. Y, por último, todos los parámetros ByRef se cambian por ByVal.

Un método EndInvoke cuyo tipo devuelto es el mismo que el delegado. Los parámetros del método son solo los parámetros del delegado que son exactamente parámetros ByRef, en el mismo orden en que aparecen en la signatura del delegado. Además de estos parámetros, hay otro parámetro de tipo System.IAsyncResult al final de la lista de parámetros.

Son tres los pasos necesarios para definir y usar delegados: declaración, creación de instancias e invocación.

Los delegados se declaran mediante la sintaxis de declaración de delegados. En el ejemplo siguiente se declara un delegado denominado SimpleDelegate que no toma argumentos:

Delegate Sub SimpleDelegate()

En el ejemplo siguiente se crea una instancia de SimpleDelegate, a la que llama inmediatamente:

Module Test  
 Sub F()  
 System.Console.WriteLine("Test.F")  
 End Sub   
  
 Sub Main()  
 Dim d As SimpleDelegate = AddressOf F  
 d()  
 End Sub   
End Module

No tiene mucho sentido crear una instancia de un delegado para un método y llamarle inmediatamente a través del delegado, es más sencillo llamar al método directamente. Los delegados demuestran su utilidad cuando se utiliza su anonimato. En el ejemplo siguiente se muestra un método MultiCall que llama repetidamente a una instancia de SimpleDelegate:

Sub MultiCall(d As SimpleDelegate, count As Integer)  
 Dim i As Integer  
  
 For i = 0 To count - 1  
 d()  
 Next i  
End Sub

No es relevante para el método MultiCall cuál sea el método de destino de SimpleDelegate, qué accesibilidad tenga este método, o si es o no es Shared. Lo único que importa es que la signatura del método de destino sea compatible con SimpleDelegate.

DelegateDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Delegate MethodSignature StatementTerminator

MethodSignature ::= SubSignature | FunctionSignature

## Tipos parciales

Las declaraciones de clases y estructuras pueden ser declaraciones parciales. Una declaración parcial puede describir de forma completa o no el tipo declarado dentro de la declaración. En su lugar, la declaración del tipo puede extenderse por muchas declaraciones parciales; los tipos parciales no pueden declararse fuera de los límites de los programas. Una declaración de tipos parcial especifica el modificador Partial en la declaración. Después, cualquier otra declaración en el programa de un tipo con el mismo nombre completo se combinará con la declaración parcial en tiempo de compilación para formar una única declaración de tipos. Por ejemplo, en el código siguiente se declara una única clase Test con los miembros Test.C1 y Test.C2.

a.vb:

Public Partial Class Test  
 Public Sub S1()  
 End Sub  
End Class

b.vb:

Public Class Test  
 Public Sub S2()  
 End Sub  
End Class

Cuando se combinan declaraciones de tipos parciales, al menos una de ellas debe tener un modificador Partial, de lo contrario se producirá un error en tiempo de compilación.

Anotación

Aunque sea posible especificar Partial en una única declaración entre muchas declaraciones parciales, es mejor especificarla en todas. En el caso en que una declaración parcial sea visible pero una o varias declaraciones parciales estén ocultas (como cuando se extiende código generado por herramientas), es aceptable dejar el modificador Partial fuera de la declaración visible pero especificarlo en las declaraciones ocultas.

Solo las clases y las estructuras se pueden declarar mediante declaraciones parciales. La aridad de un tipo se tiene en cuenta cuando se emparejan declaraciones parciales: dos clases con el mismo nombre pero distinto número de parámetros de tipo no se consideran declaraciones parciales del mismo tipo. Las declaraciones parciales pueden especificar atributos, modificadores de clases, instrucciones Inherits o instrucciones Implements. En tiempo de compilación, todas las partes de las declaraciones parciales se combinan y se usan como parte de la declaración de tipos. Si hay conflictos entre atributos, modificadores, bases, interfaces o miembros de tipo, se produce un error en tiempo de compilación. Por ejemplo:

Public Partial Class Test1  
 Implements IDisposable  
End Class  
  
Class Test1  
 Inherits Object  
 Implements IComparable  
End Class  
  
Public Partial Class Test2  
End Class  
  
Private Partial Class Test2  
End Class

En el ejemplo anterior se declara un tipo Test1 que es Public, hereda de Object e implementa System.IDisposable y System.IComparable. Las declaraciones parciales de Test2 producirán un error en tiempo de compilación porque una dice que Test2 es Public y la otra dice que Test2 es Private.

Los tipos parciales con parámetros de tipo pueden declarar restricciones y varianza para los parámetros de tipo, pero las restricciones y la varianza de cada declaración parcial deben coincidir. De este modo, las restricciones y la varianza son especiales en el sentido de que no se combinan automáticamente con otros modificadores:

Partial Public Class List(Of T As IEnumerable)  
End Class  
  
' Error: Constraints on T don't match  
Class List(Of T As IComparable)  
End Class

El hecho de que un tipo se declare mediante múltiples declaraciones parciales no afecta a las reglas de búsqueda de nombres dentro del tipo. Como resultado, una declaración de tipos parcial puede usar miembros declarados en otras declaraciones de tipos parciales o puede implementar métodos en interfaces declaradas en otras declaraciones de tipos parciales. Por ejemplo:

Public Partial Class Test1  
 Implements IDisposable  
  
 Private IsDisposed As Boolean = False  
End Class  
  
Class Test1  
 Private Sub Dispose() Implements IDisposable.Dispose  
 If Not IsDisposed Then  
 ...  
 End If  
 End Sub  
End Class

Los tipos anidados también pueden tener declaraciones parciales. Por ejemplo:

Public Partial Class Test  
 Public Partial Class NestedTest  
 Public Sub S1()  
 End Sub  
 End Class  
End Class  
  
Public Partial Class Test  
 Public Partial Class NestedTest  
 Public Sub S2()  
 End Sub  
 End Class  
End Class

Los inicializadores de una declaración parcial se ejecutarán en orden de declaración; sin embargo, no hay ningún orden garantizado de ejecución para los inicializadores que tienen lugar en declaraciones parciales independientes.

## Tipos construidos

Una declaración de tipos genérico, por sí mismo, no denota un tipo. Sin embargo, una declaración de tipos genérico se puede usar como “diseño” para formar muchos tipos diferentes aplicando los argumentos de tipo. Un tipo genérico que tiene argumentos de tipo específico aplicados se denomina tipo construido. Los argumentos de tipo de un tipo construido deben satisfacer siempre las restricciones inherentes a los parámetros de tipo con los que coinciden.

Un nombre de tipo puede identificar un tipo construido aunque 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 contenedora se utiliza de manera implícita para la búsqueda de nombres:

Class Outer(Of T)   
 Public Class Inner   
 End Class  
  
 ' Type of i is the constructed type Outer(Of T).Inner  
 Public i As Inner   
End Class

Un tipo construido C(Of T1,…,Tn) es accesible cuando se puede obtener acceso a todos los tipos genéricos y a todos los argumentos de tipo. Por ejemplo, si el tipo genérico C es Public y todos los argumentos de tipo T1,…,Tn son Public, entonces el tipo construido es Public. Si el nombre del tipo o uno de los argumentos de tipo es Private, entonces la accesibilidad del tipo construido es Private. Si un argumento de tipo del tipo construido es Protected y otro es Friend, entonces el tipo construido es accesible solo en la clase y sus subclases de este ensamblado o de cualquier ensamblado al que se haya dado acceso Friend. Dicho de otro modo, el dominio de accesibilidad de un tipo construido es la intersección de los dominios de accesibilidad de sus partes constitutivas.

Anotación

El hecho de que el dominio de accesibilidad del tipo construido sea la intersección de sus partes constitutivas tiene el curioso efecto secundario de definir un nuevo nivel de accesibilidad. Solo se puede tener acceso a un tipo construido que contiene un elemento Protected y un elemento Friend en contextos que tienen acceso a ambos miembros, Friend y Protected. Sin embargo, no hay forma de expresar este nivel de accesibilidad en el lenguaje, ya que la accesibilidad Protected Friend significa que se puede tener acceso a una entidad en un contexto al que tengan acceso tanto los miembros Friend como Protected.

La base, las interfaces y los miembros implementados de tipos construidos se determinan sustituyendo los argumentos de tipo ofrecidos en cada aparición del parámetro de tipo en el tipo genérico.

### Tipos cerrados y abiertos

Un tipo construido para el que uno o más argumentos de tipo son parámetros de un método o tipo contenedor se denomina tipo abierto. Esto es así porque algunos parámetros de tipo del tipo aún no se conocen, así que la forma real del tipo no es del todo conocida. Por el contrario, un tipo genérico cuyos argumentos sean parámetros sin tipo se denomina tipo cerrado. La forma de un tipo cerrado se conoce siempre. Por ejemplo:

Class Base(Of T, V)  
End Class  
  
Class Derived(Of V)  
 Inherits Base(Of Integer, V)  
End Class  
  
Class MoreDerived  
 Inherits Derived(Of Double)  
End Class

El tipo construido Base(Of Integer, V) es un tipo abierto porque aunque se ha suministrado el parámetro de tipo T, el parámetro de tipo U ha recibido otro parámetro de tipo. Por consiguiente, la forma completa del tipo es aún desconocida. El tipo construido Derived(Of Double), sin embargo, es un tipo cerrado porque se han proporcionado todos los parámetros de tipo de la jerarquía de herencia.

Los tipos abiertos se definen como sigue:

Un parámetro de tipo es un tipo abierto.

Un tipo de matriz es un tipo abierto si su tipo de elemento es un tipo abierto.

Un tipo construido es un tipo abierto si uno o más de sus argumentos de tipo es un tipo abierto.

Un tipo cerrado es un tipo que no es abierto.

Como el punto de entrada del programa no puede estar en un tipo genérico, todos los tipos usados en tiempo de ejecución serán tipos cerrados.

## Tipos especiales

.NET Framework contiene una serie de clases que reciben un trato especial en .NET Framework y Visual Basic:

Al tipo System.Void, que representa un tipo vacío en .NET Framework, solo se le puede hacer referencia en expresiones GetType.

Los tipos System.RuntimeArgumentHandle, System.ArgIterator y System.TypedReference pueden contener punteros a la pila, de forma que no pueden aparecer en el montón de .NET Framework. Por consiguiente, no se pueden usar como tipos de elemento de matriz, tipos devueltos, tipos de campo, argumentos de tipo genérico, tipos que admiten valores null, tipos de parámetros ByRef, el tipo de un valor que se convierte en Object o System.ValueType, el destino de una llamada a los miembros de una instancia de Object o System.ValueType, o elevado a un cierre.

# Conversiones

La conversión es el proceso de cambiar un valor de un tipo en otro. Por ejemplo, un valor de tipo Integer se puede convertir en un valor de tipo Double, o un valor de tipo Derived puede convertirse en un valor de tipo Base, suponiendo que Base y Derived sean clases y Derived herede de Base. Las conversiones tal vez no requieran el valor que cambia (como en el último ejemplo), o tal vez requieran cambios significativos en la representación del valor (como en el primer ejemplo).

Las conversiones pueden ser widening o narrowing. Una conversión widening es una conversión de un tipo en otro cuyo dominio de valor es por lo menos tan grande, o mayor, que el dominio de valor del tipo original. Las conversiones widening no deberían dar nunca error. Una conversión narrowing es una conversión de un tipo en otro cuyo dominio de valor es tan pequeño como el dominio de valor del tipo o lo suficientemente poco relacionado como para que haya que tener un cuidado especial al hacer la conversión (por ejemplo, cuando se convierta de Integer en String). Las conversiones narrowing, que pueden conllevar perdida de información, pueden dar error.

La conversión de identidad (es decir, una conversión de un tipo en sí mismo) y la conversión de valor predeterminado (es decir, una conversión de Nothing) se definen para todos los tipos.

## Conversiones implícitas y explícitas

Las conversiones pueden ser implícitas o explícitas. Las conversiones implícitas tienen lugar sin ninguna sintaxis especial. A continuación se ofrece un ejemplo de una conversión implícita de un valor Integer en un valor Long:

Module Test  
 Sub Main()  
 Dim intValue As Integer = 123  
 Dim longValue As Long = intValue  
  
 Console.WriteLine(intValue & " = " & longValue)  
 End Sub  
End Module

Las conversiones explícitas, por otra parte, requieren operadores de conversión. Si se intenta efectuar una conversión explícita en un valor sin operador de conversión de tipos se produce un error en tiempo de compilación. En el siguiente ejemplo de código se utiliza una conversión explícita para convertir un valor Long en un valor Integer.

Module Test  
 Sub Main()  
 Dim longValue As Long = 134  
 Dim intValue As Integer = CInt(longValue)  
  
 Console.WriteLine(longValue & " = " & intValue)  
 End Sub  
End Module

El conjunto de conversiones implícitas depende del entorno de compilación y la instrucción Option Strict. Si se usa semántica estricta, solo las conversiones widening se pueden producir de forma implícita. Si se usa semántica permisiva, todas las conversiones widening y narrowing (es decir, todas las conversiones) pueden producirse de forma implícita.

## Conversiones booleanas

Aunque Boolean no es un tipo numérico, tiene conversiones narrowing de y en tipos numéricos como si fuera un tipo enumerado. El literal True convierte en el literal 255 para Byte, 65535 para UShort, 4294967295 para UInteger, 18446744073709551615 para ULong y en la expresión -1 para SByte, Short, Integer, Long, Decimal, Single y Double. El literal False convierte en el literal 0. Un valor numérico cero convierte en el literal False. El resto de los valores numéricos se convierten en el literal True.

Hay una conversión narrowing de Boolean en String, que convierte en System.Boolean.TrueString o System.Boolean.FalseString. También hay una conversión narrowing de String en Boolean: si string es igual a TrueString o FalseString (en la referencia cultural actual, sin distinción de mayúsculas y minúsculas) usa el valor apropiado; en caso contrario intenta analizar la cadena en un tipo numérico (en hex u octal si es posible, si no, como float) y usa las reglas anteriores; de lo contrario produce una System.InvalidCastException.

## Conversiones numéricas

Las conversiones numéricas existen entre los tipos Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long, Decimal, Single y Double y todos los tipos enumerados. Cuando se convierten, los tipos enumerados se tratan como si fueran sus tipos subyacentes. Cuando se convierte en un tipo enumerado, no es necesario el valor de origen para cumplir con el conjunto de valores definidos en el tipo enumerado. Por ejemplo:

Enum Values  
 One  
 Two  
 Three  
End Enum  
  
Module Test  
 Sub Main()  
 Dim x As Integer = 5  
  
 ' OK, even though there is no enumerated value for 5.  
 Dim y As Values = CType(x, Values)  
 End Sub  
End Module

Las conversiones numéricas se procesan en tiempo de ejecución del modo siguiente:

En una conversión de un tipo numérico en un tipo numérico más amplio, el valor simplemente se convierte en el tipo más amplio. Las conversiones de UInteger, Integer, ULong, Long o Decimal en Single o Double se redondean hasta el valor Single o Double más próximo. Aunque esta conversión puede causar una pérdida de precisión, nunca causará una pérdida de magnitud.

En una conversión de un tipo entero en otro tipo entero o de Single, Double o Decimal en un tipo entero, el resultado depende de si está activada la comprobación de desbordamiento de enteros:

Si se comprueba el desbordamiento de enteros:

Si el origen es un tipo entero, la conversión se efectuará si el argumento de origen se encuentra dentro del intervalo del tipo de destino. La conversión producirá una excepción System.OverflowException si el argumento de origen está fuera del intervalo del tipo de destino.

Si el origen es Single, Double o Decimal, el valor de origen se redondea hacia arriba o hacia abajo hasta el valor entero más cercano y este valor entero pasa a ser el resultado de la conversión. Si el valor de origen está a la misma distancia de dos valores enteros, se redondea al valor que tiene un número impar en la posición de dígito menos significativo. Si el valor entero resultante queda fuera del intervalo del tipo de destino, se produce una excepción System.OverflowException.

Si no se comprueba el desbordamiento de enteros:

Si el origen es un tipo entero, la conversión siempre es correcta y simplemente consiste en descartar los bits más significativos del valor de origen.

Si el origen es Single, Double o Decimal, la conversión siempre se efectúa y simplemente consiste en redondear el valor de origen hasta el valor entero más cercano. Si el valor de origen está a la misma distancia de dos valores enteros, siempre se redondea al valor que tiene un número impar en la posición de dígito menos significativo.

Para una conversión de Double en Single, el valor Double se redondea al valor Single más próximo. Si el valor Double es demasiado pequeño para representarlo como Single, el resultado se convierte en cero positivo o cero negativo. Si el valor Double es demasiado grande para representarlo como Single, 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 Single o Double en Decimal, el valor de origen se convierte en la representación Decimal y se redondea hasta el número más próximo después de la posición decimal 28 si es necesario. 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 Double en Single, el valor Double se redondea al valor Single más próximo. Si el valor Double es demasiado pequeño para representarlo como Single, el resultado se convierte en cero positivo o cero negativo. Si el valor Double es demasiado grande para representarlo como Single, el resultado se convierte en infinito positivo o infinito negativo. Si el valor Double es NaN, el resultado también es NaN.

## Conversiones de referencias

Los tipos de referencia se pueden convertir en un tipo base y viceversa. Las conversiones de un tipo base en un tipo más derivado solo se efectúan en tiempo de ejecución si el valor que se convierte es un valor null, el propio tipo derivado o un tipo más derivado.

Los tipos de clase y de interfaz se pueden convertir en cualquier tipo de interfaz. Las conversiones entre un tipo y un tipo de interfaz solo se efectúan en tiempo de compilación si los tipos reales implicados tienen una relación de herencia o de implementación. Como un tipo de interfaz siempre contendrá una instancia de un tipo que deriva de Object, un tipo de interfaz también se puede convertir en y de Object.

Anotación

No es un error convertir clases NotInheritable en interfaces que no implementa porque las clases que representan clases COM pueden tener implementaciones de interfaces que no se conocen hasta el tiempo de ejecución.

Si una conversión de referencia da error en tiempo de ejecución, se produce una excepción System.InvalidCastException.

### Conversiones de varianza de referencia

Las interfaces o los delegados genéricos pueden tener parámetros de tipo variante que permiten las conversiones entre variantes del tipo compatibles. Por tanto, en tiempo de ejecución se efectuará una conversión de un tipo de clase o de un tipo de interfaz en un tipo de interfaz compatible con la variante con un tipo de interfaz de la que hereda o que implementa. De igual modo, los tipos de delegado se pueden convertir en tipos de delegado compatible con la variante. Por ejemplo, el tipo de delegado:

Delegate Function F(Of In A, Out R)(a As A) As R

permitiría una conversión de F(Of Object, Integer) en F(Of String, Integer). Es decir, un delegado F que toma Object puede usarse con seguridad como un delegado F que toma String. Cuando se llama al delegado, el método de destino esperará un objeto, y una cadena es un objeto.

Se dice que un tipo de interfaz o delegado genérico S(Of S1,…,SN) es compatible con variante con un tipo de delegado o interfaz genérico T(Of T1,…,TN) si:

S y T se construyen a partir del mismo tipo genérico U(Of U1,…,UN).

Para cada parámetro de tipo UX:

Si el parámetro de tipo se declaró sin varianza, entonces SX y TX deben ser el mismo tipo.

Si el parámetro de tipo se declaró In entonces tiene que haber una identidad de ampliación, valor predeterminado, referencia, matriz o conversión de parámetro de tipo de SX a TX.

Si el parámetro de tipo se declaró Out entonces tiene que haber una identidad de ampliación, valor predeterminado, referencia, matriz o conversión de parámetro de tipo de TX a SX.

Cuando se convierte de una clase en una interfaz genérica con parámetros de tipo de variante, si la clase implementa más de una interfaz compatible con variante, la conversión es ambigua si no hay una conversión no variante. Por ejemplo:

Class Base  
End Class  
  
Class Derived1  
 Inherits Base  
End Class  
  
Class Derived2  
 Inherits Base  
End Class  
  
Class OneAndTwo  
 Implements IEnumerable(Of Derived1)  
 Implements IEnumerable(Of Derived2)  
End Class  
  
Class BaseAndOneAndTwo  
 Implements IEnumerable(Of Base)  
 Implements IEnumerable(Of Derived1)  
 Implements IEnumerable(Of Derived2)  
End Class  
  
Module Test  
 Sub Main()  
 ' Error: conversion is ambiguous  
 Dim x As IEnumerable(Of Base) = New OneAndTwo()  
  
 ' OK, will pick up the direct implementation of IEnumerable(Of Base)  
 Dim y as IEnumerable(Of Base) = New BaseAndOneAndTwo()  
 End Sub  
End Module

### Conversiones de delegados anónimos

Cuando una expresión clasificada como método lambda se reclasifica como valor en un contexto donde no hay tipo de destino (por ejemplo, Dim x = Function(a As Integer, b As Integer) a + b), o donde el tipo de destino no es un tipo delegado, el tipo de la expresión resultante es un tipo de delegado anónimo equivalente a la signatura del método lambda. Este tipo de delegado anónimo tiene una conversión en cualquier tipo de delegado compatible: un tipo de delegado compatible es cualquier tipo de delegado que puede crearse usando una expresión de creación de delegados con el método Invoke del tipo de delegado anónimo como parámetro. Por ejemplo:

' Anonymous delegate type similar to Func(Of Object, Object, Object)  
Dim x = Function(x, y) x + y  
  
' OK because delegate type is compatible  
Dim y As Func(Of Integer, Integer, Integer) = x

Observe que los tipos System.Delegate y System.MulticastDelegate no se consideran en sí tipos delegados (aunque todos los tipos delegados hereden de ellos). Asimismo, observe que la conversión de un tipo de delegado anónimo en un tipo de delegado compatible no es una conversión de referencia.

## Conversiones de matrices

Además de las conversiones que se definen en matrices por el hecho de ser tipos de referencia, existen varias conversiones especiales para las matrices.

Para dos tipos A y B cualesquiera, si hay tipos de referencia o parámetros de tipo que no se sabe si son tipos de valor, y si A tiene una referencia, matriz, o conversión de parámetro de tipo en B, existe una conversión de una matriz de tipo A en una matriz de tipo B con el mismo rango. Esta relación se conoce como covarianza matricial. La covarianza matricial significa que un elemento de una matriz cuyo tipo de elemento es B puede ser en realidad un elemento de una matriz cuyo tipo de elemento es A, siempre que tanto A como B sean tipos de referencia y que B tenga una conversión de referencia o conversión de matriz en A. En el ejemplo siguiente, la segunda invocación de F produce una excepción System.ArrayTypeMismatchException, puesto que el tipo de elemento real de b es String y no Object:

Module Test  
 Sub F(ByRef x As Object)  
 End Sub  
  
 Sub Main()  
 Dim a(10) As Object  
 Dim b() As Object = New String(10) {}  
 F(a(0)) ' OK.  
 F(b(1)) ' Not allowed: System.ArrayTypeMismatchException.  
 End Sub  
End Module

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 sea realmente de un tipo permitido.

Module Test  
 Sub Fill(array() As Object, index As Integer, count As Integer, \_  
 value As Object)  
 Dim i As Integer  
  
 For i = index To (index + count) - 1  
 array(i) = value  
 Next i  
 End Sub  
  
 Sub Main()  
 Dim strings(100) As String  
  
 Fill(strings, 0, 101, "Undefined")  
 Fill(strings, 0, 10, Nothing)  
 Fill(strings, 91, 10, 0)  
 End Sub  
End Module

En este ejemplo, 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 la variable value es Nothing o bien una instancia de un tipo compatible con el tipo de elemento real de la matriz array. En el método Main, las dos primeras llamadas al método 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 Integer en una matriz String.

Si uno de los tipos de los elementos de la matriz es un parámetro de tipo cuyo tipo acaba siendo un tipo de valor en tiempo de ejecución, se producirá una excepción System.InvalidCastException. Por ejemplo:

Module Test  
 Sub F(Of T As U, U)(x() As T)  
 Dim y() As U = x  
 End Sub  
  
 Sub Main()  
 ' F will throw an exception because Integer() cannot be  
 ' converted to Object()  
 F(New Integer() { 1, 2, 3 })  
 End Sub  
End Module

También existen conversiones entre una matriz de un tipo enumerado y una matriz del tipo subyacente del tipo enumerado o una matriz de otro tipo enumerado con el mismo tipo subyacente, siempre que las matrices tengan el mismo rango.

Enum Color As Byte  
 Red  
 Green  
 Blue  
End Enum  
  
Module Test  
 Sub Main()  
 Dim a(10) As Color  
 Dim b() As Integer  
 Dim c() As Byte  
  
 b = a ' Error: Integer is not the underlying type of Color  
 c = a ' OK  
 a = c ' OK  
 End Sub  
End Module

En este ejemplo, una matriz de Color se convierte de y en una matriz del tipo subyacente Byte, Color. La conversión en una matriz de Integer, sin embargo, dará error porque Integer no es el tipo subyacente de Color.

Una matriz de rango 1 de tipo A() también tiene una conversión de matriz en los tipos de interfaz de la colección IList(Of B), IReadOnlyList(Of B), ICollection(Of B), IReadOnlyCollection(Of B) e IEnumerable(Of B) que se encuentran en System.Collections.Generic, siempre que se cumpla una de las siguientes condiciones:

Tanto A como B son tipos de referencia o parámetros de tipo que no se sabe si son tipos de valor; y A tiene una conversión de referencia widening, matriz o parámetro de tipo en B; o

Tanto A como B son tipos enumerados del mismo tipo subyacente; o

A o B es un tipo enumerado, y el otro es su tipo subyacente.

Toda matriz de tipo A de cualquier rango también tiene una conversión de matriz en los tipos de interfaz de colección no genérica IList, ICollection e IEnumerable que se encuentran en System.Collections.

Es posible recorrer en iteración las interfaces resultantes con For Each o mediante la invocación de los métodos GetEnumerator directamente. En el caso de las matrices de rango 1 convertidas en formas genéricas o no genéricas de IList o ICollection, también es posible obtener elementos por índice. En el caso de las matrices de rango 1 convertidas en formas genéricas o no genéricas de IList, también es posible establecer elementos por índice, sujetos a las mismas comprobaciones de covarianza de matrices en tiempo de ejecución descritas anteriormente. El comportamiento de todos los métodos de interfaz restantes no está definido por la especificación del lenguaje VB; depende del runtime subyacente.

## Conversiones de tipo de valor

Un valor de tipo de valor se puede convertir en uno de sus tipos de referencia base o en un tipo de interfaz que implementa a través de un proceso denominado boxing. Cuando a un valor de tipo de valor se le aplica la conversión boxing, el valor se copia de la ubicación en la que reside en el montón de .NET Framework. Después se devuelve una referencia a esta ubicación en el montón y se puede almacenar en una variable de tipo de referencia. Esta referencia también se denomina instancia con conversión boxing del tipo de valor. La instancia con conversión boxing tiene la misma semántica que el tipo de referencia, no la del tipo de valor.

Los tipos de valor con conversión boxing pueden volver a convertirse en su valor original mediante el proceso denominado conversión unboxing. Cuando se aplica la conversión unboxing a un tipo de valor con conversión boxing, el valor se copia del montón en la ubicación de la variable. A partir de ese momento, se comporta como si fuera un tipo de valor. Cuando se aplica la conversión unboxing a un tipo de valor, el valor debe ser null o una instancia del tipo de valor. De lo contrario, se producirá una excepción System.InvalidCastException. Si el valor es una instancia de un tipo enumerado, ese valor también puede convertirse en el tipo subyacente del tipo enumerado u otro tipo enumerado que tenga el mismo tipo subyacente. Un valor null se trata como si fuera el literal Nothing.

Para admitir los tipos de valor que aceptan valores null, el tipo de valor System.Nullable(Of T) se trata de un modo especial en las conversiones boxing y unboxing. Aplicar la conversión boxing a un valor de tipo Nullable(Of T) da como resultado un valor convertido de tipo T si la propiedad HasValue del valor es True o un valor de Nothing si la propiedad HasValue del valor es False. Aplicar la conversión unboxing a un valor de tipo T en Nullable(Of T) da como resultado una instancia de Nullable(Of T) cuya propiedad Value es el valor con conversión boxing y cuya propiedad HasValue es True. Al valor Nothing se le puede aplicar la conversión unboxing en Nullable(Of T) para cualquier T y da como resultado un valor cuya propiedad HasValue es False. Como los tipos de valor con conversión boxing se comportan como tipos de referencia, es posible crear múltiples referencias al mismo valor. Para los tipos primitivos y enumerados, esto es irrelevante porque sus instancias son inmutables. Es decir, no es posible modificar una instancia con conversión boxing de esos tipos, de forma que no es posible observar el hecho de que hay múltiples referencias al mismo valor.

Las estructuras, por otra parte, pueden ser mutables si las variables de instancia son accesibles o si sus métodos o propiedades modifican sus variables de instancia. Si se emplea una referencia a una estructura con conversión boxing para modificarla, todas las referencias a esta estructura convertida verán el cambio. Como este resultado puede ser inesperado, cuando un valor con tipo Object se copia de una ubicación en otra, los tipos de valor con conversión boxing serán clonados automáticamente en el montón, en lugar de simplemente copiar sus referencias. Por ejemplo:

Class Class1  
 Public Value As Integer = 0  
End Class  
  
Structure Struct1  
 Public Value As Integer  
End Structure  
  
Module Test  
 Sub Main()  
 Dim val1 As Object = New Struct1()  
 Dim val2 As Object = val1  
  
 val2.Value = 123  
  
 Dim ref1 As Object = New Class1()  
 Dim ref2 As Object = ref1  
  
 ref2.Value = 123  
  
 Console.WriteLine("Values: " & val1.Value & ", " & val2.Value)  
 Console.WriteLine("Refs: " & ref1.Value & ", " & ref2.Value)  
 End Sub  
End Module

El resultado del programa es:

Values: 0, 123  
Refs: 123, 123

La asignación al campo de la variable local val2 no tiene efecto en el campo de la variable local val1 porque cuando Struct1 (con conversión boxing) se asignó a val2, se hizo una copia del valor. Por el contrario, la asignación ref2.Value = 123 afecta al objeto al que tanto ref1 como ref2 hacen referencia.

Anotación

La copia de estructura no se hace con estructuras con conversión boxing y tipo System.ValueType porque no es posible el enlace en tiempo de ejecución fuera de System.ValueType.

Hay una excepción a esta regla: los tipos de valor con conversión boxing se copiarán durante la asignación. Si se almacena una referencia a un tipo de valor con conversión boxing dentro de otro tipo, la referencia interna no se copiará. Por ejemplo:

Structure Struct1  
 Public Value As Object  
End Structure  
  
Module Test  
 Sub Main()  
 Dim val1 As Struct1  
 Dim val2 As Struct1  
  
 val1.Value = New Struct1()  
 val1.Value.Value = 10  
  
 val2 = val1  
 val2.Value.Value = 123  
 Console.WriteLine("Values: " & val1.Value.Value & ", " & \_  
 val2.Value.Value)  
 End Sub  
End Module

El resultado del programa es:

Values: 123, 123

Esto es así porque el valor con conversión boxing interno no se copia cuando se copia el valor. Por consiguiente, tanto val1.Value como val2.Value tienen una referencia al mismo tipo de valor con conversión boxing.

Anotación

El hecho de que los tipos de valor con conversión boxing no se copien es una limitación del sistema de tipos de .NET: asegurar que todos los tipos de valor con conversión boxing internos se copien cada vez que se copia un valor de tipo Object resultaría prohibitivamente caro.

Como se ha descrito previamente, los tipos de valor con conversión boxing solo pueden convertirse en su tipo original. Los tipos primitivos con conversión boxing, sin embargo, se tratan de manera especial cuando son de tipo Object. Pueden convertirse en cualquier otro tipo primitivo en el que tengan una conversión. Por ejemplo:

Module Test  
 Sub Main()  
 Dim o As Object = 5  
 Dim b As Byte = CByte(o) ' Legal  
 Console.WriteLine(b) ' Prints 5  
 End Sub  
End Module

Por lo general, al valor Integer con conversión boxing 5 no se le podría aplicar la conversión unboxing en una variable Byte. Ahora bien, como Integer y Byte son tipos primitivos y tienen una conversión, la conversión se permite.

Es importante observar que convertir un tipo de valor en una interfaz es diferente que un argumento genérico restringido a una interfaz. Cuando se tiene acceso a los miembros de interfaz de un parámetro de tipo restringido (o se llama a métodos en Object), no se aplica la conversión boxing como se hace cuando el tipo de valor se convierte en una interfaz y se tiene acceso a un miembro de interfaz. 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 no una copia con conversión boxing aplicada:

Interface ICounter  
 Sub Increment()  
 ReadOnly Property Value() As Integer  
End Interface  
  
Structure Counter  
 Implements ICounter  
  
 Dim \_value As Integer  
  
 Property Value() As Integer Implements ICounter.Value  
 Get  
 Return \_value  
 End Get  
 End Property  
  
 Sub Increment() Implements ICounter.Increment  
 value += 1  
 End Sub  
End Structure  
  
Module Test  
 Sub Test(Of T As ICounter)(x As T)  
 Console.WriteLine(x.value)  
 x.Increment() ' Modify x  
 Console.WriteLine(x.value)  
 CType(x, ICounter).Increment() ' Modify boxed copy of x  
 Console.WriteLine(x.value)  
 End Sub  
  
 Sub Main()  
 Dim x As Counter  
 Test(x)  
 End Sub  
End Module

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

### Conversiones de tipos de valor que admiten null

Un tipo de valor T se puede convertir en la versión que admite null del tipo, T? y viceversa. La conversión de T? en T produce una excepción System.InvalidOperationException si el valor que se convierte es Nothing. Asimismo, T? tiene una conversión en un tipo S si T tiene una conversión intrínseca en S. Y si S es un tipo de valor, las siguientes conversiones intrínsecas existen entre T? y S?:

Una conversión de la misma clasificación (narrowing o widening) de T? a S?.

Una conversión de la misma clasificación (narrowing o widening) de T a S?.

Una conversión narrowing de S? en T.

Por ejemplo, existe una conversión widening intrínseca de Integer? en Long? porque existe una conversión widening intrínseca de Integer en Long:

Dim i As Integer? = 10  
Dim l As Long? = i

Cuando se convierte T? en S?, si el valor de T? es Nothing, entonces el valor de S? será Nothing. Cuando se convierte S? en T o T? en S, si el valor de T? o de S? es Nothing, se producirá una excepción System.InvalidCastException.

Debido al comportamiento del tipo subyacente System.Nullable(Of T), cuando se realiza la conversión boxing de un tipo de valor T? que acepta valores NULL, el resultado es un valor al que se ha aplicado la conversión boxing del tipo T, no del tipo T?. Y, al contrario, cuando se aplica una conversión unboxing a un tipo de valor que no acepta null T?, el valor se empaquetará en System.Nullable(Of T) y Nothing se convertirá en un valor que acepta valores null de tipo T?. Por ejemplo:

Dim i1? As Integer = Nothing  
Dim o1 As Object = i1  
  
Console.WriteLine(o1 Is Nothing) ' Will print True  
o1 = 10  
i1 = CType(o1, Integer?)  
Console.WriteLine(i1) ' Will print 10

Un efecto secundario de este comportamiento es que aparece un tipo de valor que acepta valores null T? para implementar todas las interfaces de T, porque convertir un tipo de valor en una interfaz exige que el tipo de valor tenga conversión boxing. Como resultado, T? es convertible en todas las interfaces en las que se puede convertir T. Es importante observar, sin embargo, que un tipo de valor que admite valores null T? no implementa en realidad las interfaces de T con el fin de comprobar las restricciones genéricas o la reflexión. Por ejemplo:

Interface I1  
End Interface  
  
Structure T1  
 Implements I1  
 ...  
End Structure  
  
Module Test  
 Sub M1(Of T As I1)(ByVal x As T)  
 End Sub  
  
 Sub Main()  
 Dim x? As T1 = Nothing  
 Dim y As I1 = x ' Valid  
 M1(x) ' Error: x? does not satisfy I1 constraint  
 End Sub  
End Module

## Conversiones de cadenas

Convertir Char en String da como resultado una cadena cuyo primer carácter es el valor del carácter. Convertir String en Char da como resultado un carácter cuyo valor es el primer carácter de la cadena. Convertir una matriz de Char en String da como resultado una cadena cuyos caracteres son los elementos de la matriz. Convertir String en una matriz de Char da como resultado una matriz de caracteres cuyos elementos son los caracteres de la cadena.

Las conversiones exactas entre String y Boolean, Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long, Decimal, Single, Double, Date y viceversa sobrepasan el ámbito de esta especificación y son dependientes de la implementación con la excepción de un detalle. Las conversiones de cadena siempre consideran la referencia cultural del entorno del tiempo de ejecución. Como tal, siempre deben efectuarse en tiempo de ejecución.

## Conversiones widening

Las conversiones widening (de ampliación) nunca se desbordan pero pueden conllevar una pérdida de precisión. Las conversiones siguientes son conversiones de ampliación:

Conversiones de identidad/predeterminadas

De un tipo en sí mismo.

De un tipo de delegado anónimo generado para una reclasificación de método lambda en cualquier tipo de delegado con una firma idéntica.

Del literal Nothing en un tipo.

Conversiones numéricas

De Byte en UShort, Short, UInteger, Integer, ULong, Long, Decimal, Single o Double.

De SByte en Short, Integer, Long, Decimal, Single o Double.

De UShort en UInteger, Integer, ULong, Long, Decimal, Single o Double.

De Short en Integer, Long, Decimal, Single o Double.

De UInteger en ULong, Long, Decimal, Single o Double.

De Integer en Long, Decimal, Single o Double.

De ULong en Decimal, Single o Double.

De Long en Decimal, Single o Double.

De Decimal en Single o Double.

De Single en Double.

Del literal 0 en un tipo enumerado.

Anotación

La conversión de 0 en cualquier tipo enumerado es de ampliación (widening) para simplificar los marcas de pruebas. Por ejemplo, si Values es un tipo enumerado con un valor One, se podría probar una variable v de tipo Values mediante (v And Values.One) = 0.

De un tipo enumerado en su tipo numérico subyacente, o en un tipo numérico cuyo tipo numérico subyacente tiene una conversión widening.

De una expresión constante de tipo ULong, Long, UInteger, Integer, UShort, Short, Byte o SByte en un tipo más estrecho, siempre que el valor de la expresión constante quede dentro del intervalo del tipo de destino.

Nota Las conversiones de UInteger o Integer en Single, ULong o Long en Single o Double, o Decimal en Single o Double pueden producir una pérdida de precisión, pero nunca una pérdida de magnitud. En las demás conversiones widening numéricas nunca se pierde información.

Conversiones de referencias

De un tipo de referencia en un tipo base.

De un tipo de referencia en un tipo de interfaz, siempre que el tipo implemente la interfaz o una interfaz compatible con variante.

De un tipo de interfaz en Object.

De un tipo de interfaz en un tipo de interfaz compatible con variante.

De un tipo de delegado en un tipo de delegado compatible con variante.

Nota: estas reglas implican muchas otras conversiones de referencia. Por ejemplo, los delegados anónimos son tipos de referencia que heredan de System.MulticastDelegate; los tipos de matriz son tipos de referencia que heredan de System.Array; los tipos anónimos son tipos de referencia que heredan de System.Object.

Conversiones de delegados anónimos

De un tipo de delegado anónimo generado para la reclasificación de un método lambda en cualquier tipo de delegado más amplio.

Conversiones de matrices

De un tipo de matriz 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.

Tanto SE como TE son tipos de referencia o parámetros de tipo que se sabe que son un tipo de referencia.

Existe una conversión de referencia widening, matriz o parámetro de tipo entre SE a TE.

De un tipo de matriz S con un tipo de elemento enumerado 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.

TE es el tipo subyacente de SE.

De un tipo de matriz S de rango 1 con un tipo de elemento enumerado SE, en System.Collections.Generic.IList(Of TE), IReadOnlyList(Of TE), ICollection(Of TE), IReadOnlyCollection(Of TE) e IEnumerable(Of TE), siempre que se cumpla una de las siguientes condiciones:

Tanto SE como TE son tipos de referencia o parámetros de tipo que se sabe que son un tipo de referencia, y existe una conversión de referencia widening, matriz o parámetro de tipo de SE en TE; o

TE es el tipo subyacente de SE; o

TE es idéntico a SE. Conversiones de tipo de valor

De un tipo de valor en un tipo base.

De un tipo de valor en un tipo de interfaz que el tipo implementa.

Conversiones de tipos de valor que admiten valores null

De un tipo T en el tipo T?.

De un tipo T? en un tipo S?, donde hay una conversión widening del tipo T en el tipo S.

De un tipo T en un tipo S?, donde hay una conversión widening del tipo T en el tipo S.

De un tipo T? en un tipo de interfaz que el tipo T implementa.

Conversiones de cadenas

De Char en String.

De Char() en String.

Conversiones de parámetros de tipo

De un parámetro de tipo en Object.

De un parámetro de tipo en una restricción de tipo de interfaz o cualquier interfaz compatible con variante con una restricción de tipo de interfaz.

De un parámetro de tipo en una interfaz implementada por una restricción de clases.

De un parámetro de tipo en una interfaz compatible con variante con una interfaz implementada por una restricción de clase.

De un parámetro de tipo en una restricción de clases o un tipo base de la restricción de clases.

De un parámetro de tipo T en una restricción de parámetro de tipo TX o cualquier cosa donde TX tenga una conversión widening.

## Conversiones de restricción

Las conversiones narrowing (de restricción) son conversiones que no se puede demostrar que siempre se realicen correctamente, conversiones en las que se sabe que se puede producir pérdida de información y conversiones entre dominios de tipos que, por sus diferencias, merecen una notación restrictiva (narrowing). Las siguientes conversiones se clasifican como conversiones narrowing (restrictivas):

Conversiones booleanas

De Boolean en Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long, Decimal, Single o Double.

De Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long, Decimal, Single o Double en Boolean.

Conversiones numéricas

De Byte en SByte.

De SByte en Byte, UShort, UInteger o ULong.

De UShort en Byte, SByte o Short.

De Short en Byte, SByte, UShort, UInteger o ULong.

De UInteger en Byte, SByte, UShort, Short o Integer.

De Integer en Byte, SByte, UShort, Short, UInteger o ULong.

De ULong en Byte, SByte, UShort, Short, UInteger, Integer o Long.

De Long en Byte, SByte, UShort, Short, UInteger, Integer o ULong.

De Decimal en Byte, SByte, UShort, Short, UInteger, Integer, ULong o Long.

De Single en Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long o Decimal.

De Double en Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long, Decimal o Single.

De un tipo numérico en un tipo enumerado.

De un tipo enumerado en un tipo numérico cuyo tipo numérico subyacente tiene una conversión de restricción (narrowing).

De un tipo enumerado en otro tipo enumerado.

Conversiones de referencias

De un tipo de referencia en un tipo más derivado.

De un tipo de clase en un tipo de interfaz, siempre que el tipo de clase no implemente el tipo de interfaz o un tipo de interfaz compatible con variante con él.

De un tipo de referencia en un tipo de clase.

De un tipo de interfaz en otro tipo de interfaz, siempre que no haya relación de herencia entre los dos tipos y siempre que no sean compatibles con variante.

Conversiones de delegados anónimos

De un tipo de delegado anónimo generado para la reclasificación de un método lambda en cualquier tipo de delegado más restrictivo.

Conversiones de matrices

De un tipo de matriz 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.

Tanto SE como TE son tipos de referencia o parámetros de tipo que no se sabe si son tipos de valor.

Existe una conversión de referencia narrowing, matriz o parámetro de tipo entre SE en TE.

De un tipo de matriz S con un tipo de elemento SE a un tipo de matriz T con un tipo de elemento enumerado TE, siempre que se cumpla todo lo siguiente:

S y T difieren solamente en el tipo de elemento.

SE es el tipo subyacente de TE o ambos son tipos enumerados diferentes que comparten el mismo tipo subyacente.

De un tipo de matriz S de rango 1 con un tipo de elemento enumerado SE, en IList(Of TE), IReadOnlyList(Of TE), ICollection(Of TE), IReadOnlyCollection(Of TE) e IEnumerable(Of TE), siempre que se cumpla una de las siguientes condiciones:

Tanto SE como TE son tipos de referencia o parámetros de tipo que se sabe que son un tipo de referencia, y existe una conversión de referencia narrowing, matriz o parámetro de tipo entre SE en TE; o

SE es el tipo subyacente de TE o ambos son tipos enumerados diferentes que comparten el mismo tipo subyacente.

Conversiones de tipos de valor

De un tipo de referencia en un tipo de valor más derivado.

De un tipo de interfaz en un tipo de valor, siempre que el tipo de valor implemente el tipo de interfaz.

Conversiones de tipos de valor que admiten valores null

De un tipo T? en un tipo T.

De un tipo T? en un tipo S?, donde hay una conversión del tipo T en el tipo S.

De un tipo T en un tipo S?, donde hay una conversión del tipo T en el tipo S.

De un tipo S? en un tipo T, donde hay una conversión del tipo S en el tipo T.

Conversiones de cadenas

De String en Char.

De String en Char().

De String en Boolean y de Boolean en String.

Conversiones entre String y Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long, Decimal, Single o Double.

De String en Date y de Date en String.

Conversiones de parámetros de tipo

De Object en un parámetro de tipo.

De un parámetro de tipo en un tipo de interfaz, siempre que el parámetro de tipo no esté restringido a esa interfaz ni a una clase que implemente esa interfaz.

De un tipo de interfaz en un parámetro de tipo.

De un parámetro de tipo en un tipo derivado de una restricción de clase.

De un parámetro de tipo T en cualquier cosa donde una restricción de parámetro de tipo TX tenga una conversión narrowing.

## Conversiones de parámetros de tipo

Las conversiones de parámetros de tipo están determinadas por las restricciones que pudieran tener. Un parámetro de tipo T siempre puede convertirse en sí mismo, de y en Object, así como de y en un tipo de interfaz. Observe que si el tipo T es un tipo de valor en tiempo de ejecución, convertir de T en Object o un tipo de interfaz será una conversión boxing, y convertir de Object o un tipo de interfaz en T será una conversión unboxing. Un parámetro de tipo con una restricción de clase C define conversiones adicionales del parámetro de tipo en C y sus clases base, y viceversa. Un parámetro de tipo T con una restricción de parámetro de tipo TX define una conversión en TX y cualquier cosa en la que TX se convierta.

Una matriz cuyo tipo de elemento es un parámetro de tipo con una restricción de interfaz I tiene las mismas conversiones de matriz covariante que una matriz cuyo tipo de elemento es I, siempre que el tipo de parámetro tenga también una Class o restricción de clase (puesto que solo los tipos de elementos de matriz de referencia pueden ser covariantes). Una matriz cuyo tipo de elemento es un parámetro de tipo con una restricción de clase C tiene las mismas conversiones de matriz covariante que una matriz cuyo tipo de elemento sea C.

Las reglas de conversión anteriores no permiten conversiones de parámetros de tipo sin restricciones en tipos que no son de interfaz, lo que puede resultar sorprendente. El razón es evitar la confusión sobre la semántica de dichas conversiones. Por ejemplo, en la siguiente declaración:

Class X(Of T)  
 Public Shared Function F(t As T) As Long   
 Return CLng(t) ' Error, explicit conversion not permitted  
 End Function  
End Class

Si se permitiese la conversión de T en Integer, el resultado esperado de X(Of Integer).F(7) sería 7L. Sin embargo, no es así porque las conversiones numéricas solo se consideran cuando se sabe que 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(Of T)  
 Public Shared Function F(t As T) As Long  
 Return CLng(CObj(t)) ' OK, conversions permitted  
 End Function  
End Class

## Conversiones definidas por el usuario

Las conversiones intrínsecas son conversiones definidas por el lenguaje (es decir, enumeradas en esta especificación), mientras que las conversiones definidas por el usuario se definen mediante la sobrecarga del operador CType. En las conversiones entre tipos, si no hay conversiones intrínsecas aplicables, se tendrán en cuenta las conversiones definidas por el usuario. Si hay una conversión definida por el usuario que es más específica para los tipos de origen y de destino, se usará la conversión definida por el usuario. En caso contrario, se producirá un error en tiempo de compilación. La conversión más específica es aquella cuyo operando es el “más próximo” al tipo de origen y cuyo tipo de resultado es el “más próximo” al tipo de destino. Cuando se determina qué conversión definida por el usuario utilizar, se empleará la conversión widening más específica; si ninguna conversión widening es más específica, se utilizará la conversión narrowing más específica. Si no hay ninguna conversión narrowing más específica, la conversión no se define y se produce un error en tiempo de compilación.

En las secciones siguientes se trata el modo en que se determinan las conversiones más específicas. Usan los términos siguientes:

Si existe una conversión widening intrínseca de un tipo A en un tipo B y si ni A ni B son interfaces, entonces se dice que A está abarcado por B y que B abarca A.

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 intuitivos, el tipo que más abarca es el “mayor” del conjunto, el tipo en el que pueden convertirse todos los demás tipos mediante una conversión widening.

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 intuitivos, el tipo más abarcado es el “menor” del conjunto, aquel que puede convertirse en todos los demás tipos mediante una conversión narrowing.

Cuando se recopilan las conversiones definidas por el usuario para un tipo T?, se utilizan los operadores de conversión definidos por el usuario que son definidos por T. Si el tipo en el que se convierte es también un tipo de valor que admite valores null, se elevan los operadores de las conversiones definidas por el usuario de T que implican únicamente tipos de valor que no admiten valores null. Un operador de conversión de T en S se eleva para ser una conversión de T? en S? y se evalúa convirtiendo T? en T, si es necesario, evaluando el operador de conversión definido por el usuario de T en S y después convirtiendo S en S?, si es necesario. Ahora bien, si el valor que se convierte es Nothing, un operador de conversión elevado se convierte directamente en un valor de Nothing de tipo S?. Por ejemplo:

Structure S  
 ...  
End Structure  
  
Structure T  
 Public Shared Widening Operator CType(ByVal v As T) As S  
 ...  
 End Operator  
End Structure  
  
Module Test  
 Sub Main()  
 Dim x As T?  
 Dim y As S?  
  
 y = x ' Legal: y is still null  
 x = New T()  
 y = x ' Legal: Converts from T to S  
 End Sub  
End Module

Para resolver las conversiones, siempre se prefieren los operadores de conversión definidos por el usuario a los operadores de conversión de elevación. Por ejemplo:

Structure S  
 ...  
End Structure  
  
Structure T  
 Public Shared Widening Operator CType(ByVal v As T) As S  
 ...  
 End Operator  
  
 Public Shared Widening Operator CType(ByVal v As T?) As S?  
 ...  
 End Operator  
End Structure  
  
Module Test  
 Sub Main()  
 Dim x As T?  
 Dim y As S?  
  
 y = x ' Calls user-defined conversion, not lifted conversion  
 End Sub  
End Module

En tiempo de ejecución, la evaluación de una conversión definida por el usuario puede implicar hasta tres pasos:

Primero, el valor se convierte del tipo de origen en el tipo de operando mediante una conversión intrínseca, si es necesario.

Después se llama a la conversión definida por el usuario.

Por último, el resultado de una conversión definida por el usuario se convierte en el tipo de destino mediante el uso de una conversión intrínseca, si es necesario.

Es importante observar que la evaluación de una conversión definida por el usuario nunca necesitará más de un operador de conversión definida por el usuario.

### Conversión widening más específica

Determinar el operador más específico de conversión widening definida por el usuario entre dos tipos se consigue mediante los pasos siguientes:

Primero se recopilan todos los operadores de conversión candidatos. Los operadores de conversión candidatos son todos los operadores de conversión widening definida por el usuario del tipo de origen y todos los operadores de conversión widening definida por el usuario del tipo de destino.

Después, se quitan del conjunto todos los operadores de conversión no aplicables. Un operador de conversión es aplicable a un tipo de origen y a un tipo de destino si hay un operador de conversión widening intrínseco del tipo de origen en el tipo de operando y hay un operador de conversión widening intrínseco del resultado del operador en el tipo de destino. Si no hay operadores de conversión aplicables, entonces no hay conversión widening más específica.

El tipo de origen más específico de los operadores de conversión aplicables se determina:

Si alguno de los operadores de conversión se convierte directamente del tipo de origen, entonces el tipo de origen es el tipo de origen más específico.

De lo contrario, el tipo de origen más específico es el tipo más abarcado del conjunto combinado de tipos de origen de los operadores de conversión. Si no se encuentra el tipo más abarcado, entonces no hay conversión widening más específica.

El tipo de destino más específico de los operadores de conversión aplicables se determina:

Si alguno de los operadores de conversión se convierte directamente en el tipo de destino, entonces el tipo de destino es el tipo de destino más específico.

De lo contrario, el tipo de destino más específico es el tipo que más abarca del conjunto combinado de tipos de destino de los operadores de conversión. Si no se encuentra el tipo que más abarca, entonces no hay conversión widening más específica.

Si exactamente un operador de conversión se convierte del operador de conversión más específico en el tipo de destino más específico, entonces este es el operador de conversión más específico. Si existe más de un operador así, no hay conversión widening más específica.

### Conversión narrowing más específica

Determinar el operador de conversión narrowing definida por el usuario más específico entre dos tipos se consigue mediante los pasos siguientes:

Primero se recopilan todos los operadores de conversión candidatos. Los operadores de conversión candidatos son todos los operadores de conversión definida por el usuario del tipo de origen y todos los operadores de conversión definida por el usuario del tipo de destino.

Después, se quitan del conjunto todos los operadores de conversión no aplicables. Un operador de conversión es aplicable a un tipo de origen y a un tipo de destino si hay un operador de conversión intrínseco del tipo de origen en el tipo de operando y hay un operador de conversión intrínseco del resultado del operador en el tipo de destino. Si no hay operadores de conversión aplicables, entonces no hay conversión narrowing más específica.

El tipo de origen más específico de los operadores de conversión aplicables se determina:

Si alguno de los operadores de conversión se convierte directamente del tipo de origen, entonces el tipo de origen es el tipo de origen más específico.

De lo contrario, si uno de los operadores de conversión se convierte de tipos que abarca el tipo de origen, entonces el tipo de origen más específico es el tipo más abarcado del conjunto combinado de tipos de origen de estos operadores de conversión. Si no se encuentra el tipo más abarcado, entonces no hay conversión narrowing más específica.

De lo contrario, el tipo de origen más específico es el tipo que más abarca del conjunto combinado de tipos de origen de los operadores de conversión. Si no se encuentra el tipo que más abarca, entonces no hay conversión narrowing más específica.

El tipo de destino más específico de los operadores de conversión aplicables se determina:

Si alguno de los operadores de conversión se convierte directamente en el tipo de destino, entonces el tipo de destino es el tipo de destino más específico.

De lo contrario, si uno de los operadores de conversión se convierte en tipos que abarca el tipo de destino, entonces el tipo de destino más específico es el tipo que más abarca del conjunto combinado de tipos de origen de estos operadores de conversión. Si no se encuentra el tipo que más abarca, entonces no hay conversión narrowing más específica.

De lo contrario, el tipo de destino más específico es el tipo más abarcado del conjunto combinado de tipos de destino de los operadores de conversión. Si no se encuentra el tipo más abarcado, entonces no hay conversión narrowing más específica.

Si exactamente un operador de conversión se convierte del operador de conversión más específico en el tipo de destino más específico, entonces este es el operador de conversión más específico. Si existe más de un operador así, entonces no hay conversión narrowing más específica.

## Conversiones nativas

Varias de estas conversiones están clasificadas como conversiones nativas porque se admiten en forma nativa en .NET Framework. Estas conversiones son las que se pueden optimizar mediante los operadores de conversión DirectCast y TryCast, además de otros comportamientos especiales. Las conversiones que se clasifican como nativas son: conversiones de identidad, predeterminadas, de referencia, de matriz, de tipo de valor y de parámetro de tipo.

## Tipo dominante

Dado un conjunto de pasos, a menudo es necesario en situaciones como la inferencia de tipos determinar el tipo dominante del conjunto. El tipo dominante de un conjunto de tipos se determina quitando primero cualquier tipo con los que uno o varios tipos no tengan una conversión implícita. Si no quedan tipos en este punto, no hay tipo dominante. El tipo dominante es, en este caso, el tipo más abarcado de los tipos restantes. Si existe más de un tipo que es el más abarcado, entonces no hay tipo dominante.

# Miembros de tipo

Los miembros de tipo definen las ubicaciones de almacenamiento y el código ejecutable. Pueden ser métodos, constructores, eventos, constantes, variables y propiedades.

## Implementación de métodos de interfaz

Los métodos, las propiedades y los eventos pueden implementar miembros de interfaz. Para implementar un miembro de interfaz, una declaración de miembro especifica la palabra clave Implements y enumera uno o varios miembros de interfaz. Los métodos y las propiedades que implementan miembros de interfaz son implícitamente NotOverridable salvo que se declaren para ser MustOverride, Overridable o invaliden otro miembro. Es un error que un miembro que implementa un miembro de interfaz sea Shared. La accesibilidad de un miembro no influye en su capacidad para implementar miembros de interfaz.

Para que una implementación de interfaz sea válida, la lista Implements del tipo contenedor debe nombrar una interfaz que contenga un miembro compatible. Un miembro compatible es aquel cuya firma coincide con la del miembro que implementa. Si se está implementando una interfaz genérica, entonces el argumento de tipo suministrado en la cláusula Implements se sustituye en la firma cuando se comprueba la compatibilidad. Por ejemplo:

Interface I1(Of T)  
 Sub F(x As T)  
End Interface  
  
Class C1  
 Implements I1(Of Integer)  
  
 Sub F(x As Integer) Implements I1(Of Integer).F  
 End Sub  
End Class  
  
Class C2(Of U)  
 Implements I1(Of U)  
  
 Sub F(x As U) Implements I1(Of U).F  
 End Sub  
End Class

Si un evento declarado usando un tipo delegado está implementando un evento de interfaz, entonces un evento compatible es aquel cuyo delegado subyacente sea el mismo tipo. De lo contrario, el evento usa el tipo de delegado del evento de interfaz que está implementando. Si tal evento implementa múltiples eventos de interfaz, todos los eventos de interfaz deben tener el mismo tipo de delegado subyacente. Por ejemplo:

Interface ClickEvents  
 Event LeftClick(x As Integer, y As Integer)  
 Event RightClick(x As Integer, y As Integer)  
End Interface  
  
Class Button  
 Implements ClickEvents  
  
 ' OK. Signatures match, delegate type = ClickEvents.LeftClickHandler.  
 Event LeftClick(x As Integer, y As Integer) \_  
 Implements ClickEvents.LeftClick  
  
 ' OK. Signatures match, delegate type = ClickEvents.RightClickHandler.  
 Event RightClick(x As Integer, y As Integer) \_  
 Implements ClickEvents.RightClick  
End Class  
  
Class Label  
 Implements ClickEvents  
  
 ' Error. Signatures match, but can't be both delegate types.  
 Event Click(x As Integer, y As Integer) \_  
 Implements ClickEvents.LeftClick, ClickEvents.RightClick  
End Class

Un miembro de interfaz de la lista Implements se especifica usando un nombre de tipo, un punto y un identificador. El nombre de tipo debe ser una interfaz de la lista Implements o una interfaz base de una interfaz de la lista Implements, y el identificador debe ser un miembro de la interfaz especificada. Un único miembro puede implementar más de un miembro de interfaz coincidente.

Interface ILeft  
 Sub F()  
End Interface  
  
Interface IRight  
 Sub F()  
End Interface  
  
Class Test  
 Implements ILeft, IRight  
  
 Sub F() Implements ILeft.F, IRight.F  
 End Sub  
End Class

Si el miembro de interfaz que se está implementando no está disponible en todas las interfaces implementadas de forma explícita debido a una herencia de múltiples interfaces, el miembro que implementa debe hacer referencia explícita a una interfaz base en la que el miembro esté disponible. Por ejemplo, si I1 e I2 contienen un miembro M e I3 hereda de I1 e I2, un tipo que implemente I3 implementará I1.M e I2.M. Si una interfaz prevalece sobre múltiples miembros heredados, un tipo que implementa tendrá que implementar los miembros heredados y los miembros que prevalecen.

Interface ILeft  
 Sub F()  
End Interface  
  
Interface IRight  
 Sub F()  
End Interface  
  
Interface ILeftRight  
 Inherits ILeft, IRight  
  
 Shadows Sub F()  
End Interface  
  
Class Test  
 Implements ILeftRight  
  
 Sub LeftF() Implements ILeft.F  
 End Sub  
  
 Sub RightF() Implements IRight.F  
 End Sub  
  
 Sub LeftRightF() Implements ILeftRight.F  
 End Sub  
End Class

Si la interfaz contenedora del miembro de interfaz implementado es genérica, se deben ofrecer los mismos argumentos de tipo que la interfaz que se implementa. Por ejemplo:

Interface I1(Of T)  
 Function F() As T  
End Interface  
  
Class C1  
 Implements I1(Of Integer)  
 Implements I1(Of Double)  
  
 Function F1() As Integer Implements I1(Of Integer).F  
 End Function  
  
 Function F2() As Double Implements I1(Of Double).F  
 End Function  
  
 ' Error: I1(Of String) is not implemented by C1  
 Function F3() As String Implements I1(Of String).F  
 End Function  
End Class  
  
Class C2(Of U)  
 Implements I1(Of U)  
  
 Function F() As U Implements I1(Of U).F  
 End Function  
End Class

ImplementsClause ::= [ Implements ImplementsList ]

ImplementsList ::=  
 InterfaceMemberSpecifier |  
 ImplementsList Comma InterfaceMemberSpecifier

InterfaceMemberSpecifier ::= NonArrayTypeName Period IdentifierOrKeyword

## Métodos

Los métodos contienen las instrucciones programables de un programa. Los métodos, que tienen una lista opcional de parámetros y un valor de retorno opcional, pueden ser compartidos o no compartidos. Se tiene acceso a los métodos compartidos a través de la clase o de instancias de la clase. A los métodos no compartidos, también llamados métodos de instancia, se tiene acceso a través de instancias de la clase. En el ejemplo siguiente se muestra una clase Stack que tiene varios métodos compartidos (Clone y Flip) y varios métodos de instancia (Push, Pop y ToString):

Public Class Stack  
 Public Shared Function Clone(s As Stack) As Stack  
 ...  
 End Function  
  
 Public Shared Function Flip(s As Stack) As Stack  
 ...  
 End Function  
  
 Public Function Pop() As Object  
 ...  
 End Function  
  
 Public Sub Push(o As Object)  
 ...  
 End Sub   
  
 Public Overrides Function ToString() As String  
 ...  
 End Function   
End Class   
  
Module Test  
 Sub Main()  
 Dim s As Stack = New Stack()  
 Dim i As Integer  
  
 While i < 10  
 s.Push(i)  
 End While  
  
 Dim flipped As Stack = Stack.Flip(s)  
 Dim cloned As Stack = Stack.Clone(s)  
  
 Console.WriteLine("Original stack: " & s.ToString())  
 Console.WriteLine("Flipped stack: " & flipped.ToString())  
 Console.WriteLine("Cloned stack: " & cloned.ToString())  
 End Sub  
End Module

Los métodos admiten sobrecargas, lo que significa que varios métodos pueden tener el mismo nombre siempre que sus firmas sean únicas. La signatura de un método está formada por el número de parámetros y sus tipos. La signatura de un método no incluye específicamente el tipo de valor devuelto ni los modificadores de parámetros como Optional, ByRef o ParamArray. En el ejemplo siguiente se muestra una clase con un número de sobrecargas:

Module Test  
 Sub F()  
 Console.WriteLine("F()")  
 End Sub   
  
 Sub F(o As Object)  
 Console.WriteLine("F(Object)")  
 End Sub  
  
 Sub F(value As Integer)  
 Console.WriteLine("F(Integer)")  
 End Sub   
  
 Sub F(a As Integer, b As Integer)  
 Console.WriteLine("F(Integer, Integer)")  
 End Sub   
  
 Sub F(values() As Integer)  
 Console.WriteLine("F(Integer())")  
 End Sub   
  
 Sub G(s As String, Optional s2 As String = 5)  
 Console.WriteLine("G(String, Optional String")  
 End Sub  
  
 Sub G(s As String)  
 Console.WriteLine("G(String)")  
 End Sub  
  
  
 Sub Main()  
 F()  
 F(1)  
 F(CType(1, Object))  
 F(1, 2)  
 F(New Integer() { 1, 2, 3 })  
 G("hello")  
 G("hello", "world")  
 End Sub  
End Module

El resultado del programa es:

F()  
F(Integer)  
F(Object)  
F(Integer, Integer)  
F(Integer())  
G(String)  
G(String, Optional String)

Anotación

Las sobrecargas que solamente se diferencian en parámetros opcionales se pueden usar para el "control de versiones" de las bibliotecas. Por ejemplo, el valor v1 de una biblioteca puede incluir una función con parámetros opcionales:

Sub fopen(fileName As String, Optional accessMode as Integer = 0)

A continuación, el valor v2 de la biblioteca desea agregar otro parámetro opcional "password" sin romper la compatibilidad de origen (para que las aplicaciones destinadas a v1 se puedan recompilar) y sin romper la compatibilidad binaria (para que las aplicaciones que hacían referencia a v1 puedan ahora hacer referencia a v2 sin recompilarse). Este es el aspecto que tendrá v2:

Sub fopen(file As String, mode as Integer)  
 Sub fopen(file As String, Optional mode as Integer = 0, \_  
 Optional pword As String = "")

Observe que los parámetros opcionales en una API pública no son conformes a CLS. Sin embargo, los pueden usar al menos Visual Basic, C# 4 y F#.

MethodMemberDeclaration ::= MethodDeclaration | ExternalMethodDeclaration

InterfaceMethodMemberDeclaration ::= InterfaceMethodDeclaration

### Declaraciones de métodos normales, Async e Iterator

Hay dos tipos de métodos: subrutinas, que no devuelven valores, y funciones, que sí lo hacen. El cuerpo y el constructor End de un método solo se puede omitir si el método se define en una interfaz o tiene el modificador MustOverride. Si no se especifica ningún tipo devuelto en una función y se utiliza semántica estricta, se produce un error en tiempo de compilación; de lo contrario el tipo es implícitamente Object o el tipo del carácter de tipo del método. El dominio de accesibilidad del tipo devuelto y los tipos de parámetro de un método debe ser el mismo o un superconjunto del dominio de accesibilidad del propio método.

Un **método normal** es aquel que no tiene modificadores Async ni Iterator. Puede ser una subrutina o una función. En la sección 10.1.1 se detalla qué sucede al invocar un método normal.

Un **método iterator** es aquel que tiene un modificador Iterator y ninguno Async. Debe ser una función y su tipo de valor devuelto debe ser IEnumerator, IEnumerable, o bien IEnumerator(Of T) o IEnumerable(Of T) para T y no debe tener parámetros ByRef. En la sección 10.1.2 se detalla qué sucede al invocar un método iterator.

Un **método async** es aquel que tiene un modificador Async y ninguno Iterator. Debe ser una subrutina o una función con tipo de valor devuelto Task o Task(Of T) para T y no debe tener parámetros ByRef. En la sección 10.1.3 se detalla qué sucede al invocar un método async.

Si un método no pertenece a uno de estos tres tipos, hay un error en tiempo de compilación. Las declaraciones de funciones y subrutinas son especiales porque sus instrucciones de inicio y fin deben comenzar en el inicio de una línea lógica. De manera adicional, el cuerpo de una declaración de función o subrutina que no es MustOverride debe comenzar en el inicio de una línea lógica. Por ejemplo:

Module Test  
 ' Illegal: Subroutine doesn’t start the line  
 Public x As Integer : Sub F() : End Sub  
  
 ' Illegal: First statement doesn’t start the line  
 Sub G() : Console.WriteLine("G")  
 End Sub  
  
 ' Illegal: End Sub doesn’t start the line  
 Sub H() : End Sub  
End Module

MethodDeclaration ::=  
 SubDeclaration |  
 MustOverrideSubDeclaration |  
 FunctionDeclaration |  
 MustOverrideFunctionDeclaration

InterfaceMethodDeclaration ::=  
 InterfaceSubDeclaration |  
 InterfaceFunctionDeclaration

SubSignature ::= Sub Identifier [ TypeParameterList ]  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ]

FunctionSignature ::= Function Identifier [ TypeParameterList ]  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ] [ As [ Attributes ] TypeName ]

SubDeclaration ::=  
 [ Attributes ] [ ProcedureModifier+ ] SubSignature [ HandlesOrImplements ] LineTerminator  
 Block  
 End Sub StatementTerminator

MustOverrideSubDeclaration ::=  
 [ Attributes ] MustOverrideProcedureModifier+ SubSignature [ HandlesOrImplements ]  
 StatementTerminator

InterfaceSubDeclaration ::=  
 [ Attributes ] [ InterfaceProcedureModifier+ ] SubSignature StatementTerminator

FunctionDeclaration ::=  
 [ Attributes ] [ ProcedureModifier+ ] FunctionSignature [ HandlesOrImplements ]  
 LineTerminator  
 Block  
 End Function StatementTerminator

MustOverrideFunctionDeclaration ::=  
 [ Attributes ] MustOverrideProcedureModifier+ FunctionSignature [ HandlesOrImplements ]  
 StatementTerminator

InterfaceFunctionDeclaration ::=  
 [ Attributes ] [ InterfaceProcedureModifier+ ] FunctionSignature StatementTerminator

ProcedureModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 Overridable |  
 NotOverridable |  
 Overrides |  
 Overloads |  
 Partial |  
 Iterator |  
 Async

MustOverrideProcedureModifier ::= ProcedureModifier | MustOverride

InterfaceProcedureModifier ::= Shadows | Overloads

HandlesOrImplements ::= HandlesClause | ImplementsClause

### Declaraciones de métodos externas

Una declaración de método externo presenta un nuevo método cuya implementación se ofrece de manera externa al programa. Como una declaración de método externo no ofrece implementación real, no tiene construcción End ni cuerpo de método. Los métodos externos son implícitamente compartidos, no pueden tener parámetros de tipo y no pueden controlar eventos ni implementar miembros de interfaz. Si no se especifica un tipo devuelto en una función y se usa semántica estricta, se produce un error en tiempo de compilación. De lo contrario, el tipo es implícitamente Object o el tipo del carácter de tipo del método. El dominio de accesibilidad del tipo devuelto y los tipos de parámetro de un método externo debe ser el mismo o un superconjunto del dominio de accesibilidad del propio método externo.

La cláusula de biblioteca de una declaración de método externo especifica el nombre del archivo externo que implementa el método. La cláusula de alias opcional es una cadena que especifica el ordinal numérico (precedido de un carácter #) o el nombre del método de un archivo externo. También se puede especificar un modificador de juego de caracteres único, que controla el juego de caracteres usado para calcular las referencias de cadenas durante una llamada al método externo. El modificador Unicode calcula las referencias de todas las cadenas como valores Unicode, el modificador Ansi calcula las referencias de todas las cadenas como valores ANSI y el modificador Auto calcula las referencias de las cadenas según las reglas de .NET Framework basándose en el nombre del método o el nombre del alias especificado. Si no se especifica un modificador, el valor predeterminado es Ansi.

Si se especifica Ansi o Unicode, se busca el nombre del método en el archivo externo sin modificación. Si se especifica Auto, la búsqueda del nombre del método depende de la plataforma. Si se considera que la plataforma es ANSI (por ejemplo, Windows 95, Windows 98, Windows ME), entonces el nombre del método se busca sin modificación. Si la búsqueda da error, se anexa A y se vuelve a intentar la búsqueda. Si se considera que la plataforma es Unicode (por ejemplo, Windows NT, Windows 2000, Windows XP), se anexa W y el nombre del método se busca sin modificación. Si la búsqueda da error, se vuelve a intentar sin W. Por ejemplo:

Module Test  
 ' All platforms bind to "ExternSub".  
 Declare Ansi Sub ExternSub Lib "ExternDLL" ()  
  
 ' All platforms bind to "ExternSub".  
 Declare Unicode Sub ExternSub Lib "ExternDLL" ()  
  
 ' ANSI platforms: bind to "ExternSub" then "ExternSubA".  
 ' Unicode platforms: bind to "ExternSubW" then "ExternSub".  
 Declare Auto Sub ExternSub Lib "ExternDLL" ()  
End Module

Las referencias de los tipos de datos que se pasan a los métodos externos se calculan según las convenciones de cálculo de referencias de datos de .NET Framework con una excepción. Las referencias de las variables de cadena que se pasan por valor (es decir, ByVal x As String) se calculan con el tipo OLE Automation BSTR y los cambios efectuados al BSTR en el método externo se reflejan en el argumento de cadena. Esto se debe a que el tipo String en métodos externos es mutable y este cálculo de referencias especial imita ese comportamiento. Las referencias de los parámetros de cadena que se pasan por referencia (es decir, ByRef x As String) se calculan como un puntero al tipo OLE Automation BSTR. Es posible invalidar estos comportamientos especiales especificando el atributo System.Runtime.InteropServices.MarshalAsAttribute en el parámetro.

En el ejemplo se muestra el uso de métodos externos.

Class Path  
 Declare Function CreateDirectory Lib "kernel32" ( \_  
 Name As String, sa As SecurityAttributes) As Boolean  
 Declare Function RemoveDirectory Lib "kernel32" ( \_  
 Name As String) As Boolean  
 Declare Function GetCurrentDirectory Lib "kernel32" ( \_  
 BufSize As Integer, Buf As String) As Integer  
 Declare Function SetCurrentDirectory Lib "kernel32" ( \_  
 Name As String) As Boolean  
End Class

ExternalMethodDeclaration ::=  
 ExternalSubDeclaration |  
 ExternalFunctionDeclaration

ExternalSubDeclaration ::=  
 [ Attributes ] [ ExternalMethodModifier+ ] Declare [ CharsetModifier ] Sub Identifier  
 LibraryClause [ AliasClause ] [ OpenParenthesis [ ParameterList ] CloseParenthesis ]   
 StatementTerminator

ExternalFunctionDeclaration ::=  
 [ Attributes ] [ ExternalMethodModifier+ ] Declare [ CharsetModifier ] Function Identifier  
 LibraryClause [ AliasClause ] [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 [ As [ Attributes ] TypeName ]  
 StatementTerminator

ExternalMethodModifier ::= AccessModifier | Shadows | Overloads

CharsetModifier ::= Ansi | Unicode | Auto

LibraryClause ::= Lib StringLiteral

AliasClause ::= Alias StringLiteral

### Métodos invalidables

El modificador Overridable indica que un método es invalidable. El modificador Overrides indica que un método invalida un método invalidable de tipo base que tiene la misma signatura. El modificador NotOverridable indica que un método invalidable no puede volver a invalidarse. Un modificador MustOverride indica que un método debe invalidarse en clases derivadas.

Algunas combinaciones de estos modificadores no son válidas:

Overridable y NotOverridable se excluyen mutuamente y no se pueden combinar.

MustOverride implica Overridable (y por tanto no puede especificarlo) y no puede combinarse con NotOverridable.

NotOverridable no puede combinarse con Overridable o MustOverride y debe combinarse con Overrides.

Overrides implica Overridable (y por tanto no puede especificarlo) y no puede combinarse con MustOverride.

También hay restricciones adicionales en los métodos invalidables:

Un método MustOverride no puede incluir un cuerpo de método o una construcción End, no puede invalidar otro método y solo puede aparecer en clases MustInherit.

Si un método especifica Overrides y no hay método base coincidente que invalidar, se produce un error en tiempo de compilación. Un método que invalida no puede especificar Shadows.

Un método no puede invalidar otro método si el dominio de accesibilidad del método que invalida no es el mismo que el del método que se está invalidando. La excepción es que un método que invalida un método Protected Friend en otro ensamblado que no tiene acceso Friend debe especificar Protected (no Protected Friend).

Los métodos Private no pueden ser Overridable, NotOverridable ni MustOverride y no pueden invalidar otros métodos.

Los métodos de clases NotInheritable no se pueden declarar Overridable ni MustOverride.

En el siguiente ejemplo se ilustran las diferencias entre los métodos invalidables y no invalidables:

Class Base  
 Public Sub F()  
 Console.WriteLine("Base.F")  
 End Sub  
  
 Public Overridable Sub G()  
 Console.WriteLine("Base.G")  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Public Shadows Sub F()  
 Console.WriteLine("Derived.F")  
 End Sub  
  
 Public Overrides Sub G()  
 Console.WriteLine("Derived.G")  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim d As Derived = New Derived()  
 Dim b As Base = d  
  
 b.F()  
 d.F()  
 b.G()  
 d.G()  
 End Sub  
End Module

En el ejemplo, la clase Base presenta un método F y un método Overridable G. La clase Derivedpresenta un nuevo método F, con lo cual se oculta el método F heredado y además se invalida el método G heredado. El ejemplo produce el siguiente resultado:

Base.F  
Derived.F  
Derived.G  
Derived.G

Observe que la instrucción b.G() invoca Derived.G, no Base.G. Esto se debe a que el tipo en tiempo de ejecución de la instancia (que es Derived), y no el tipo en tiempo de compilación (que es Base), determina la implementación del método real que se invoca.

### Métodos compartidos

El modificador Shared indica que un método es un método compartido. Un método compartido no opera en una instancia concreta de un tipo y no puede invocarse directamente desde un tipo en lugar de a través de una instancia concreta de un tipo. Es válido, sin embargo, usar una instancia para calificar un método compartido. No es válido hacer referencia a Me, MyClass o MyBase en un método compartido. Los métodos compartidos no pueden ser Overridable, NotOverridable ni MustOverride y no pueden invalidar otros métodos. Los métodos definidos en módulos e interfaces estándar no pueden especificar Shared, porque ya son implícitamente Shared.

Un método declarado en una clase o estructura sin un modificador Shared es un método de instancia. Un método de instancia opera en una instancia determinada de un tipo. Los métodos de instancia solo pueden invocarse mediante una instancia de un tipo y se puede hacer referencia a la instancia a través de la expresión Me.

En el ejemplo siguiente se ilustran las reglas para el acceso a miembros de instancia y compartidos:

Class Test  
 Private x As Integer  
 Private Shared y As Integer  
  
 Sub F()  
 x = 1 ' Ok, same as Me.x = 1.  
 y = 1 ' Ok, same as Test.y = 1.  
 End Sub  
  
 Shared Sub G()  
 x = 1 ' Error, cannot access Me.x.  
 y = 1 ' Ok, same as Test.y = 1.  
 End Sub  
  
 Shared Sub Main()  
 Dim t As Test = New Test()  
  
 t.x = 1 ' Ok.  
 t.y = 1 ' Ok.  
 Test.x = 1 ' Error, cannot access instance member through type.  
 Test.y = 1 ' Ok.  
 End Sub  
End Class

El método F muestra que en un miembro de función de instancia se puede utilizar un identificador para tener acceso tanto a miembros de instancia como a miembros compartidos. El método G muestra que en un miembro de función compartido, se produce un error si se obtiene acceso a un miembro de instancia a través de un identificador. El método Main muestra que en una expresión de acceso de miembro, se deben utilizar instancias para tener acceso a los miembros de instancia, pero a los miembros compartidos se puede tener acceso a través de tipos o instancias.

### Parámetros de métodos

Un parámetro es una variable que se puede usar para pasar información fuera y dentro de un método. Los parámetros de un método los declara la lista de parámetros del método, que consiste en uno o más parámetros separados por comas. Si no se especifica un tipo para un parámetro y se usa semántica estricta, se produce un error en tiempo de compilación. De lo contrario, el tipo predeterminado es Object o el tipo del carácter de tipo del parámetro. Incluso con una semántica permisiva, si un parámetro incluye una cláusula As, todos los parámetros deben especificar tipos.

Los parámetros se especifican como parámetros de valor, referencia, opcional o paramarray mediante los modificadores ByVal, ByRef, Optional y ParamArray, respectivamente. Un parámetro que no especifica ByRef o ByVal tiene el valor predeterminado ByVal.

El ámbito de los nombres de parámetro es el cuerpo completo del método y siempre son públicamente accesibles. La invocación de un método crea una copia, específica de esa invocación, de los parámetros, y la lista de argumentos de la invocación asigna valores o referencias de variables a los parámetros recién creados. Como las declaraciones de métodos externos y las declaraciones de delegados no tienen cuerpo, los nombres de parámetros duplicados se permiten en las listas de parámetros pero se desaconseja.

El identificador puede ir seguido del modificador de nombre que admite valores null ? para indicar que admite valores null, así como por modificadores de nombre de matriz para indicar que es una matriz. Pueden estar combinados, por ejemplo "ByVal x?() As Integer". No se permite el uso de límites de matriz explícitos; además, si el modificador de nombre que admite valores null está presente, también debe haber una cláusula As.

ParameterList ::=  
 Parameter |  
 ParameterList Comma Parameter

Parameter ::=  
 [ Attributes ] [ ParameterModifier+ ] ParameterIdentifier [ As TypeName ]  
 [ *Equals* ConstantExpression ]

ParameterModifier ::= ByVal | ByRef | Optional | ParamArray

ParameterIdentifier ::= Identifier IdentifierModifiers

#### Parámetros de valor

Un parámetro de valor se declara con un modificador ByVal explícito. Si se usa el modificador ByVal, no se puede especificar el modificador ByRef. Un parámetro de valor se genera al invocar el miembro al que pertenece el parámetro y se inicializa con el valor del argumento especificado en la invocación. Un parámetro de valores deja de existir al regresar el miembro.

Un método puede asignar nuevos valores a un parámetro de valor. Tales asignaciones solo 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.

Se usa un parámetro de valor cuando se pasa el valor de un argumento a un método y las modificaciones del parámetro no influyen en el argumento original. Un parámetro de valor se refiere a su propia variable, una que es distinta de la variable del argumento correspondiente. Esta variable se inicializa copiando el valor del argumento correspondiente. En el ejemplo siguiente se muestra un método F que tiene un parámetro de valor denominado p:

Module Test  
 Sub F(p As Integer)  
 Console.WriteLine("p = " & p)  
 p += 1  
 End Sub   
  
 Sub Main()  
 Dim a As Integer = 1  
  
 Console.WriteLine("pre: a = " & a)  
 F(a)  
 Console.WriteLine("post: a = " & a)  
 End Sub  
End Module

El ejemplo produce el resultado siguiente, aunque se modifique el parámetro de valor p:

pre: a = 1  
p = 1  
post: a = 1

#### Parámetros de referencia

Un parámetro de referencia es un parámetro que se declara con un modificador ByRef. Si se especifica el modificador ByRef, no se puede usar el modificador ByVal. Un parámetro de referencia no crea una nueva ubicación de almacenamiento. En lugar de ello, un parámetro de referencia representa la variable especificada como argumento en la invocación del método o del constructor. Conceptualmente, el valor de un parámetro de referencia siempre es el mismo que el de la variable subyacente.

Los parámetros de referencia actúan de dos modos, como *alias* o mediante las *opciones de copia*.

**Alias.** Un parámetro de referencia se usa cuando el parámetro actúa como un alias para un argumento suministrado por el llamador. Un parámetro de referencia no define en sí mismo una variable, sino que hace referencia a la variable del argumento correspondiente. Las modificaciones de un parámetro de referencia influyen de manera directa e inmediata en el argumento correspondiente. En el ejemplo siguiente se muestra un método Swap que tiene dos parámetros de referencia:

Module Test  
 Sub Swap(ByRef a As Integer, ByRef b As Integer)  
 Dim t As Integer = a  
 a = b  
 b = t  
 End Sub   
  
 Sub Main()  
 Dim x As Integer = 1  
 Dim y As Integer = 2  
  
 Console.WriteLine("pre: x = " & x & ", y = " & y)  
 Swap(x, y)  
 Console.WriteLine("post: x = " & x & ", y = " & y)  
 End Sub   
End Module

El resultado del programa es:

pre: x = 1, y = 2  
post: x = 2, y = 1

Para la invocación del método Swap en la clase Main, a representa a x, y b representa a y. Por lo tanto, la invocación tiene el efecto de intercambiar los valores de x y de y.

En un método que toma parámetros de referencia, varios nombres pueden representar la misma ubicación de almacenamiento.

Module Test  
 Private s As String  
  
 Sub F(ByRef a As String, ByRef b As String)  
 s = "One"  
 a = "Two"  
 b = "Three"  
 End Sub  
  
 Sub G()  
 F(s, s)  
 End Sub  
End Module

En el ejemplo, la invocación del método 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 la variable de instancia s.

**Opciones de copia.** Si el tipo de la variable que se pasa a un parámetro de referencia no es compatible con el tipo del parámetro de referencia, o si un tipo de no variable (p. ej., una propiedad) se pasa como argumento a un parámetro de referencia, o si la invocación se enlaza en tiempo de ejecución, entonces se asigna y se pasa una variable temporal al parámetro de referencia. El valor que se pasa se copiará en esta variable temporal antes de invocar el método y se volverá a copiar en la variable original (si hay una y es editable) cuando el método vuelva. De este modo, un parámetro de referencia no tiene que contener necesariamente una referencia al almacenamiento exacto de la variable que se pasa y cualquier cambio en el parámetro de referencia tal vez no ser refleje en la variable hasta que el método exista. Por ejemplo:

Class Base  
End Class  
  
Class Derived  
 Inherits Base  
End Class  
  
Module Test  
 Sub F(ByRef b As Base)  
 b = New Base()  
 End Sub  
  
 Property G() As Base  
 Get  
 End Get  
 Set  
 End Set  
 End Property  
  
 Sub Main()  
 Dim d As Derived  
  
 F(G) ' OK.  
 F(d) ' Throws System.InvalidCastException after F returns.  
 End Sub  
End Module

En el caso de la primera invocación de F, se crea una variable temporal y se asigna y se pasa el valor de la propiedad G a F. Al volver de F, el valor de la variable temporal se vuelve a asignar a la propiedad de G. En el segundo caso, se crea otra variable temporal y se le asigna el valor d y se pasa a F. Cuando vuelve de F, el valor de la variable temporal se vuelve a convertir en el tipo de la variable, Derived, y se asigna a d. Como el valor que se vuelve a pasar no se puede convertir en Derived, se produce una excepción en tiempo de ejecución.

#### Parámetros opcionales

Los parámetros opcionales se declaran con el modificador Optional. Los parámetros que siguen a un parámetro opcional en la lista de parámetros formales deben ser también opcionales; si no se especifica el modificador Optional en los parámetros siguientes se producirá un error en tiempo de compilación. Un parámetro opcional de algún tipo *T?* que acepta valores NULL o tipo *T* que no acepta valores NULL, debe especificar una expresión constante *e* que se usará como valor predeterminado si no se especifica ningún argumento. Si *e* se evalúa como Nothing de tipo Object, el valor predeterminado del *tipo de parámetro* se empleará como el valor predeterminado del parámetro. De lo contrario, CType(e, T) debe ser una expresión constante y se toma como el valor predeterminado para el parámetro.

Los parámetros opcionales son la única situación en la que es válido un inicializador en un parámetro. La inicialización siempre se hace como parte de una expresión de invocación, no dentro del cuerpo del método.

Module Test  
 Sub F(x As Integer, Optional y As Integer = 20)  
 Console.WriteLine("x = " & x & ", y = " & y)  
 End Sub  
  
 Sub Main()  
 F(10)  
 F(30,40)  
 End Sub  
End Module

El resultado del programa es:

x = 10, y = 20  
x = 30, y = 40

No se pueden especificar parámetros opcionales en las declaraciones de eventos o de delegados ni en expresiones lambda.

#### Parámetros ParamArray

Los parámetros ParamArray se declaran con el modificador ParamArray. Si el modificador ParamArray está presente, se debe especificar el modificador ByVal y ningún otro parámetro puede usar el modificador ParamArray. El tipo del parámetro ParamArray debe ser una matriz unidimensional y debe ser el último parámetro de la lista.

Un parámetro ParamArray representa un número indeterminado de parámetros del tipo de ParamArray. Dentro del propio método, un parámetro ParamArray se trata como su tipo declarado y no tiene semántica especial. El parámetro ParamArray es implícitamente opcional, con un valor predeterminado de una matriz unidimensional vacía del tipo de ParamArray.

ParamArray permite especificar argumentos en cualquiera de las dos formas en una invocación de método:

El argumento especificado para ParamArray puede ser una única expresión de un tipo que se amplía al tipo ParamArray. En este caso, ParamArray actúa exactamente como un parámetro de valor.

Alternativamente, la invocación puede especificar ninguno o algún argumento para ParamArray, donde cada argumento es una expresión de un tipo implícitamente convertible en el tipo de elemento de ParamArray. En este caso, la invocación crea una instancia del tipo de ParamArray 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, ParamArray equivale exactamente a un parámetro de valor del mismo tipo, como se muestra en el ejemplo siguiente.

Module Test  
 Sub F(ParamArray args() As Integer)  
 Dim i As Integer  
  
 Console.Write("Array contains " & args.Length & " elements:")  
 For Each i In args  
 Console.Write(" " & i)  
 Next i  
 Console.WriteLine()  
 End Sub  
  
 Sub Main()  
 Dim a As Integer() = { 1, 2, 3 }  
  
 F(a)  
 F(10, 20, 30, 40)  
 F()  
 End Sub  
End Module

El ejemplo 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 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 de cero elementos y pasa esa instancia como parámetro de valor. La segunda y tercera invocación equivalen exactamente a escribir:

F(New Integer() {10, 20, 30, 40})  
F(New Integer() {})

Los parámetros ParamArray no se pueden especificar en las declaraciones de eventos o de delegados.

### Control de eventos

Los métodos pueden controlar declarativamente eventos generados por objetos en variables compartidas o de instancia. Para controlar eventos, una declaración de método especifica la palabra clave Handles y enumera uno o varios eventos. Un evento de la lista Handles es especificado por dos identificadores separados por un punto:

El primer identificador debe ser una variable compartida o de instancia en el tipo contenedor que especifica el modificador WithEvents o la palabra clave MyBase, MyClass o Me; de lo contrario, se produce un error en tiempo de compilación. Esta variable contiene el objeto que generará los eventos que controla este método.

El segundo identificador debe especificar un miembro del tipo del primer identificador. El miembro debe ser un evento y puede ser compartido. Si se especifica una variable compartida para el primer identificador, entonces el evento debe ser compartido o se producirá un error.

Un método de controlador M se considera un controlador de eventos válido para un evento E si la instrucción AddHandler E, AddressOf M fuera también válida. A diferencia de una instrucción AddHandler, sin embargo, los controladores de eventos explícitos permiten controlar un evento con un método sin argumentos, con independencia de que se use semántica estricta:

Option Strict On  
  
Class C1  
 Event E(x As Integer)  
End Class  
  
Class C2  
 withEvents C1 As New C1()  
  
 ' Valid  
 Sub M1() Handles C1.E  
 End Sub  
  
 Sub M2()  
 ' Invalid  
 AddHandler C1.E, AddressOf M1  
 End Sub  
End Class

Un miembro único puede controlar muchos eventos coincidentes y muchos métodos pueden controlar un único evento. La accesibilidad de un método no influye en su capacidad para controlar eventos. En el ejemplo siguiente se muestra cómo un método puede controlar eventos:

Class Raiser  
 Event E1()  
  
 Sub Raise()  
 RaiseEvent E1  
 End Sub  
End Class  
  
Module Test  
 WithEvents x As Raiser  
  
 Sub E1Handler() Handles x.E1  
 Console.WriteLine("Raised")  
 End Sub  
  
 Sub Main()  
 x = New Raiser()  
 x.Raise()  
 x.Raise()  
 End Sub  
End Module

Esto imprimirá:

Raised  
Raised

Un tipo hereda todos los controladores de eventos suministrados por su tipo base. Un tipo derivado no puede modificar en modo alguno las asignaciones de eventos que hereda de sus tipos base, pero puede agregar controladores adicionales al evento.

HandlesClause ::= [ Handles EventHandlesList ]

EventHandlesList ::=  
 EventMemberSpecifier |  
 EventHandlesList Comma EventMemberSpecifier

EventMemberSpecifier ::=  
 Identifier Period IdentifierOrKeyword |  
 MyBase Period IdentifierOrKeyword |  
 MyClass Period IdentifierOrKeyword |  
 Me Period IdentifierOrKeyword

### Métodos de extensión

Es posible agregar métodos a tipos desde fuera de la declaración de tipos mediante los métodos de extensión. Los métodos de extensión son métodos con el atributo System.Runtime.CompilerServices.ExtensionAttribute aplicado. Solo se pueden declarar en módulos estándar y deben tener al menos un parámetro que especifique el tipo que el método extiende. Por ejemplo, el siguiente método de extensión extiende el tipo String:

Imports System.Runtime.CompilerServices  
  
Module StringExtensions  
 <Extension> \_  
 Sub Print(s As String)  
 Console.WriteLine(s)  
 End Sub  
End Module

Anotación

Aunque Visual Basic requiere que los métodos de extensión sean declarados en un módulo estándar, otros lenguajes como C# permiten que se declaren en otros tipos. Siempre que los métodos sigan las demás convenciones aquí señaladas y que el tipo contenedor no sea un tipo genérico abierto, del que no se pueden crear instancias, Visual Basic reconocerá los métodos de extensión.

Cuando se invoca un método de extensión, la instancia en la que se invoca se pasa al primer parámetro. El primer parámetro no se puede declarar Optional ni ParamArray. Cualquier tipo, incluido un parámetro de tipo, puede aparecer como primer parámetro de un método de extensión. Por ejemplo, el método siguiente extiende los tipos Integer(), cualquier tipo que implemente System.Collections.Generic.IEnumerable(Of T) y cualquier otro tipo:

Imports System.Runtime.CompilerServices  
  
Module Extensions  
 <Extension> \_  
 Sub PrintArray(a() As Integer)  
 ...  
 End Sub  
  
 <Extension> \_  
 Sub PrintList(Of T)(a As IEnumerable(Of T))  
 ...  
 End Sub  
  
 <Extension> \_  
 Sub Print(Of T)(a As T)  
 ...  
 End Sub  
End Module

Como se muestra en el ejemplo anterior, es posible extender las interfaces. Los métodos de extensión de interfaces proporcionan la implementación del método, de manera que los tipos que implementan una interfaz que tiene métodos de extensión definidos siguen implementando únicamente los miembros declarados en principio por la interfaz. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Interface IAction  
 Sub DoAction()  
End Interface  
  
Module IActionExtensions   
 <Extension> \_  
 Public Sub DoAnotherAction(i As IAction)   
 i.DoAction()  
 End Sub  
End Module  
  
Class C  
 Implements IAction  
  
 Sub DoAction() Implements IAction.DoAction  
 ...  
 End Sub  
  
 ' ERROR: Cannot implement extension method IAction.DoAnotherAction  
 Sub DoAnotherAction() Implements IAction.DoAnotherAction  
 ...  
 End Sub  
End Class

Los métodos de extensión también pueden tener restricciones de tipo en los parámetros de tipo y, al igual que con los métodos genéricos de no extensión, se pueden inferir los argumentos de tipo:

Imports System.Runtime.CompilerServices  
  
Module IEnumerableComparableExtensions  
 <Extension> \_  
 Public Function Sort(Of T As IComparable(Of T))(i As IEnumerable(Of T)) \_  
 As IEnumerable(Of T)   
 ...  
 End Function  
End Module

A los métodos de extensión también se puede tener acceso a través de expresiones de instancias implícitas dentro del tipo que se está extendiendo:

Imports System.Runtime.CompilerServices  
  
Class C1  
 Sub M1()  
 Me.M2()  
 M2()  
 End Sub  
End Class  
  
Module C1Extensions  
 <Extension> \_  
 Sub M2(c As C1)  
 ...  
 End Sub  
End Module

Por motivos de accesibilidad, los métodos de extensión también se tratan como miembros del módulo estándar en el que se declaran: no tienen acceso extra a los miembros del tipo que están extendiendo más allá del acceso que tienen en virtud de su contexto de declaración.

Los métodos de extensiones solo están disponibles cuando el método del módulo estándar está en el ámbito. De lo contrario, el tipo original no parecerá haber sido extendido. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Class C1  
End Class  
  
Namespace N1  
 Module C1Extensions  
 <Extension> \_  
 Sub M1(c As C1)  
 ...  
 End Sub  
 End Module  
End Namespace  
  
Module Test  
 Sub Main()  
 Dim c As New C1()  
  
 ' Error: c has no member named "M1"  
 c.M1()  
 End Sub  
End Module

Hacer referencia a un tipo cuando solo hay un método de extensión disponible en el tipo seguirá produciendo un error en tiempo de compilación.

Es importante observar que los métodos de extensión se consideran miembros del tipo en todos los contextos en los que están enlazados los miembros, como el patrón For Each fuertemente tipado. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Class C1  
End Class  
  
Class C1Enumerator  
 ReadOnly Property Current() As C1  
 Get  
 ...  
 End Get  
 End Property  
  
 Function MoveNext() As Boolean  
 ...  
 End Function  
End Class  
  
Module C1Extensions  
 <Extension> \_  
 Function GetEnumerator(c As C1) As C1Enumerator  
 ...  
 End Function  
End Module  
  
Module Test  
 Sub Main()  
 Dim c As New C1()  
  
 ' Valid  
 For Each o As Object In c  
 ...  
 Next o  
 End Sub  
End Module

También se pueden crear delegados que hagan referencia a métodos de extensión. Por consiguiente, el código:

Delegate Sub D1()  
  
Module Test  
 Sub Main()  
 Dim s As String = "Hello, World!"  
 Dim d As D1  
  
 d = AddressOf s.Print  
 d()  
 End Sub  
End Module

es más o menos equivalente a:

Delegate Sub D1()  
  
Module Test  
 Sub Main()  
 Dim s As String = "Hello, World!"  
 Dim d As D1  
  
 d = CType([Delegate].CreateDelegate(GetType(D1), s, \_  
 GetType(StringExtensions).GetMethod("Print")), D1)  
 d()  
 End Sub  
End Module

Anotación

Por lo general, Visual Basic inserta una marca en una llamada a método de instancia que produce una excepción System.NullReferenceException si la instancia en la que se invoca el método es Nothing. En el caso de los métodos de extensión, no hay un modo eficaz de insertar esta marca, de forma que los métodos de extensión tendrán que buscar explícitamente Nothing.

Además, se aplicará una conversión boxing a un tipo de valor cuando se pase como argumento ByVal a un parámetro cuyo tipo sea una interfaz. Esto significa que los efectos secundarios del método de extensión operarán en una copia de la estructura en lugar de en el original. Aunque el lenguaje no pone restricciones en el primer argumento de un método de extensión, se recomienda que no se usen los métodos de extensión para extender tipos de valor o cuando al extender los tipos de valor, el primer parámetro se pasa ByRef para asegurar que los efectos secundarios actúen en el valor original.

### Métodos parciales

Un método parcial es un método que especifica la signatura pero no el cuerpo completo del método. El cuerpo del método puede proporcionarlo otra declaración del método con el mismo nombre y firma, probablemente en otra declaración parcial del tipo. Por ejemplo:

a.vb:

' Designer generated code  
Public Partial Class MyForm  
 Private Partial Sub ValidateControls()  
 End Sub  
  
 Public Sub New()  
 ' Initialize controls  
 ...  
  
 ValidateControls()  
 End Sub   
End Class

b.vb:

Public Partial Class MyForm  
 Public Sub ValidateControls()  
 ' Validation logic goes here  
 ...  
 End Sub  
End Class

En este ejemplo, una declaración parcial de la clase MyForm declara un método parcial ValidateControls sin implementación. El constructor de la declaración parcial llama al método parcial, aunque no se proporcione cuerpo en el archivo. La otra declaración parcial de MyForm proporciona entonces la implementación del método.

Se puede llamar a métodos parciales con independencia de que se haya proporcionado un cuerpo; si no se proporciona, la llamada se ignora. Por ejemplo:

Public Class C1  
 Private Partial Sub M1()  
 End Sub  
  
 Public Sub New()  
 ' Since no implementation is supplied, this call will not be made.  
 M1()  
 End Sub  
End Class

Todas las expresiones que se pasan como argumentos a una llamada a un método parcial que se ignora también se ignoran y no se evalúan.

Anotación

Esto significa que los métodos parciales constituyen un modo muy eficaz de proporcionar este comportamiento que se define entre dos tipos parciales, ya que los métodos parciales no tienen ningún costo si no se utilizan.

La declaración de métodos parciales se debe declarar Private y siempre debe ser una subrutina sin ninguna instrucción en el cuerpo. Los métodos parciales no pueden implementar métodos de interfaz, pero sí puede hacerlo el método que proporciona el cuerpo.

Solo un método puede proporcionar un cuerpo a un método parcial. Un método que proporciona un cuerpo a un método parcial debe tener la misma firma, las mismas restricciones en los parámetros de tipo, los mismos modificadores de declaración y los mismos parámetros y nombres de parámetros de tipo que el método parcial. Los atributos del método parcial y el método que proporciona su cuerpo se combinan, al igual que los atributos de los parámetros de los métodos. Del mismo modo se combina la lista de los eventos que los métodos controlan. Por ejemplo:

Class C1  
 Event E1()  
 Event E2()  
  
 Private Partial Sub S() Handles Me.E1  
 End Sub  
  
 ' Handles both E1 and E2  
 Private Sub S() Handles Me.E2  
 ...  
 End Sub  
End Class

## Constructores

Los constructores son métodos especiales que permiten controlar la inicialización. Se ejecutan después de que se inicie el programa o cuando se crea una instancia de un tipo. A diferencia de otros miembros, los constructores no se heredan y no introducen un nombre en el espacio de declaración del tipo. Los constructores solo pueden invocarse mediante expresiones de creación de objetos o por .NET Framework; tal vez nunca se les invoque directamente.

Nota Los constructores tienen la misma restricción sobre la ubicación en las líneas que las subrutinas. La primera y la última instrucción, y el bloque deben aparecen al principio de una línea lógica.

ConstructorMemberDeclaration ::=  
 [ Attributes ] [ ConstructorModifier+ ] Sub New  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ] LineTerminator  
 [ Block ]  
 End Sub StatementTerminator

ConstructorModifier ::= AccessModifier | Shared

### Constructores de instancias

Los constructores de instancias inicializan las instancias de un tipo y las ejecuta .NET Framework cuando se crea una instancia. La lista de parámetros de un constructor está sujeta a las mismas reglas que la lista de parámetros de un método. Los constructores de instancias pueden estar sobrecargados.

Todos los constructores de los tipos de referencia deben invocar otro constructor. Si la invocación es explícita, debe ser la primera instrucción del cuerpo del método del constructor. La instrucción puede invocar otro de los constructores de instancias del tipo, por ejemplo, Me.New(...) o MyClass.New(...) o, si no es una estructura, puede llamar a un constructor de instancia del tipo base del tipo, por ejemplo, MyBase.New(...). No es válido que un constructor se llame a sí mismo. Si un constructor omite una llamada a otro, MyBase.New() está implícito. Si no hay un constructor del tipo base sin parámetros, se produce un error en tiempo de compilación. Como Me no se considera que se construya hasta después de la llamada a un constructor de clase base, los parámetros de una instrucción de llamada a un constructor no pueden hacer referencia a Me, MyClass o MyBase implícita o explícitamente.

Cuando la primera instrucción de un constructor está en la forma MyBase.New(...), el constructor lleva a cabo implícitamente la inicialización especificada por los inicializadores de variable de instancia declarados en el tipo. Esto corresponde a una secuencia de asignaciones que se ejecutan inmediatamente después de llamar al constructor directo del tipo base. Tal ordenamiento garantiza que todas las variables de instancia se inicialicen por sus inicializadores de variable antes de que se ejecute cualquier instrucción que tenga acceso a la instancia. Por ejemplo:

Class A  
 Protected x As Integer = 1  
End Class  
  
Class B  
 Inherits A  
  
 Private y As Integer = x  
  
 Public Sub New()  
 Console.WriteLine("x = " & x & ", y = " & y)  
 End Sub  
End Class

Cuando se utiliza New B() para crear una instancia de B, el resultado que se produce es:

x = 1, y = 1

El valor de y es 1 porque el inicializador de variable se ejecuta después de que se invoque el constructor de la clase base. Los inicializadores de variable se ejecutan en el orden en que aparecen en la declaración de tipos.

Cuando un tipo declara únicamente constructores Private, no es posible en general que otros tipos deriven ni creen instancias del tipo; la única excepción son los tipos anidados dentro del tipo. Los constructores Private se usan normalmente en tipos que solo contienen miembros Shared.

Si un tipo no contiene ninguna declaración de constructor de instancias, se proporciona automáticamente un constructor de instancias predeterminado. El constructor predeterminado simplemente invoca el constructor sin parámetros del tipo base directo. Si el tipo base directo no tiene un constructor sin parámetros accesible, se producirá un error en tiempo de compilación. El tipo de acceso declarado para el constructor predeterminado es Public salvo que el tipo sea MustInherit, en cuyo caso el constructor predeterminado es Protected.

Anotación

El acceso predeterminado de un constructor predeterminado de tipo MustInherit es Protected porque las clases MustInherit no se pueden crear directamente. Por tanto, no tiene sentido hacer de Public el constructor predeterminado.

En el ejemplo siguiente se proporciona un constructor predeterminado porque la clase no contiene ninguna declaración de constructor:

Class Message  
 Dim sender As Object  
 Dim text As String  
End Class

Así, el ejemplo es exactamente equivalente a lo siguiente:

Class Message  
 Dim sender As Object  
 Dim text As String  
  
 Sub New()  
 End Sub  
End Class

Los constructores predeterminados que se emiten en una clase generada del diseñador marcada con el atributo Microsoft.VisualBasic.CompilerServices.DesignerGeneratedAttribute llamarán al método Sub InitializeComponent(), si existe, después de la llamada al constructor base.

Anotación

Esto permite que los archivos generados por el diseñador, como los creados por el diseñador WinForms, omitan el constructor en el archivo generado. Esto permite al programador especificarlo si así lo quiere.

### Constructores compartidos

Los constructores compartidos inicializan las variables compartidas de un tipo; se ejecutan después de que el programa comience a ejecutarse, pero antes de cualquier referencia a un miembro del tipo. Un constructor compartido especifica el modificador Shared, salvo que sea un módulo estándar, en cuyo caso el modificador Shared está implícito.

A diferencia de los constructores de instancia, los constructores compartidos tienen acceso público implícito, no tienen parámetros y no pueden llamar a otros constructores. Antes de su primera instrucción, el constructor compartido lleva a cabo implícitamente la inicialización especificada por los inicializadores de variable de las variables compartidas declaradas en el tipo. Esto corresponde a una secuencia de asignaciones que se ejecutan inmediatamente después de entrar en el constructor. Los inicializadores de variable se ejecutan en el orden en que aparecen en la declaración de tipos.

En el ejemplo siguiente se muestra una clase Employee con un constructor compartido que inicializa una variable compartida:

Imports System.Data  
  
Class Employee  
 Private Shared ds As DataSet  
  
 Shared Sub New()  
 ds = New DataSet()  
 End Sub  
  
 Public Name As String  
 Public Salary As Decimal  
End Class

Un constructor compartido separado existe para cada tipo genérico cerrado. Puesto que el constructor compartido se ejecuta exactamente una vez para cada tipo cerrado, resulta un lugar conveniente para exigir las comprobaciones en tiempo de ejecución en el parámetro de tipo que no se pueden comprobar en tiempo de compilación mediante restricciones. Por ejemplo, el siguiente tipo utiliza un constructor compartido para exigir que el parámetro de tipo sea Integer o Double:

Class EnumHolder(Of T)  
 Shared Sub New()   
 If Not GetType(T).IsEnum() Then  
 Throw New ArgumentException("T must be an enumerated type.")  
 End If  
 End Sub  
End Class

Exactamente cuándo se ejecutan los constructores compartidos depende de la implementación, aunque se ofrecen varias garantías si se define de forma explícita un constructor compartido:

Los constructores compartidos se ejecutan antes del primer acceso a cualquier campo estático del tipo.

Los constructores compartidos se ejecutan antes de la primera invocación de cualquier método estático del tipo.

Los constructores compartidos se ejecutan antes de la primera invocación de cualquier constructor del tipo.

Las garantías anteriores no se aplican cuando un constructor compartido se crea de manera implícita para los inicializadores compartidos. El resultado del ejemplo siguiente es incierto, porque el orden exacto de la carga y, por consiguiente, de la ejecución del constructor compartido no está definido:

Module Test  
 Sub Main()  
 A.F()  
 B.F()  
 End Sub  
End Module  
  
Class A  
 Shared Sub New()  
 Console.WriteLine("Init A")  
 End Sub  
  
 Public Shared Sub F()  
 Console.WriteLine("A.F")  
 End Sub  
End Class  
  
Class B  
 Shared Sub New()  
 Console.WriteLine("Init B")  
 End Sub  
  
 Public Shared Sub F()  
 Console.WriteLine("B.F")  
 End Sub  
End Class

El resultado podría ser uno de los siguientes:

Init A  
A.F  
Init B  
B.F

o

Init B  
Init A  
A.F  
B.F

En contraste, en el ejemplo siguiente se muestra un resultado predecible. Observe que el constructor Shared de la clase A nunca se ejecuta, aunque la clase B derive de ella:

Module Test  
 Sub Main()  
 B.G()  
 End Sub  
End Module  
  
Class A  
 Shared Sub New()  
 Console.WriteLine("Init A")  
 End Sub  
End Class  
  
Class B  
 Inherits A  
  
 Shared Sub New()  
 Console.WriteLine("Init B")  
 End Sub  
  
 Public Shared Sub G()  
 Console.WriteLine("B.G")  
 End Sub  
End Class

El resultado es:

Init B  
B.G

También es posible construir dependencias circulares que permitan a las variables Shared con inicializadores de variable ser observadas en su estado de valor predeterminado, como en el ejemplo siguiente:

Class A  
 Public Shared X As Integer = B.Y + 1  
End Class  
  
Class B  
 Public Shared Y As Integer = A.X + 1  
  
 Shared Sub Main()  
 Console.WriteLine("X = " & A.X & ", Y = " & B.Y)  
 End Sub  
End Class

Se genera el siguiente resultado:

X = 1, Y = 2

Para ejecutar el método Main, el sistema carga primero la clase B. El constructor Shared de la clase B pasa a calcular el valor inicial de Y, que de manera recursiva hace que la clase A se cargue porque hay una referencia al valor de A.X. El constructor Shared de la clase A a su vez pasa a calcular el valor inicial de X y, al hacerlo, trae el valor predeterminado de Y, que es cero. A.X por tanto se inicializa en 1. El proceso de carga A después finaliza y vuelve al cálculo del valor inicial de Y, cuyo resultado es 2.

Si el método Main hubiera estado en la clase A, el ejemplo habría arrojado el resultado siguiente:

X = 2, Y = 1

Conviene evitar las referencias circulares en inicializadores de variable Shared, ya que suele ser imposible determinar el orden en que se cargan las clases que contienen las referencias.

## Eventos

Los eventos se usan para notificar código de una aparición concreta. Una declaración de evento consiste en un identificador, bien un tipo de delegado o una lista de parámetros, y una cláusula Implements opcional. Si se especifica un tipo de delegado, éste no puede tener un tipo devuelto. Si se especifica una lista de parámetros, no puede contener los parámetros Optional o ParamArray. El dominio de accesibilidad de los tipos de parámetros y/o los tipos de parámetro debe ser el mismo o un superconjunto del dominio de accesibilidad del propio evento. Los eventos se pueden compartir si se especifica el modificador Shared.

Además del nombre del miembro agregado al espacio de declaración del tipo, una declaración de eventos declara implícitamente otros miembros. Dado un evento denominado X, los miembros siguientes se agregan al espacio de declaración:

Si la declaración está en forma de declaración de método, se introduce una clase de delegado anidada denominada XEventHandler. La clase de delegado anidada coincide con la declaración de método y tiene la misma accesibilidad que el evento. Los atributos de la lista de atributos se aplican a los parámetros de la clase delegada.

Una variable de instancia Private con el tipo del delegado, denominada XEvent.

Dos métodos denominados add\_X y remove\_X que no se pueden invocar, invalidar o sobrecargar.

Si un tipo intenta declarar un nombre que coincide con uno de los nombres anteriores, se producirá un error en tiempo de compilación y las declaraciones implícitas add\_X y remove\_X se ignoran por motivos de enlace de nombres. No es posible invalidar ni sobrecargar ninguno de los miembros introducidos, aunque es posible ocultarlos en tipos derivados. Por ejemplo, la declaración de clase

Class Raiser  
 Public Event Constructed(i As Integer)  
End Class

equivale a la siguiente declaración

Class Raiser  
 Public Delegate Sub ConstructedEventHandler(i As Integer)  
  
 Protected ConstructedEvent As ConstructedEventHandler  
  
 Public Sub add\_Constructed(d As ConstructedEventHandler)  
 ConstructedEvent = \_  
 CType( \_  
 [Delegate].Combine(ConstructedEvent, d), \_  
 Raiser.ConstructedEventHandler)  
 End Sub  
  
 Public Sub remove\_Constructed(d As ConstructedEventHandler)  
 ConstructedEvent = \_  
 CType( \_  
 [Delegate].Remove(ConstructedEvent, d), \_  
 Raiser.ConstructedEventHandler)  
 End Sub  
End Class

Declarar un evento sin especificar un tipo de delegado es la sintaxis más sencilla y compacta, pero presenta el inconveniente de declarar un nuevo tipo de delegado para cada evento. Por ejemplo, en el ejemplo siguiente se crean tres tipos de delegados ocultos, aunque los tres tengan la misma lista de parámetros:

Public Class Button  
 Public Event Click(sender As Object, e As EventArgs)  
 Public Event DoubleClick(sender As Object, e As EventArgs)  
 Public Event RightClick(sender As Object, e As EventArgs)  
End Class

En el ejemplo siguiente, los eventos simplemente usan el mismo delegado, EventHandler:

Public Delegate Sub EventHandler(sender As Object, e As EventArgs)  
  
Public Class Button  
 Public Event Click As EventHandler  
 Public Event DoubleClick As EventHandler  
 Public Event RightClick As EventHandler  
End Class

Los eventos se puede controlar de dos maneras: estática o dinámicamente. El control estático de los eventos es más sencillo y solo requiere una variable WithEvents y una cláusula Handles. En el ejemplo siguiente, la clase Form1 controla estáticamente el evento Click del objeto Button:

Public Class Form1  
 Public WithEvents Button1 As New Button()  
  
 Public Sub Button1\_Click(sender As Object, e As EventArgs) \_  
 Handles Button1.Click  
 Console.WriteLine("Button1 was clicked!")  
 End Sub  
End Class

El control dinámico de los eventos es más complicado porque el evento debe estar explícitamente conectado y desconectado en el código. La instrucción AddHandler agrega un controlador para un evento y la instrucción RemoveHandler quita el controlador del evento. En el ejemplo siguiente se muestra una clase Form1 que agrega Button1\_Click como un controlador de eventos para el evento Click de Button1.

Public Class Form1  
 Public Sub New()  
 ' Add Button1\_Click as an event handler for Button1's Click event.  
 AddHandler Button1.Click, AddressOf Button1\_Click  
 End Sub   
  
 Private Button1 As Button = New Button()  
  
 Sub Button1\_Click(sender As Object, e As EventArgs)  
 Console.WriteLine("Button1 was clicked!")  
 End Sub  
  
 Public Sub Disconnect()  
 RemoveHandler Button1.Click, AddressOf Button1\_Click  
 End Sub   
End Class

En el método Disconnect, se quita el controlador de eventos.

EventMemberDeclaration ::=  
 RegularEventMemberDeclaration |  
 CustomEventMemberDeclaration

RegularEventMemberDeclaration ::=  
 [ Attributes ] [ EventModifiers+ ] Event Identifier ParametersOrType [ ImplementsClause ]  
 StatementTerminator

InterfaceEventMemberDeclaration ::=  
 [ Attributes ] [ InterfaceEventModifiers+ ] Event Identifier ParametersOrType StatementTerminator

ParametersOrType ::=  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ] |  
 As NonArrayTypeName

EventModifiers ::= AccessModifier | Shadows | Shared

InterfaceEventModifiers ::= Shadows

### Eventos personalizados

Como se ha visto en la sección anterior, las declaraciones de eventos definen implícitamente un campo, un método add\_ y un método remove\_ que se emplean para hacer seguimiento de los controladores de eventos. En algunas situaciones, sin embargo, puede ser deseable proporcionar código personalizado para el seguimiento de los controladores de eventos. Por ejemplo, si la clase define cuarenta eventos de los cuales solo se controlarán algunos, el uso de una tabla hash en lugar de cuarenta campos para el seguimiento de los controladores de cada evento puede ser más eficiente. Los eventos personalizados permiten definir los métodos add\_X y remove\_X explícitamente, lo que permite el almacenamiento personalizado de los controladores de eventos.

Los eventos personalizados se declaran igual que los eventos que especifican un tipo de delegado, con la excepción de que la palabra clave Custom debe preceder a la palabra clave Event. Una declaración de evento personalizado contiene tres declaraciones: AddHandler, RemoveHandler y RaiseEvent. Ninguna de ellas puede tener modificadores, aunque pueden tener atributos. Por ejemplo:

Class Test  
 Private Handlers As EventHandler  
  
 Public Custom Event TestEvent As EventHandler  
 AddHandler(value As EventHandler)  
 Handlers = CType([Delegate].Combine(Handlers, value), \_  
 EventHandler)  
 End AddHandler  
  
 RemoveHandler(value as EventHandler)  
 Handlers = CType([Delegate].Remove(Handlers, value), \_  
 EventHandler)  
 End RemoveHandler  
  
 RaiseEvent(sender As Object, e As EventArgs)  
 Dim TempHandlers As EventHandler = Handlers  
  
 If TempHandlers IsNot Nothing Then  
 TempHandlers(sender, e)  
 End If  
 End RaiseEvent  
 End Event  
End Class

Las declaraciones AddHandler y RemoveHandler toman un parámetro ByVal, que debe ser del tipo delegado del evento. Cuando se ejecuta una instrucción AddHandler o RemoveHandler (o una cláusula Handles controla automáticamente un evento), se llamará a la declaración correspondiente. La declaración de RaiseEvent toma los mismos parámetros que el delegado del evento y se le llamará cuando se ejecute una instrucción RaiseEvent. Se deben proporcionar todas las declaraciones y se consideran subrutinas.

Nota AddHandler, RemoveHandler y RaiseEvent tienen la misma restricción en la ubicación en las líneas que las subrutinas. La primera y la última instrucción, y el bloque deben aparecen al principio de una línea lógica.

Anotación

En Microsoft Visual Basic 11.0, los eventos declarados en un archivo compilado con /target:winmdobj, o declarados en una interfaz en un archivo de este tipo y, a continuación, implementados en otro lugar, se tratan de una forma algo distinta.

1. Las herramientas externas usadas para compilar el archivo winmd normalmente solo permitirán determinados tipos delegados, como System.EventHandler(Of T) o System.TypedEventHandle(Of T, U), y no permitirán otros.

2. El campo XEvent es de tipo System.Runtime.InteropServices.WindowsRuntime.EventRegistrationTokenTable(Of T), donde T es el tipo delegado.

3. El descriptor de acceso AddHandler devuelve System.Runtime.InteropServices.WindowsRuntime.EventRegistrationToken y el descriptor de acceso RemoveHandler toma un único parámetro del mismo tipo.

A continuación se muestra un ejemplo de dicho evento personalizado.

Imports System.Runtime.InteropServices.WindowsRuntime

Public NotInheritable Class ClassInWinMD  
 Private XEvent As EventRegistrationTokenTable(Of EventHandler(Of Integer))

Public Custom Event X As EventHandler(Of Integer)  
 AddHandler(handler As EventHandler(Of Integer))  
 Return EventRegistrationTokenTable(Of EventHandler(Of Integer)).  
 GetOrCreateEventRegistrationTokenTable(XEvent).  
 AddEventHandler(handler)  
 End AddHandler

RemoveHandler(token As EventRegistrationToken)  
 EventRegistrationTokenTable(Of EventHandler(Of Integer)).  
 GetOrCreateEventRegistrationTokenTable(XEvent).  
 RemoveEventHandler(token)  
 End RemoveHandler

RaiseEvent(sender As Object, i As Integer)  
 Dim table = EventRegistrationTokenTable(Of EventHandler(Of Integer)).  
 GetOrCreateEventRegistrationTokenTable(XEvent).  
 InvocationList  
 If table IsNot Nothing Then table(sender, i)  
 End RaiseEvent  
 End Event  
End ClassAdemás del nombre del miembro agregado al espacio de declaración del tipo, una declaración de eventos personalizada declara implícitamente otros miembros. Dado un evento denominado X, los miembros siguientes se agregan al espacio de declaración:

Un método denominado add\_X, que corresponde a la declaración de AddHandler.

Un método denominado remove\_X, que corresponde a la declaración de RemoveHandler.

Un método denominado fire\_X, que corresponde a la declaración de RaiseEvent.

Si un tipo intenta declarar un nombre que coincide con uno de los nombres anteriores, se producirá un error en tiempo de compilación y las declaraciones implícitas se ignoran por motivos de enlace de nombres. No es posible invalidar ni sobrecargar ninguno de los miembros introducidos, aunque es posible ocultarlos en tipos derivados.

Nota Custom no es una palabra reservada.

CustomEventMemberDeclaration ::=  
 [ Attributes ] [ EventModifiers+ ] Custom Event Identifier As TypeName [ ImplementsClause ]  
 StatementTerminator  
 EventAccessorDeclaration+  
 End Event StatementTerminator

EventAccessorDeclaration ::=  
 AddHandlerDeclaration |  
 RemoveHandlerDeclaration |  
 RaiseEventDeclaration

AddHandlerDeclaration ::=  
 [ Attributes ] AddHandler OpenParenthesis ParameterList CloseParenthesis LineTerminator  
 [ Block ]  
 End AddHandler StatementTerminator

RemoveHandlerDeclaration ::=  
 [ Attributes ] RemoveHandler OpenParenthesis ParameterList CloseParenthesis LineTerminator  
 [ Block ]  
 End RemoveHandler StatementTerminator

RaiseEventDeclaration ::=  
 [ Attributes ] RaiseEvent OpenParenthesis ParameterList CloseParenthesis LineTerminator  
 [ Block ]  
 End RaiseEvent StatementTerminator

## Constantes

Una constante es un valor constante que es un miembro de un tipo. Las constantes son implícitamente compartidas. Si la declaración contiene una cláusula As, la cláusula especifica el tipo del miembro que se incluye en la declaración. Si el tipo se omite, el tipo de la constante se infiere. El tipo de una constante solo puede ser un tipo primitivo u Object. Si una constante es de tipo Object y no hay otro carácter de tipo, el tipo real de la constante será el tipo de la expresión constante. De lo contrario, el tipo de la constante es el tipo del carácter de tipo de la constante.

En el ejemplo siguiente se muestra una clase denominada Constants que tiene dos constantes públicas:

Class Constants  
 Public Const A As Integer = 1  
 Public Const B As Integer = A + 1  
End Class

A las constantes se tiene acceso a través de la clase, como en el ejemplo siguiente, que imprime los valores de Constants.A y Constants.B.

Module Test  
 Sub Main()  
 Console.WriteLine(Constants.A & ", " & Constants.B)  
 End Sub   
End Module

Una declaración de constante que declara varias constantes equivale a varias declaraciones de una sola constante. En el ejemplo siguiente se declaran tres constantes en una instrucción de declaración.

Class A  
 Protected Const x As Integer = 1, y As Long = 2, z As Short = 3  
End Class

Esta declaración equivale a lo siguiente:

Class A  
 Protected Const x As Integer = 1  
 Protected Const y As Long = 2  
 Protected Const z As Short = 3  
End Class

El dominio de accesibilidad del tipo de la constante debe ser el mismo o un superconjunto del dominio de accesibilidad de la propia constante. La expresión constante debe dar un valor del tipo de la constante o de un tipo que se pueda convertir implícitamente en el tipo de la constante. La expresión constante no puede ser circular, es decir, no se puede definir con sus mismos términos.

El compilador evalúa automáticamente las declaraciones de constantes en el orden apropiado. En el ejemplo siguiente, el compilador evalúa primero Y, después Z y finalmente X, y da los valores 10, 11 y 12, respectivamente.

Class A  
 Public Const X As Integer = B.Z + 1  
 Public Const Y As Integer = 10  
End Class  
  
Class B  
 Public Const Z As Integer = A.Y + 1  
End Class

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, en su lugar puede utilizarse una variable de solo lectura.

ConstantMemberDeclaration ::=  
 [ Attributes ] [ ConstantModifier+ ] Const ConstantDeclarators StatementTerminator

ConstantModifier ::= AccessModifier | Shadows

ConstantDeclarators ::=  
 ConstantDeclarator |  
 ConstantDeclarators Comma ConstantDeclarator

ConstantDeclarator ::= Identifier [ As TypeName ] Equals ConstantExpression StatementTerminator

## Variables compartidas y de instancias

Una variable compartida o de instancia es un miembro de un tipo que puede almacenar información. El modificador Dim se debe especificar si no se han especificado otros modificadores; si no, puede omitirse. Una declaración de variable única puede incluir varios declaradores de variable; cada declarador presenta una nueva instancia o miembro compartido.

Si se especifica un inicializador, solo se puede declarar una instancia o variable compartida en el declarador de variable:

Class Test  
 Dim a, b, c, d As Integer = 10 ' Invalid: multiple initialization  
End Class

Esta restricción no se aplica a los inicializadores de objetos.

Class Test  
 Dim a, b, c, d As New Collection() ' OK  
End Class

Una variable declarada con el modificador Shared es una variable compartida. Una variable compartida identifica exactamente una ubicación de almacenamiento con independencia del número de instancias del tipo que se creen. Una variable compartida comienza a existir cuando un programa empieza a ejecutarse y deja de hacerlo cuando el programa finaliza.

Una variable compartida solo se comparte entre las instancias de un tipo genérico cerrado concreto. Por ejemplo, el programa:

Class C(Of V)   
 Shared InstanceCount As Integer = 0  
  
 Public Sub New()   
 InstanceCount += 1   
 End Sub  
   
 Public Shared ReadOnly Property Count() As Integer   
 Get  
 Return InstanceCount  
 End Get  
 End Property  
End Class  
  
Class Application   
 Shared Sub Main()   
 Dim x1 As New C(Of Integer)()  
 Console.WriteLine(C(Of Integer).Count)  
  
 Dim x2 As New C(Of Double)()   
 Console.WriteLine(C(Of Integer).Count)  
  
 Dim x3 As New C(Of Integer)()   
 Console.WriteLine(C(Of Integer).Count)  
 End Sub  
End Class

Imprime:

1  
1  
2

Una variable declarada sin el modificador Shared se denomina variable de instancia. Todas las instancias de una clase contienen una copia independiente de todas las variables de instancia de la clase. Una variable de instancia de un tipo de referencia se genera cuando se crea una nueva instancia de ese tipo, y deja de existir cuando no hay referencias a esa instancia y se ha ejecutado el método Finalize. Una variable de instancia de un tipo de valor tiene exactamente el mismo período de duración que la variable a la que pertenece. Es decir, cuando se genera o deja de existir una variable de tipo de valor, le ocurre lo mismo a la variable de instancia del tipo de valor.

Si el declarador contiene una cláusula As, la cláusula especifica el tipo de los miembros que se incluye en la declaración. Si el tipo se omite y se usa semántica estricta, se produce un error en tiempo de compilación. De lo contrario, el tipo de los miembros es implícitamente Object o el tipo del carácter de tipo del método.

Nota   No hay ambigüedad en la sintaxis: si un declarador omite un tipo, siempre usará el tipo de un declarador siguiente.

El dominio de accesibilidad de un tipo de variable compartida o de instancia o el tipo de elemento de matriz debe ser el mismo o un superconjunto del dominio de accesibilidad de la propia variable compartida o de instancia.

En el ejemplo siguiente se muestra una clase Color que tiene variables de instancias internas denominadas redPart, greenPart y bluePart:

Class Color  
 Friend redPart As Short  
 Friend bluePart As Short  
 Friend greenPart As Short  
  
 Public Sub New(red As Short, blue As Short, green As Short)  
 redPart = red  
 bluePart = blue  
 greenPart = green  
 End Sub  
End Class

VariableMemberDeclaration ::=  
 [ Attributes ] VariableModifier+ VariableDeclarators StatementTerminator

VariableModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 ReadOnly |  
 WithEvents |  
 Dim

VariableDeclarators ::=  
 VariableDeclarator |  
 VariableDeclarators Comma VariableDeclarator

VariableDeclarator ::=  
 VariableIdentifiers As ObjectCreationExpression |  
 VariableIdentifiers [ As TypeName ] [ Equals Expression ]

VariableIdentifiers ::=  
 VariableIdentifier |  
 VariableIdentifiers Comma VariableIdentifier

VariableIdentifier ::= Identifier IdentifierModifiers

### Variables de solo lectura

Cuando una declaración de variable compartida o de instancia incluye un modificador ReadOnly, las asignaciones a las variables que aparecen en la declaración solo pueden tener lugar como parte de la declaración o en un constructor de la misma clase. Concretamente, las asignaciones a una instancia de solo lectura o una variable compartida solo se permiten en las situaciones siguientes:

En la declaración de variable que introduce la variable compartida o de instancia (mediante la inclusión de un inicializador de variable en la declaración).

Para una variable de instancia, en los constructores de instancia de la clase que contiene la declaración de variable. A la variable de instancia solo se puede tener acceso de un modo incompleto o a través de Me o MyClass.

Para una variable compartida, en el constructor compartido de la clase que contiene la declaración de variable compartida.

Una variable de solo lectura compartida es útil 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 lo puede calcular una expresión constante en tiempo de compilación.

A continuación se ofrece un ejemplo de la primera aplicación, donde variables compartidas de color se declaran ReadOnly para impedir que otros programas las cambien:

Class Color  
 Friend redPart As Short  
 Friend bluePart As Short  
 Friend greenPart As Short  
  
 Public Sub New(red As Short, blue As Short, green As Short)  
 redPart = red  
 bluePart = blue  
 greenPart = green  
 End Sub   
  
 Public Shared ReadOnly Red As Color = New Color(&HFF, 0, 0)  
 Public Shared ReadOnly Blue As Color = New Color(0, &HFF, 0)  
 Public Shared ReadOnly Green As Color = New Color(0, 0, &HFF)  
 Public Shared ReadOnly White As Color = New Color(&HFF, &HFF, &HFF)  
End Class

Las constantes y las variables compartidas de solo lectura tienen semántica diferente. 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 una variable compartida de solo lectura, el valor de la variable no se obtiene hasta el tiempo de ejecución. Vea la aplicación siguiente, que consta de dos programas independientes.

file1.vb:

Namespace Program1  
 Public Class Utils  
 Public Shared ReadOnly X As Integer = 1  
 End Class  
End Namespace

file2.vb:

Namespace Program2  
 Module Test  
 Sub Main()  
 Console.WriteLine(Program1.Utils.X)  
 End Sub  
 End Module  
End Namespace

Los espacios de nombres Program1 y Program2 indican dos programas que se compilan de forma independiente. Debido a que la variable Program1.Utils.X se declara como Shared ReadOnly, 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.

### Variables WithEvents

Un tipo puede declarar que controla algunos conjuntos de eventos generados por su variable compartida o de instancia que genera eventos con el modificador WithEvents. Por ejemplo:

Class Raiser  
 Public Event E1()  
  
 Public Sub Raise()  
 RaiseEvent E1  
 End Sub  
End Class  
  
Module Test  
 Private WithEvents x As Raiser  
  
 Private Sub E1Handler() Handles x.E1  
 Console.WriteLine("Raised")  
 End Sub  
  
 Public Sub Main()  
 x = New Raiser()  
 End Sub  
End Module

En este ejemplo, el método E1Handler controla el evento E1 que genera la instancia del tipo Raiser almacenado en la variable de instancia x.

El modificador WithEvents causa que se cambie el nombre de la variable por un carácter de subrayado inicial y se reemplace por una propiedad del mismo nombre que hace el enlace de eventos. Por ejemplo, si el nombre de la variable es F, se cambia por \_F y se declara implícitamente una propiedad F. Si se produce una colisión entre el nuevo nombre de la variable y otra declaración, se informará de un error en tiempo de compilación. Cualquier atributo aplicado a la variable se lleva a la variable con nuevo nombre.

La propiedad implícita creada por una declaración WithEvents se encarga del enlazar y desenlazar los controladores de eventos pertinentes. Cuando se asigna un valor a la variable, la propiedad llama primero al método remove para el evento de la instancia actual en la variable existente (eliminando el controlador de eventos, si hay alguno). Después se lleva a cabo la asignación y la propiedad llama al método add para el evento en la nueva instancia en la variable (enlazando el nuevo controlador de eventos). El código siguiente es equivalente al código anterior para el módulo estándar Test:

Module Test  
 Private \_x As Raiser  
  
 Public Property x() As Raiser  
 Get  
 Return \_x  
 End Get  
  
 Set (Value As Raiser)  
 ' Unhook any existing handlers.  
 If \_x IsNot Nothing Then  
 RemoveHandler \_x.E1, AddressOf E1Handler  
 End If  
  
 ' Change value.  
 \_x = Value  
  
 ' Hook-up new handlers.  
 If \_x IsNot Nothing Then  
 AddHandler \_x.E1, AddressOf E1Handler  
 End If  
 End Set  
 End Property  
  
 Sub E1Handler()  
 Console.WriteLine("Raised")  
 End Sub  
  
 Sub Main()  
 x = New Raiser()  
 End Sub  
End Module

No es válido declarar una variable compartida o de instancia como WithEvents si su tipo es una estructura. Además, no se puede especificar WithEvents en una estructura y WithEvents y ReadOnly no se pueden combinar.

### Inicializadores de variable

Las declaraciones de variables compartidas y de instancias en declaraciones de variables de instancias (pero no compartidas) en estructuras puede incluir inicializadores de variable. Para las variables Shared, los inicializadores de variable corresponden a las instrucciones de asignación que se ejecutan después de que se inicia el programa, pero antes de que se haga referencia a la variable Shared por primera vez. En las variables de instancia, los inicializadores de variable corresponden a instrucciones de asignación que se ejecutan cuando se crea una instancia de la clase. Las estructuras no pueden tener inicializadores de variable de instancia porque no se pueden modificar sus constructores sin parámetros.

Considere el ejemplo siguiente:

Class Test  
 Public Shared x As Double = Math.Sqrt(2.0)  
 Public i As Integer = 100  
 Public s As String = "Hello"  
End Class  
  
Module TestModule  
 Sub Main()  
 Dim a As New Test()  
  
 Console.WriteLine("x = " & Test.x & ", i = " & a.i & ", s = " & a.s)  
 End Sub  
End Module

El ejemplo produce el siguiente resultado:

x = 1.4142135623731, i = 100, s = Hello

Tiene lugar una asignación a x cuando se carga la clase y las asignaciones a i y s ocurren cuando se crea una nueva instancia de la clase.

Es útil considerar los inicializadores de variable de instancia como instrucciones de asignación que se insertan automáticamente en el bloque del constructor del tipo. El ejemplo siguiente contiene varios inicializadores de variable de instancia.

Class A  
 Private x As Integer = 1  
 Private y As Integer = -1  
 Private count As Integer  
  
 Public Sub New()  
 count = 0  
 End Sub  
  
 Public Sub New(n As Integer)  
 count = n  
 End Sub  
End Class  
  
Class B  
 Inherits A  
  
 Private sqrt2 As Double = Math.Sqrt(2.0)  
 Private items As ArrayList = New ArrayList(100)  
 Private max As Integer  
  
 Public Sub New()  
 Me.New(100)  
 items.Add("default")  
 End Sub  
  
 Public Sub New(n As Integer)  
 MyBase.New(n - 1)  
 max = n  
 End Sub  
End Class

El ejemplo corresponde al código que se muestra abajo, donde cada comentario indica una instrucción insertada automáticamente.

Class A  
 Private x, y, count As Integer  
  
 Public Sub New()  
 MyBase.New ' Invoke object() constructor.  
 x = 1 ' This is a variable initializer.  
 y = -1 ' This is a variable initializer.  
 count = 0  
 End Sub  
  
 Public Sub New(n As Integer)  
 MyBase.New ' Invoke object() constructor.   
 x = 1 ' This is a variable initializer.  
 y = - 1 ' This is a variable initializer.  
 count = n  
 End Sub  
End Class  
  
Class B  
 Inherits A  
  
 Private sqrt2 As Double  
 Private items As ArrayList  
 Private max As Integer  
  
 Public Sub New()  
 Me.New(100)   
 items.Add("default")  
 End Sub  
  
 Public Sub New(n As Integer)  
 MyBase.New(n - 1)   
 sqrt2 = Math.Sqrt(2.0) ' This is a variable initializer.  
 items = New ArrayList(100) ' This is a variable initializer.  
 max = n  
 End Sub  
End Class

Todas las variables se inicializan en el valor predeterminado de su tipo antes de que se ejecuten los inicializadores de variable. Por ejemplo:

Class Test  
 Public Shared b As Boolean  
 Public i As Integer  
End Class  
  
Module TestModule  
 Sub Main()  
 Dim t As New Test()  
 Console.WriteLine("b = " & Test.b & ", i = " & t.i)  
 End Sub  
End Module

Como b se inicializa automáticamente en su valor predeterminado cuando se carga la clase e i se inicializa automáticamente en su valor predeterminado cuando se crea una instancia de la clase, el código anterior produce este resultado:

b = False, i = 0

Cada inicializador de variable debe dar un valor del tipo de la variable o de un tipo que sea implícitamente convertible en el tipo de la variable. Un inicializador de variable puede ser circular o hacer referencia a una variable que se inicialice tras él, en cuyo caso el valor de la variable de referencia es su valor predeterminado para el propósito del inicializador. Tal inicializador es de valor incierto.

Hay tres modalidades de inicializadores de variable: regulares, de tamaño de matriz y de objeto. Las dos primeras aparecen después de un signo igual que va después del nombre de tipo, las dos siguientes forman parte de la propia declaración. Solo se puede usar una modalidad de inicializador para una declaración concreta.

#### Inicializadores regulares

Un inicializador regular es una expresión que se puede convertir implícitamente en el tipo de la variable. Aparece después de un signo igual que sigue al nombre del tipo y debe clasificarse como valor. Por ejemplo:

Module Test  
 Dim x As Integer = 10  
 Dim y As Integer = 20  
  
 Sub Main()  
 Console.WriteLine("x = " & x & ", y = " & y)  
 End Sub  
End Module

Este programa produce el siguiente resultado:

x = 10, y = 20

Si la declaración de una variable tiene un inicializador regular, solo se puede declarar una variable cada vez. Por ejemplo:

Module Test  
 Sub Main()  
 ' OK, only one variable declared at a time.  
 Dim x As Integer = 10, y As Integer = 20  
  
 ' Error: Can’t initialize multiple variables at once.  
 Dim a, b As Integer = 10  
 End Sub  
End Module

#### Inicializadores de objetos

Un inicializador de objetos se especifica mediante una expresión de creación de objetos en lugar del nombre del tipo. Un inicializador de objetos es equivalente al incializador normal que asigna el resultado de la expresión de creación de objetos a la variable. En consecuencia

Module TestModule  
 Sub Main()  
 Dim x As New Test(10)  
 End Sub  
End Module

equivale a

Module TestModule  
 Sub Main()  
 Dim x As Test = New Test(10)  
 End Sub  
End Module

Los paréntesis de un inicializador de objetos siempre se interpretan como la lista de argumentos del constructor y nunca como modificadores de tipo de matriz. Un nombre de variable con un inicializador de objetos no puede tener un modificador de tipos de matriz ni un modificador de tipos que admita valores null.

#### Inicializadores de tamaño de matriz

Un inicializador de tamaño de matriz es un modificador del nombre de la variable que ofrece un conjunto de límites superiores de dimensión denotado por expresiones. Las expresiones de límite superior deben clasificarse como valores y deben poder convertirse implícitamente en Integer. El conjunto de límites superiores es equivalente al inicializador de variable de una expresión de creación de matrices con los límites superiores especificados. El número de dimensiones del tipo de matriz se infiere del inicializador de tamaño de la matriz. En consecuencia

Module Test  
 Sub Main()  
 Dim x(5, 10) As Integer  
 End Sub  
End Module

equivale a

Module Test  
 Sub Main()  
 Dim x As Integer(,) = New Integer(5, 10) {}  
 End Sub  
End Module

Todos los límites superiores deben ser iguales o mayores que -1 y todas las dimensiones deben tener especificado un límite superior. Si el tipo de elemento de la matriz que se inicializa es él mismo como tipo de matriz, los modificadores del tipo de matriz van a la derecha del inicializador de tamaño de matriz. Por ejemplo:

Module Test  
 Sub Main()  
 Dim x(5,10)(,,) As Integer  
 End Sub  
End Module

declara una variable local x cuyo tipo es una matriz bidimensional de matrices tridimensionales de Integer, inicializada en una matriz con límites de 0..5 en la primera dimensión y 0..10 en la segunda. No es posible usar un inicializador de tamaño de matriz para inicializar los elementos de una variable cuyo tipo es una matriz de matrices.

Una declaración de variable con un inicializador de tamaño de matriz no puede tener un modificador de tipos de matriz en su tipo ni un inicializador normal.

ArraySizeInitializationModifier ::=  
 OpenParenthesis BoundList CloseParenthesis [ ArrayTypeModifiers ]

BoundList::=  
 Bound |  
 BoundList Comma Bound

Bound ::=  
 Expression |  
 0 To Expression

### Clases System.MarshalByRefObject

El cálculo de referencias de las clases que se derivan de la clase System.MarshalByRefObject se hace entre límites contextuales usando servidores proxy (es decir, por referencia), en lugar de mediante copia (es decir, por valor). Esto significa que una instancia de tal clase no tiene por qué ser una instancia real, sino que puede ser simplemente un código auxiliar que calcula las referencias del acceso a variables y llamadas a métodos entre límites contextuales.

Como resultado, no es posible crear una referencia a la ubicación de almacenamiento de variables definidas en esas clases. Esto significa que las variables de tipo clases que se derivan de System.MarshalByRefObject no se pueden pasar para hacer referencia a parámetros, y no se puede tener acceso a los métodos y variables de variables cuyo tipo es un tipo de valor. En su lugar, Visual Basic trata las variables definidas en estas clases como si fueran propiedades (ya que las restricciones son las mismas en las propiedades).

Hay una excepción a esta regla: un miembro calificado explícita o implícitamente con Me está exento de las restricciones anteriores porque Me tiene siempre la garantía de ser un objeto real, no un proxy.

## Propiedades

Las propiedades son una extensión natural de las variables; ambas son miembros con nombre y con tipos asociados, y la sintaxis que se utiliza para el acceso a las variables y las propiedades es la misma. No obstante, a diferencia de las variables, las propiedades no denotan ubicaciones de almacenamiento. En cambio, las propiedades tienen descriptores de acceso que especifican las instrucciones que se van a ejecutar para leer o escribir sus valores.

Las propiedades se definen mediante declaraciones de propiedad. La primera parte de una declaración de propiedad se parece a una declaración de campo. La segunda parte incluye un descriptor de acceso Get o un descriptor de acceso Set. En el ejemplo siguiente, la clase Button define una propiedad Caption.

Public Class Button  
 Private captionValue As String  
  
 Public Property Caption() As String  
 Get  
 Return captionValue  
 End Get  
  
 Set (Value As String)  
 captionValue = value  
 Repaint()  
 End Set  
 End Property  
  
 ...  
End Class

Basándose en la clase Button definida anteriormente, en el siguiente ejemplo se muestra el uso de la propiedad Caption:

Dim okButton As Button = New Button()  
  
okButton.Caption = "OK" ' Invokes Set accessor.  
Dim s As String = 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.

Si no se especifica ningún tipo para una propiedad y se utiliza semántica estricta, se produce un error en tiempo de compilación; de lo contrario, el tipo de la propiedad es implícitamente Object o el tipo del carácter de tipo de la propiedad. Una declaración de propiedad puede contener un descriptor de acceso Get, que recupera el valor de la propiedad, un descriptor de acceso Set, que almacena el valor de la propiedad, o ambos. Como una propiedad declara métodos de forma implícita, una propiedad se puede declarar con los mismos modificadores que un método. Si la propiedad se define como una interfaz o con el modificador MustOverride, el cuerpo de la propiedad y el constructor End se deben omitir, de lo contrario se producirá un error en tiempo de compilación.

La lista de parámetros de índice constituye la firma de la propiedad, de forma que se pueden sobrecargar las propiedades en los parámetros del índice pero no en el tipo de la propiedad. La lista de parámetros de índice es la misma para un método normal. Sin embargo, ninguno de ellos puede modificarse con el modificador ByRef y ninguno puede llamarse Value (que se reserva para el parámetro de valor implícito del descriptor de acceso Set).

Se puede declarar una propiedad del modo siguiente:

Si la propiedad no especifica ningún modificador de tipo de propiedad, entonces debe tener los dos descriptores de acceso Get y Set. Se dice que la propiedad es de lectura y escritura.

Si la propiedad especifica el modificador ReadOnly, entonces debe tener el descriptor de acceso Get pero no puede tener el descriptor Set. Se dice que la propiedad es de solo lectura. Supone un error en tiempo de compilación que una propiedad de solo lectura sea el destino de una asignación.

Si la propiedad especifica el modificador WriteOnly, entonces debe tener el descriptor de acceso Set pero no puede tener el descriptor Get. Se dice que la propiedad es de solo escritura. Supone un error en tiempo de compilación hacer referencia en una expresión a una propiedad de solo escritura a excepción de como destino de una asignación o como argumento de un método.

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. En el ejemplo siguiente no se declara una propiedad de lectura y escritura única. Se declaran dos propiedades con el mismo nombre, una de solo lectura y otra de solo escritura:

Class A  
 Private nameValue As String  
  
 ' Error, contains a duplicate member name.  
 Public ReadOnly Property Name() As String   
 Get  
 Return nameValue  
 End Get  
 End Property  
  
 ' Error, contains a duplicate member name.  
 Public WriteOnly Property Name() As String   
 Set (Value As String)  
 nameValue = value  
 End Set  
 End Property  
End Class

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.

De manera predeterminada, la accesibilidad de los descriptores Get y Set de una propiedad es la misma que la accesibilidad de la propiedad. Sin embargo, los descriptores Get y Set también pueden especificar su accesibilidad separada de la propiedad. En ese caso, la accesibilidad de un descriptor de acceso debe ser más restrictiva que la de la propiedad y solo un descriptor puede tener una accesibilidad diferente de la propiedad. Los tipos de acceso se consideran más o menos restrictivos del modo siguiente:

Private es más restrictivo que Public, Protected Friend, Protected o Friend.

Friend es más restrictivo que Protected Friend o Public.

Protected es más restrictivo que Protected Friend o Public.

Protected Friend es más restrictivo que Public.

Cuando uno de los descriptores de acceso de una propiedad es accesible pero el otro no, la propiedad se trata como de solo lectura o de solo escritura. Por ejemplo:

Class A  
 Public Property P() As Integer  
 Get  
 ...  
 End Get  
  
 Private Set (Value As Integer)  
 ...  
 End Set  
 End Property  
End Class  
  
Module Test  
 Sub Main()  
 Dim a As A = New A()  
  
 ' Error: A.P is read-only in this context.  
 a.P = 10  
 End Sub  
End Module

Cuando un tipo derivado prevalece sobre una propiedad, la propiedad derivada oculta la propiedad reemplazada tanto para lectura como para escritura. En el ejemplo siguiente, la propiedad P en B oculta la propiedad P en A para lectura y para escritura.

Class A  
 Public WriteOnly Property P() As Integer  
 Set (Value As Integer)  
 End Set  
 End Property  
End Class  
  
Class B  
 Inherits A  
  
 Public Shadows ReadOnly Property P() As Integer  
 Get  
 End Get  
 End Property  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As B = New B  
  
 B.P = 10 ' Error, B.P is read-only.  
 End Sub  
End Module

El dominio de accesibilidad del tipo devuelto o los tipos de parámetro debe ser el mismo o un superconjunto del dominio de accesibilidad de la propiedad. Una propiedad solo puede tener un descriptor de acceso Set y un descriptor de acceso Get.

Excepto por las diferencias en la sintaxis de declaración e invocación, las propiedades Overridable, NotOverridable, Overrides, MustOverride y MustInherit se comportan igual que los métodos Overridable, NotOverridable, Overrides, MustOverride y MustInherit. Cuando se invalida una propiedad, la propiedad que invalida debe ser del mismo tipo (lectura-escritura, solo lectura, solo escritura). Una propiedad Overridable no puede contener un descriptor de acceso Private.

En el ejemplo siguiente, X es una propiedad Overridable de solo lectura, Y es una propiedad Overridable de lectura y escritura, y Z es una propiedad MustOverride de lectura y escritura.

MustInherit Class A  
 Private \_y As Integer  
  
 Public Overridable ReadOnly Property X() As Integer  
 Get  
 Return 0  
 End Get  
 End Property  
  
 Public Overridable Property Y() As Integer  
 Get  
 Return \_y  
 End Get  
 Set (Value As Integer)  
 \_y = value  
 End Set  
 End Property  
  
 Public MustOverride Property Z() As Integer  
End Class

Puesto que Z es MustOverride, la clase contenedora A debe declararse como MustInherit.

Por contraste, a continuación se muestra una clase que se deriva de la clase A:

Class B  
 Inherits A  
  
 Private \_z As Integer  
  
 Public Overrides ReadOnly Property X() As Integer  
 Get  
 Return MyBase.X + 1  
 End Get  
 End Property  
  
 Public Overrides Property Y() As Integer  
 Get  
 Return MyBase.Y  
 End Get  
 Set (Value As Integer)  
 If value < 0 Then  
 MyBase.Y = 0  
 Else  
 MyBase.Y = Value  
 End If  
 End Set  
 End Property  
  
 Public Overrides Property Z() As Integer  
 Get  
 Return \_z  
 End Get  
 Set (Value As Integer)  
 \_z = Value  
 End Set  
 End Property  
End Class

Aquí, las declaraciones de las propiedades X, Y y Z invalidan las propiedades base. 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 la propiedad X y el descriptor de acceso Set de la propiedad Y usan la palabra clave MyBase para tener acceso a las propiedades heredadas. La declaración de la propiedad Z invalida la propiedad MustOverride, por tanto, no hay miembros pendientes MustOverride en la clase B y se permite que B sea una clase normal.

Pueden utilizarse propiedades para retrasar la inicialización de un recurso hasta el momento en que se haga referencia al mismo. Por ejemplo:

Imports System.IO  
  
Public Class ConsoleStreams  
 Private Shared reader As TextReader  
 Private Shared writer As TextWriter  
 Private Shared errors As TextWriter  
  
 Public Shared ReadOnly Property [In]() As TextReader  
 Get  
 If reader Is Nothing Then  
 reader = Console.In  
 End If  
 Return reader  
 End Get  
 End Property  
  
 Public Shared ReadOnly Property Out() As TextWriter  
 Get  
 If writer Is Nothing Then  
 writer = Console.Out  
 End If  
 Return writer  
 End Get  
 End Property  
  
 Public Shared ReadOnly Property [Error]() As TextWriter  
 Get  
 If errors Is Nothing Then  
 errors = Console.Error  
 End If  
 Return errors  
 End Get  
 End Property  
End Class

La clase ConsoleStreams 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 ConsoleStreams puede retrasar su inicialización hasta que se utilicen realmente. Por ejemplo, en la primera referencia a la propiedad Out, como en ConsoleStreams.Out.WriteLine("hello, world"), se inicializa 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.

PropertyMemberDeclaration ::=  
 RegularPropertyMemberDeclaration |  
 MustOverridePropertyMemberDeclaration |  
 AutoPropertyMemberDeclaration

PropertySignature ::=  
 Property Identifier [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 [ As [ Attributes ] TypeName ]

RegularPropertyMemberDeclaration ::=  
 [ Attributes ] [ PropertyModifier+ ] PropertySignature [ ImplementsClause ] LineTerminator  
 PropertyAccessorDeclaration+  
 End Property StatementTerminator

MustOverridePropertyMemberDeclaration ::=  
 [ Attributes ] MustOverridePropertyModifier+ PropertySignature [ ImplementsClause ]  
 StatementTerminator

AutoPropertyMemberDeclaration ::=  
 [ Attributes ] [ AutoPropertyModifier+ ] Property Identifier  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 [ As [ Attributes ] TypeName ] [ Equals Expression ] [ ImplementsClause ] LineTerminator |  
 [ Attributes ] [ AutoPropertyModifier+ ] Property Identifier  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 As [ Attributes ] New [ NonArrayTypeName  
 [ OpenParenthesis [ ArgumentList ] CloseParenthesis ] ] [ ObjectCreationExpressionInitializer ]  
 [ ImplementsClause ] LineTerminator

InterfacePropertyMemberDeclaration ::=  
 [ Attributes ] [ InterfacePropertyModifier+ ] PropertySignature StatementTerminator

AutoPropertyModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 Overridable |  
 NotOverridable |  
 Overrides |  
 Overloads

PropertyModifier ::=  
 AutoPropertyModifier |  
 Default |  
 ReadOnly |  
 WriteOnly |  
 Iterator

MustOverridePropertyModifier ::= PropertyModifier | MustOverride

InterfacePropertyModifier ::=  
 Shadows |  
 Overloads |  
 Default |  
 ReadOnly |  
 WriteOnly

PropertyAccessorDeclaration ::= PropertyGetDeclaration | PropertySetDeclaration

### Declaraciones del descriptor de acceso Get

Un descriptor de acceso Get (getter) se declara mediante una declaración Get de propiedad. Una declaración Get de propiedad consiste en la palabra clave Get seguida de un bloque de instrucciones. Dada una propiedad denominada P, una declaración de acceso Get declara implícitamente un método con el nombre get\_P con los mismos modificadores, tipo y lista de parámetros que la propiedad. Si el tipo contiene una declaración con ese nombre, se produce un error en tiempo de compilación, pero la declaración implícita se ignora por razones de enlace de nombres.

Una variable local especial, que se declara implícitamente en el espacio de declaración del cuerpo del descriptor de acceso Get con el mismo nombre que la propiedad, representa el valor devuelto de la propiedad. La variable local tiene una semántica de resolución de nombres especial cuando se usa en expresiones. Si la variable local se usa en un contexto que espera una expresión que se clasifica como un grupo de métodos, como una expresión de invocación, entonces el nombre se resuelve como la función y no como la variable local. Por ejemplo:

ReadOnly Property F(i As Integer) As Integer  
 Get  
 If i = 0 Then  
 F = 1 ' Sets the return value.  
 Else  
 F = F(i - 1) ' Recursive call.  
 End If  
 End Get  
End Property

El uso de los paréntesis puede causar situaciones ambiguas (como F(1) donde F es una propiedad cuyo tipo es una matriz unidimensional). En todas las situaciones ambiguas, el nombre se resuelve como la propiedad, en lugar de como la variable local. Por ejemplo:

ReadOnly Property F(i As Integer) As Integer()  
 Get  
 If i = 0 Then  
 F = new Integer(2) { 1, 2, 3 }  
 Else  
 F = F(i – 1) ' Recursive call, not index.  
 End If  
 End Get  
End Property

Cuando el flujo de control deja el cuerpo del descriptor de acceso Get, el valor de la variable local vuelve a la expresión de invocación. Puesto que invocar un descriptor de acceso Get equivale conceptualmente a leer el valor de una variable, se considera una técnica de programación poco recomendable que los descriptores de acceso Get muestren efectos secundarios, como se ilustra en este ejemplo:

Class Counter  
 Private Value As Integer  
  
 Public ReadOnly Property NextValue() As Integer  
 Get  
 Value += 1  
 Return Value  
 End Get  
 End Property  
End Class

El valor de la propiedad NextValue 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 estos deban escribirse siempre únicamente para que devuelvan los valores almacenados en variables. De hecho, los descriptores de acceso Get a menudo calculan el valor de una propiedad mediante el acceso a varias variables 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.

Nota Los descriptores de acceso Get tienen la misma restricción en la ubicación en las líneas que las subrutinas. La primera y la última instrucción, y el bloque deben aparecen al principio de una línea lógica.

PropertyGetDeclaration ::=  
 [ Attributes ] [ AccessModifier ] Get LineTerminator  
 [ Block ]  
 End Get StatementTerminator

### Declaraciones del descriptor de acceso Set

Un descriptor de acceso Set (setter) se declara mediante una declaración Set de propiedad. Una declaración Set de propiedad consta de una palabra clave Set, una lista de parámetros opcionales y un bloque de instrucciones. Dada una propiedad denominada P, una declaración de acceso setter declara implícitamente un método con el nombre set\_P con los mismos modificadores, tipo y lista de parámetros que la propiedad. Si el tipo contiene una declaración con ese nombre, se produce un error en tiempo de compilación, pero la declaración implícita se ignora por razones de enlace de nombres.

Si se especifica una lista de parámetros, debe tener por lo menos un miembro, que no debe tener más modificadores que ByVal y su tipo debe ser el mismo que el de la propiedad. El parámetro representa el valor de propiedad que se está estableciendo. Si el parámetro se omite, se declara un nombre de parámetro Value de forma implícita.

Nota Los descriptores de acceso Set tienen la misma restricción en la ubicación en las líneas que las subrutinas. La primera y la última instrucción, y el bloque deben aparecen al principio de una línea lógica.

PropertySetDeclaration ::=  
 [ Attributes ] [ AccessModifier ] Set [ OpenParenthesis [ ParameterList ] CloseParenthesis ] LineTerminator  
 [ Block ]  
 End Set StatementTerminator

### Propiedades predeterminadas

Una propiedad que especifica el modificador Default se denomina propiedad predeterminada. Cualquier tipo que permita propiedades puede tener una propiedad predeterminada, incluidas las interfaces. Es posible hacer referencia a la propiedad predeterminada sin tener que calificar la instancia con el nombre de la propiedad. Por tanto, dada una clase

Class Test  
 Public Default ReadOnly Property Item(i As Integer) As Integer  
 Get  
 Return i  
 End Get  
 End Property  
End Class

el código

Module TestModule  
 Sub Main()  
 Dim x As Test = New Test()  
 Dim y As Integer  
  
 y = x(10)  
 End Sub  
End Module

equivale a

Module TestModule  
 Sub Main()  
 Dim x As Test = New Test()  
 Dim y As Integer  
  
 y = x.Item(10)  
 End Sub  
End Module

Una vez que una propiedad se declara Default, todas las propiedades sobrecargadas en ese nombre en la jerarquía de herencia se convierten en la propiedad predeterminada, se haya declarado Default o no. Declarar una propiedad como Default en una clase derivada cuando la clase base declaró una propiedad predeterminada por otro nombre no requiere ningún otro modificador como Shadows u Overrides. Esto es así porque la propiedad predeterminada no tiene identificador ni firma, de manera que no puede sobrecargarse ni ocultarse. Por ejemplo:

Class Base  
 Public ReadOnly Default Property Item(i As Integer) As Integer  
 Get  
 Console.WriteLine("Base = " & i)  
 End Get  
 End Property  
End Class  
  
Class Derived  
 Inherits Base  
  
 ' This hides Item, but does not change the default property.  
 Public Shadows ReadOnly Property Item(i As Integer) As Integer  
 Get  
 Console.WriteLine("Derived = " & i)  
 End Get  
 End Property  
End Class  
  
Class MoreDerived  
 Inherits Derived  
  
 ' This declares a new default property, but not Item.  
 ' This does not need to be declared Shadows  
 Public ReadOnly Default Property Value(i As Integer) As Integer  
 Get  
 Console.WriteLine("MoreDerived = " & i)  
 End Get  
 End Property  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As MoreDerived = New MoreDerived()  
 Dim y As Integer  
 Dim z As Derived = x  
  
 y = x(10) ' Calls MoreDerived.Value.  
 y = x.Item(10) ' Calls Derived.Item  
 y = z(10) ' Calls Base.Item  
 End Sub  
End Module

Este programa produce el siguiente resultado:

MoreDerived = 10  
Derived = 10  
Base = 10

Todas las propiedades predeterminadas declaradas dentro de un tipo deben tener el mismo nombre y, por cuestiones de claridad, deben especificar el modificador Default. Como una propiedad predeterminada sin parámetros de índice causaría una situación ambigua cuando se asignan instancias de la clase contenedora, las propiedades predeterminadas deben tener parámetros de índice. Además, si una propiedad sobrecargada en un nombre concreto que incluye el modificador Default, todas las propiedades sobrecargadas en ese nombre deben especificarlo. Las propiedades predeterminadas no pueden ser Shared y por lo menos un descriptor de acceso de la propiedad no debe ser Private.

### Propiedades implementadas automáticamente

Si una propiedad omite la declaración de los descriptores de acceso, se proporcionará automáticamente una implementación de la propiedad salvo que la propiedad se declare en una interfaz o se declare MustOverride. Las propiedades de solo lectura/escritura sin argumentos se pueden implementar automáticamente; de lo contrario se produce un error en tiempo de compilación.

Una propiedad implementada automáticamente x, incluso una que invalide otra propiedad, introduce una variable local privada \_x con el mismo tipo que la propiedad. Si se produce una colisión entre el nombre de la variable local y otra declaración, se informará de un error en tiempo de compilación. El descriptor de acceso Get de la propiedad implementada automáticamente devuelve el valor del descriptor de acceso Set que establece el valor de la local. Por ejemplo, la declaración:

Public Property x() As Integer

es más o menos equivalente a:

Private \_x As Integer  
Public Property x() As Integer  
 Get  
 Return \_x  
 End Get  
 Set (value As Integer)  
 \_x = value  
 End Set  
End Property

Al igual que las declaraciones de variables, una propiedad automáticamente implementada puede incluir un inicializador. Por ejemplo:

Public Property x() As Integer = 10  
Public Shared Property y() As New Customer() With { .Name = "Bob" }

Anotación

Cuando se inicializa una propiedad automáticamente implementada, se hace a través de la propiedad, no del campo subyacente. Esto es para que las propiedades que invalidan puedan interceptar la inicialización si lo necesitan.

En las propiedades automáticamente implementadas se permiten los inicializadores de matrices, excepto que no hay modo de especificar los límites de la matriz de forma explícita. Por ejemplo:

' Valid  
Property x As Integer() = {1, 2, 3}  
Property y As Integer(,) = {{1, 2, 3}, {12, 13, 14}, {11, 10, 9}}  
  
' Invalid  
Property x4(5) As Short

### Propiedades de iterador

Una *propiedad de iterador* es una propiedad con el modificador Iterator. Se usa por el mismo motivo que un método iterador (10.1.2): es una manera conveniente de generar una secuencia que la instrucción For Each pueda usar. El descriptor de acceso Get de una propiedad de iterador se interpreta de la misma forma que un método iterador.

Una propiedad de iterador debe tener un descriptor de acceso Get explícito y su tipo debe ser IEnumerator, IEnumerable, IEnumerator(Of T) o IEnumerable(Of T) para algún T.

A continuación se muestra un ejemplo de una propiedad de iterador:

Class Family  
 Property Daughters As New List(Of String) From {"Beth", "Diane"}  
 Property Sons As New List(Of String) From {"Abe", "Carl"}

ReadOnly Iterator Property Children As IEnumerable(Of String)  
 Get  
 For Each name In Daughters : Yield name : Next  
 For Each name In Sons : Yield name : Next  
 End Get  
 End Property  
End Class

Module Module1  
 Sub Main()  
 Dim x As New Family  
 For Each c In x.Children  
 Console.WriteLine(c) ' prints Beth, Diane, Abe, Carl  
 Next  
 End Sub  
End Module

## Operadores

Los operadores son métodos que definen el significado de un operador existente de Visual Basic para la clase contenedora. Cuando el operador se aplica a la clase en una expresión, el operador se compila en una llamada al método del operador definido en la clase. La definición de un operador de una clase también se denomina sobrecarga del operador. No es posible sobrecargar un operador que ya existe; en la práctica, esto se aplica básicamente a los operadores de conversión. Por ejemplo, no es posible sobrecargar la conversión de una clase derivada en una clase base:

Class Base  
End Class  
  
Class Derived  
 ' Cannot redefine conversion from Derived to Base,  
 ' conversion will be ignored.  
 Public Shared Widening Operator CType(s As Derived) As Base  
 ...  
 End Operator  
End Class

Los operadores también se pueden sobrecargar en el sentido común del término:

Class Base  
 Public Shared Widening Operator CType(b As Base) As Integer  
 ...  
 End Operator  
  
 Public Shared Narrowing Operator CType(i As Integer) As Base  
 ...  
 End Operator  
End Class

Las declaraciones de operadores no agregan nombres explícitamente al espacio de declaración del tipo contenedor; sin embargo, declaran de forma implícita un método correspondiente que comienza con los caracteres “op\_”. En las secciones siguientes se enumeran los nombres de método correspondiente de cada operador.

Se pueden definir tres clases de operadores: operadores unarios, operadores binarios y operadores de conversión. Todas las declaraciones de operadores comparten ciertas restricciones:

Siempre deben ser Public y Shared. Se puede omitir el modificador Public en contextos en los que se pueda suponer.

Los parámetros de un operador no se pueden declarar ByRef, Optional o ParamArray.

El tipo de al menos uno de los operandos o el valor devuelto debe ser el tipo que contiene el operador.

No hay variable devuelta de función definida para los operadores. Por consiguiente, la instrucción Return debe usarse para devolver valores de un cuerpo de operador.

La única excepción a estas restricciones se aplica a los tipos de valor que aceptan valores null. Como los tipos de valor que admiten valores null no tienen una definición de tipo real, un tipo de valor puede declarar operadores definidos por el usuario para la versión que admite valores null del tipo. Cuando se determina si un tipo puede declarar un operador definido por el usuario concreto, los modificadores ? se quitan los primeros de todos los tipos implicados en la declaración por cuestiones de comprobación de validez. Esta relajación no se aplica al tipo de valor devuelto de los operadores IsTrue e IsFalse; deben seguir devolviendo Boolean, no Boolean?.

La precedencia y asociatividad de un operador no puede modificarse mediante una declaración de operador.

Nota Los operadores tienen la misma restricción sobre la ubicación en las líneas que las subrutinas. La primera y la última instrucción, y el bloque deben aparecen al principio de una línea lógica.

OperatorDeclaration ::=  
 [ Attributes ] [ OperatorModifier+ ] Operator OverloadableOperator OpenParenthesis ParameterList CloseParenthesis  
 [ As [ Attributes ] TypeName ] LineTerminator  
 [ Block ]  
 End Operator StatementTerminator

OperatorModifier ::= Public | Shared | Overloads | Shadows | Widening | Narrowing

OverloadableOperator ::=  
 + | - | \* | / | \ | & | Like | Mod | And | Or | Xor | ^ | < < | > > |  
 = | < > | > | < | > = | < = | Not | IsTrue | IsFalse | CType

### Operadores unarios

Los siguientes operadores unarios se pueden sobrecargar:

El operador unario más + (método correspondiente: op\_UnaryPlus)

El operador unario menos - (método correspondiente: op\_UnaryNegation)

El operador lógico Not (método correspondiente: op\_OnesComplement)

Los operadores IsTrue e IsFalse (métodos correspondientes: op\_True, op\_False)

Todos los operadores unarios sobrecargados deben tomar un único parámetro del tipo contenedor y pueden devolver cualquier tipo, salvo IsTrue e IsFalse, que deben devolver Boolean. Si el tipo contenedor es un tipo genérico, los parámetros de tipo deben coincidir con los parámetros de tipo del tipo contenedor. Por ejemplo,

Structure Complex  
 ...  
  
 Public Shared Operator +(v As Complex) As Complex  
 Return v  
 End Operator  
End Structure

Si un tipo sobrecarga uno de IsTrue o IsFalse, entonces debe sobrecargar también el otro. Si solo se sobrecarga uno, se produce un error en tiempo de compilación.

Nota   IsTrue e IsFalse no son palabras reservadas.

### Operadores binarios

Los siguientes operadores binarios se pueden sobrecargar:

Los operadores de suma +, resta -, multiplicación \*, división /, división de enteros \, módulo Mod y exponenciación ^ (método correspondiente: op\_Addition, op\_Subtraction, op\_Multiply, op\_Division, op\_IntegerDivision, op\_Modulus, op\_Exponent)

Los operadores relacionales =, <>, <, >, <=, >= (métodos correspondientes: op\_Equality, op\_Inequality, op\_LessThan, op\_GreaterThan, op\_LessThanOrEqual, op\_GreaterThanOrEqual)

Nota Aunque el operador de igualdad se puede sobrecargar, el operador de asignación (usado solo en instrucciones de asignación) no.

El operador Like (método correspondiente: op\_Like)

El operador de concatenación & (método correspondiente: op\_Concatenate)

Los operadores lógicos And, Or y Xor (métodos correspondientes: op\_BitwiseAnd, op\_BitwiseOr, op\_ExclusiveOr)

Los operadores de desplazamiento << y >> (métodos correspondientes: op\_LeftShift, op\_RightShift)

Todos los operadores binarios sobrecargados deben tomar el tipo contenedor como uno de los parámetros. Si el tipo contenedor es un tipo genérico, los parámetros de tipo deben coincidir con los parámetros de tipo del tipo contenedor. Los operadores de desplazamiento restringen aún más esta regla y requieren que el primer parámetro sea del tipo contenedor; el segundo debe ser siempre de tipo Integer.

Los operadores binarios siguientes se deben declarar en pares:

Los operadores = y <>

Los operadores > y <

Los operadores >= y <=

Si se declara uno de los pares, también se debe declarar el otro con tipos devueltos y parámetros coincidentes, de lo contrario se producirá un error en tiempo de compilación.

Anotación

El objeto de requerir declaraciones emparejadas de operadores relacionales es intentar asegurar al menos un nivel mínimo de coherencia lógica en los operadores sobrecargados.

En contraste con los operadores relacionales, se desaconseja la sobrecarga de los operadores de división y división entero, aunque no sea un error.

Anotación

En general, los dos tipos de división deberían ser totalmente distintas: un tipo que admite la división integral (en cuyo caso debería admitir \) o no (en cuyo caso debería admitir /). Pensamos en considerar un error definir ambos operadores pero como sus lenguajes no distinguen por lo genera entre dos tipos de división del modo que lo hace Visual Basic, creímos que era más seguro permitirlo, pero desaconsejarlo en todo momento.

Los operadores de asignación compuestos no se pueden sobrecargar directamente. En cambio, cuando se sobrecarga el operador binario correspondiente, el operador de asignación compuesto usará el operador sobrecargado. Por ejemplo:

Structure Complex  
 ...  
  
 Public Shared Operator +(x As Complex, y As Complex) \_  
 As Complex  
 ...  
 End Operator  
End Structure  
  
Module Test  
 Sub Main()  
 Dim c1, c2 As Complex  
 ' Calls the overloaded + operator  
 c1 += c2  
 End Sub  
End Module

### Operadores de conversión

Los operadores de conversión definen nuevas conversiones entre tipos. Las nuevas conversiones se denominan conversiones definidas por el usuario. 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. Las conversiones se deben clasificar como 'widening' o 'narrowing'. Una declaración de operador de conversión que incluye la palabra clave Widening presenta una conversión widening definida por el usuario (método correspondiente: op\_Implicit). Una declaración de operador de conversión que incluye la palabra clave Narrowing presenta una conversión narrowing definida por el usuario (método correspondiente: op\_Explicit).

En general, las conversiones widening definidas por el usuario deben diseñarse para que nunca produzcan excepciones ni pierdan información. Si una conversión definida por el usuario puede producir 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 narrowing. En el siguiente ejemplo:

Structure Digit  
 Dim value As Byte  
  
 Public Sub New(value As Byte)  
 if value < 0 OrElse value > 9 Then Throw New ArgumentException()  
 Me.value = value  
 End Sub  
  
 Public Shared Widening Operator CType(d As Digit) As Byte  
 Return d.value  
 End Operator  
  
 Public Shared Narrowing Operator CType(b As Byte) As Digit  
 Return New Digit(b)  
 End Operator  
End Structure

la conversión de Digit en Byte es widening porque nunca produce excepciones ni pierde información, pero la conversión de Byte en Digit es narrowing ya que Digit solo puede representar un subconjunto de los posibles valores de un Byte.

A diferencia de los demás miembros de tipo que se pueden sobrecargar, la firma de un operador de conversión incluye el tipo de destino de conversión. Este es el único miembro de tipo en el que el tipo de valor devuelto participa en la firma. La clasificación widening o narrowing de un operador de conversión, sin embargo, no forma parte de la firma del operador. Por lo tanto, una clase o una estructura no puede declarar a la vez operadores de conversión widening y narrowing con los mismos tipos de origen y destino.

Un operador de conversión definido por el usuario debe convertir de o en un tipo contenedor; por ejemplo, es posible que la clase C defina una conversión de C en Integer y de Integer en C, pero no de Integer en Boolean. Si el tipo contenedor es un tipo genérico, los parámetros de tipo deben coincidir con los parámetros de tipo del tipo contenedor. Además, tampoco es posible redefinir una conversión intrínseca (es decir, no definida por el usuario). Como resultado, un tipo no puede declarar una conversión donde:

El tipo de origen y de destino son el mismo.

Tanto el tipo de origen como el de destino no son el tipo que define el operador de conversión.

El tipo de origen o el de destino es un tipo de interfaz.

El tipo de origen y el de destino están relacionados por herencia (incluido Object).

La única excepción a estas reglas se aplica a los tipos de valor que aceptan valores null. Como los tipos de valor que admiten valores null no tienen una definición de tipo real, un tipo de valor puede declarar conversiones definidas por el usuario para la versión que admite valores null del tipo. Cuando se determina si un tipo puede declarar una conversión definida por el usuario concreta, los modificadores ? se quitan los primeros de todos los tipos implicados en la declaración por cuestiones de comprobación de validez. Por tanto, la siguiente declaración es válida porque S puede definir una conversión de S en T:

Structure T  
 ...  
End Structure  
  
Structure S  
 Public Shared Widening Operator CType(ByVal v As S?) As T  
 ...  
 End Operator  
End Structure

Sin embargo, la siguiente declaración no es válida porque la estructura S no puede definir una conversión de S en S:

Structure S  
 Public Shared Widening Operator CType(ByVal v As S) As S?  
 ...  
 End Operator  
End Structure

### Asignación de operadores

Como el conjunto de operadores que Visual Basic admite puede no coincidir exactamente con el conjunto de operadores de otros lenguajes .NET Framework, algunos operadores se asignan específicamente a otros operadores cuando se definen o usan. Concretamente:

Si se define un operador de división de enteros se definirá automáticamente un operador de división normal (utilizable solo desde otros lenguajes) que llamará al operador de división de enteros.

Si se sobrecargan los operadores Not, And y Or solo se sobrecarga el operador bit a bit de la perspectiva de otros lenguajes que distinguen entre los operadores lógicos y los operadores bit a bit.

Una clase que sobrecarga solo los operadores lógicos en un lenguaje que distingue entre operadores lógicos y bit a bit (es decir, lenguajes que usan op\_LogicalNot, op\_LogicalAnd y op\_LogicalOr para Not, And y Or, respectivamente) tendrán sus operadores lógicos asignados a los operadores lógicos de Visual Basic. Si ambos operadores lógicos y bit a bit están sobrecargados, solo se utilizarán los operadores bit a bit.

Si se sobrecargan los operadores << y >>, solo se sobrecargan los operadores firmados desde la perspectiva de otros lenguajes que distinguen entre los operadores con firma y los operadores de desplazamiento sin signo.

Una clase que sobrecarga solo un operador de desplazamiento sin signo tendrá un operador de desplazamiento sin signo asignado al correspondiente de Visual Basic. Si ambos operadores de desplazamiento sin signo y con signo están sobrecargados, solo se usará el operador de desplazamiento con signo.

# Instrucciones

Las instrucciones representan código ejecutable.

Statement ::=  
 LabelDeclarationStatement |  
 LocalDeclarationStatement |  
 WithStatement |  
 SyncLockStatement |  
 EventStatement |  
 AssignmentStatement |  
 InvocationStatement |  
 ConditionalStatement |  
 LoopStatement |  
 ErrorHandlingStatement |  
 BranchStatement |  
 ArrayHandlingStatement |  
 UsingStatement |  
 AwaitStatement |  
 YieldStatement

Anotación

El compilador de Microsoft Visual Basic solo permite las instrucciones que comienzan con una palabra clave o un identificador. Por consiguiente, se permite la instrucción de invocación "Call (Console).WriteLine", pero no se permite "(Console).WriteLine".

## Control Flow

El *flujo de control* es la secuencia en la que se ejecutan las instrucciones y expresiones. El orden de ejecución depende de la instrucción o expresión concreta.

Por ejemplo, al evaluar un operador de suma (sección 11.3), el operando izquierdo se evalúa en primer lugar, a continuación, el operando derecho y, finalmente, el propio operador. En los bloques (sección 10.1.2) se ejecuta en primer lugar la primera subinstrucción y, a continuación, se ejecutan una a una las instrucciones del bloque.

Implícito en este orden está el concepto de un *punto de control*, que es la siguiente operación que se va a ejecutar. Cuando se invoca (o "se llama") un método, decimos que crea una *instancia* del método. Una instancia de método tiene su propia copia de los parámetros y variables locales del método, así como su propio punto de control.

### Métodos normales

A continuación se muestra un ejemplo de un método normal:

Function Test() As Integer  
 Console.WriteLine("hello")  
 Return 1  
End Sub

Dim x = Test() ' invokes the function, prints "hello", assigns 1 to x

Al invocarse un método normal,

1. En primer lugar, se crea una instancia del método específica para esa invocación. Esta instancia incluye una copia de todos los parámetros y variables locales del método.
2. A continuación, todos sus parámetros se inicializan en los valores proporcionados y, todas sus variables locales, en los valores predeterminados de sus tipos.
3. En el caso de Function, también se inicializa una variable local implícita denominada *variable devuelta de función*, cuyo nombre es el nombre de la función, su tipo es el tipo de valor devuelto de la función y su valor inicial es el valor predeterminado de su tipo.
4. El punto de control de la instancia del método se establece a continuación en la primera instrucción del cuerpo del método, que empieza a ejecutarse inmediatamente a partir de aquí (sección 10.1.4).

Cuando el flujo de control sale del cuerpo del método normalmente (al llegar a End Function o End Sub que marcan su fin, o mediante una instrucción Return o Exit explícita), suele volver al llamador de la instancia del método. Si hay una variable devuelta de función, el resultado de la invocación es el valor final de esta variable.

Cuando el flujo de control sale del cuerpo del método mediante una excepción no controlada, esta se propaga al llamador.

Tras la salida del flujo de control, ya no hay referencias activas a la instancia del método. Si la instancia del método contenía las únicas referencias a su copia de variables o parámetros locales, pueden someterse a la recolección de elementos no utilizados.

### Métodos Iterator

Los métodos Iterator se usan como una manera conveniente de generar una secuencia que la instrucción For Each pueda usar. Los métodos Iterator usan la instrucción Yield (sección 10.15) para proporcionar elementos de la secuencia. (Un método iterator sin instrucciones Yield producirá una secuencia vacía). A continuación se muestra un ejemplo de un método iterator:

Iterator Function Test() As IEnumerable(Of Integer)  
 Console.WriteLine("hello")  
 Yield 1  
 Yield 2  
End Function

Dim en = Test()  
For Each x In en ' prints "hello" before the first x  
 Console.WriteLine(x) ' prints "1" and then "2"  
Next

Al invocarse un método iterador cuyo tipo de valor devuelto es IEnumerator(Of T),

1. En primer lugar, se crea una instancia del método iterator específica para esa invocación. Esta instancia incluye una copia de todos los parámetros y variables locales del método.
2. A continuación, todos sus parámetros se inicializan en los valores proporcionados y, todas sus variables locales, en los valores predeterminados de sus tipos.
3. También se inicializa una variable local implícita denominada *variable de iterador actual*, cuyo tipo es T y cuyo valor inicial es el valor predeterminado de su tipo.
4. El punto de control de la instancia del método se establece a continuación en la primera instrucción del cuerpo del método.
5. A continuación se crea un *objeto iterador*, asociado a esta instancia del método. El objeto iterador implementa el tipo de valor devuelto declarado y su comportamiento se describe a continuación.
6. El flujo de control se reanuda entonces *inmediatamente* en el llamador y el resultado de la invocación es el objeto iterador. Observe que esta transferencia se realiza sin salir de la instancia del método iterator y no provoca finalmente la ejecución de los controladores. El objeto iterador sigue haciendo referencia a la instancia de método, que no se recolectará como elementos utilizados siempre que haya una referencia activa al objeto iterador.

Cuando se obtiene acceso a la propiedad Current del objeto iterador, se devuelve la *variable actual* de la invocación.

Cuando se invoca el método MoveNext del objeto iterador, la invocación no crea una nueva instancia de método. En su lugar, se usa la instancia de método existente (así como su punto de control, variables y parámetros locales): la instancia que se creó cuando el método iterator se invocó en primer lugar. El flujo de control reanuda la ejecución en el punto de control de dicha instancia de método y avanza por el cuerpo del método iterator de la forma habitual.

Cuando se invoca el método Dispose del objeto iterador, se vuelve a usar la instancia de método existente. El flujo de control se reanuda en el punto de control de dicha instancia de método, pero entonces se comporta inmediatamente como si una instrucción Exit Function fuera la siguiente operación.

Las descripciones de comportamiento anteriores de la invocación de MoveNext o Dispose en un objeto iterador solo se aplican si todas las invocaciones anteriores de MoveNext o Dispose en dicho objeto iterador ya han vuelto a sus llamadores. En caso contrario, el comportamiento es indefinido.

Cuando el flujo de control sale del cuerpo del método iterator normalmente (al llegar a End Function que marca su fin, o mediante una instrucción Return o Exit explícita), debe haberlo hecho en el contexto de una invocación de la función MoveNext o Dispose en un objeto iterador para reanudar la instancia de método iterator, y habrá estado usando la instancia de método que se creó al invocar el método iterator en primer lugar. El punto de control de la instancia se deja en la instrucción End Function y el flujo de control se reanuda en el llamador; además, si lo hubiera reanudado una llamada a MoveNext, el valor False se devuelve al llamador.

Cuando el flujo de control sale del cuerpo del método iterator mediante una excepción no controlada, la excepción se propaga al llamador, que de nuevo será una invocación de MoveNext o de Dispose.

En cuanto al resto de tipos de valor devuelto posibles de una función de iterador:

1. Al invocarse un método iterator cuyo tipo de valor devuelto es IEnumerable(Of T) para algún T, se crea una instancia por primera vez (específica para dicha invocación del método iterator) de todos los parámetros del método y se inicializan con los valores proporcionados. El resultado de la invocación es un objeto que implementa el tipo de valor devuelto. Si se llama al método GetEnumerator de este objeto, crea una instancia (específica para dicha invocación de GetEnumerator) de todos los parámetros y variables locales del método. Inicializa los parámetros en los valores ya guardados y continúa de la misma forma que en el método iterator anterior.
2. Al invocarse un método iterador cuyo tipo de valor devuelto es la interfaz no genérica IEnumerator, el comportamiento es exactamente igual que el de IEnumerator(Of Object).
3. Al invocarse un método iterador cuyo tipo devuelto es la interfaz no genérica IEnumerable, el comportamiento es exactamente igual que el de IEnumerable(Of Object).

### Métodos asincrónicos

Los métodos asincrónicos son una manera conveniente de trabajar durante períodos prolongados sin, por ejemplo, bloquear la interfaz de usuario de una aplicación. Async es la abreviatura de *Asynchronous* (asincrónico): significa que el llamador del método asincrónico reanudará su ejecución sin demora, pero la finalización eventual de la instancia del método asincrónico puede producirse posteriormente en el futuro. Por convención, los métodos asincrónicos se denominan con el sufijo "Async".

Async Function TestAsync() As Task(Of String)  
 Console.WriteLine("hello")  
 Await Task.Delay(100)  
 Return "world"  
End Function

Dim t = TestAsync() ' prints "hello"  
Console.WriteLine(Await t) ' prints "world"

Anotación

Los métodos asincrónicos *no* se ejecutan en un subproceso en segundo plano. En su lugar, permiten que un método se suspenda automáticamente mediante el operador Await y que este se programe para reanudarse en respuesta a algún evento.

Al invocarse un método asincrónico

1. En primer lugar, se crea una instancia del método asincrónico específica de esa invocación. Esta instancia incluye una copia de todos los parámetros y variables locales del método.
2. A continuación, todos sus parámetros se inicializan en los valores proporcionados y, todas sus variables locales, en los valores predeterminados de sus tipos.
3. En el caso de un método asincrónico con el tipo de valor devuelto Task(Of T) para algún T, también se inicializa una variable local implícita denominada *variable devuelta de tarea*, cuyo tipo es T y cuyo valor inicial es el valor predeterminado de T.
4. Si el método asincrónico es Function con el tipo de valor devuelto Task o Task(Of T) para algún T, se crea un objeto de ese tipo implícitamente, asociado a la invocación actual. Este se llama *objeto asincrónico* y representa el trabajo posterior que se realizará mediante la ejecución de la instancia del método asincrónico. Cuando el control se reanuda en el llamador de esta instancia de método asincrónico, el llamador recibirá este objeto asincrónico como resultado de la invocación.
5. El punto de control de la instancia se establece a continuación en la primera instrucción del cuerpo del método asincrónico, que empieza a ejecutarse inmediatamente a partir de aquí [sección 10.1.2].

**Delegado de reanudación y llamador actual**

Como se detalla en la sección 11.25 "Operador Await", la ejecución de una expresión Await tiene la capacidad de suspender el punto de control de la instancia del método y deja que el flujo de control se dirija a otra parte. El flujo de control puede reanudarse posteriormente en el mismo punto de control de la instancia mediante la invocación de un *delegado de reanudación*. Observe que esta suspensión se realiza sin salir del método asincrónico y no provoca la ejecución de los controladores finally. Tanto el delegado de reanudación como el resultado de Task o Task(Of T) (si lo hay) seguirán haciendo referencia a la instancia de método, que no se someterá a la recolección como elementos utilizados siempre que haya una referencia activa al delegado o resultado.

Anotación

Es útil imaginar la instrucción "Dim x = Await WorkAsync()" aproximadamente como forma sintáctica abreviada de lo siguiente:

Dim temp = WorkAsync().GetAwaiter()  
 If Not temp.IsCompleted Then  
 temp.OnCompleted(resumptionDelegate)  
 Return [task]  
 CONT: ' invocation of *resumptionDelegate* will resume here  
 End If  
 Dim x = temp.GetResult()

En el ejemplo siguiente, el *llamador actual* de la instancia de método se define como llamador original o llamador del delegado de reanudación, lo que sea más reciente.

Cuando el flujo de control sale del cuerpo del método asincrónico (al llegar a End Sub o End Function que marcan su fin, mediante una instrucción Return o Exit explícita o mediante una excepción no controlada), el punto de control de la instancia se establece al final del método. Entonces, el comportamiento depende del tipo de valor devuelto del método asincrónico.

1. En el caso de una Async Function con tipo de valor devuelto Task:
   1. Si el flujo de control sale mediante una excepción no controlada, el estado del objeto asincrónico se establece en TaskStatus.Faulted y su propiedad Exception.InnerException se establece en la excepción (salvo: determinadas excepciones definidas según su implementación como OperationCanceledException lo cambian a TaskStatus.Canceled). El flujo de control se reanuda en el llamador actual.
   2. De lo contrario, el estado del objeto asincrónico se establece en TaskStatus.Completed. El flujo de control se reanuda en el llamador actual.

Anotación  
  
El objetivo de la tarea y lo que hace que los métodos asincrónicos resulten interesantes es que, cuando una tarea se completa, se ejecutarán los delegados de reanudación de los métodos que estaban a la espera, es decir, se desbloquearán.

1. En el caso de una Async Function con el tipo de valor devuelto Task(Of T) para algún T: el comportamiento es como antes, salvo que en los casos sin excepciones, la propiedad Result del objeto asincrónico también se establece en el valor final de la variable devuelta de tarea.
2. En el caso de Async Sub:
   1. Si el flujo de control sale mediante una excepción no controlada, dicha excepción se propaga al entorno de alguna forma específica de la implementación. El flujo de control se reanuda en el llamador actual.
   2. De lo contrario, el flujo de control simplemente se reanuda en el llamador actual.

Anotación

Existe algún tipo de comportamiento específico de Microsoft de Async Sub.

Si SynchronizationContext.Current es Nothing al inicio de la invocación, las excepciones no controladas de Async Sub se enviarán a Threadpool.

Si SynchronizationContext.Current no es Nothing al inicio de la invocación, OperationStarted() se invoca en ese contexto antes del inicio del método y OperationCompleted() tras el final. Además, las excepciones no controladas se enviarán para reproducirse en el contexto de sincronización.

Esto significa que, en las aplicaciones de interfaz de usuario, las excepciones no detectadas por cualquier elemento Async Sub invocado en el subproceso de la interfaz de usuario, volverán a enviarse al subproceso de dicha interfaz.

Anotación

En general, las estructuras mutables se consideran una práctica incorrecta no compatible con los métodos asincrónicos o iterator. En concreto, cada una de las invocaciones de un método asincrónico o iterator en una estructura actuará implícitamente en una *copia* de dicha estructura que se copia en el momento de invocación. Así, por ejemplo,

Structure S  
 Dim x As Integer  
 Async Sub Mutate()  
 x = 2  
 End Sub  
 End Structure

Dim s As New S With {.x = 1}  
 s.Mutate()  
 Console.WriteLine(s.x) ’ prints “1”

### Bloques y etiquetas

Un grupo de instrucciones ejecutables se denomina bloque de instrucciones. La ejecución de un bloque de instrucciones comienza por la primera instrucción del bloque. Una vez que la instrucción se ha ejecutado, se ejecuta la siguiente en orden léxico, salvo que una instrucción transfiera la ejecución a otro lugar o se produzca una excepción.

Dentro de un bloque de instrucciones, la división de las instrucciones en líneas lógicas no es significativa a excepción de las instrucciones de declaración de etiquetas. Una etiqueta es un identificador que identifica una posición concreta dentro del bloque de instrucciones que se puede usar como el destino de una instrucción de bifurcación como GoTo.

Las instrucciones de declaración de etiquetas debe aparecer al comienzo de una línea lógica y las etiquetas deben ser un identificador o un literal de número entero. Como tanto las instrucciones de declaración de etiquetas como las instrucciones de llamada pueden constar de un único identificador, el identificador del comienzo de una línea local siempre se considera una instrucción de declaración de etiquetas. Las instrucciones de declaración de etiquetas siempre deben ir seguidas por un punto y coma, aunque no haya más instrucciones en la misma línea lógica.

Las etiquetas tienen su propio espacio de declaración y no interfieren con otros identificadores. El ejemplo siguiente es válido y usa la variable de nombre x como un parámetro y como una etiqueta.

Function F(x As Integer) As Integer  
 If x >= 0 Then  
 GoTo x  
 End If  
 x = -x  
x:   
 Return x  
End Function

El ámbito de una etiqueta es el cuerpo del método que la contiene.

Por cuestiones de legibilidad, las producciones de instrucciones que implican varias subinstrucciones se tratan como una producción única en esta especificación, aunque las subinstrucciones puedan estar cada una en una línea etiquetada.

Block ::= [ Statements+ ]

LabelDeclarationStatement ::= LabelName :

LabelName ::= Identifier | IntLiteral

Statements ::=  
 [ Statement ] |  
 Statements : [ Statement ]

### Parámetros y variables locales

En las secciones 10.1.1-10.1.3 se detalla cómo y cuándo se crean las instancias de método y, con ellas, las copias de las variables y parámetros locales de un método. Además, cada vez que se entra en el cuerpo de un bucle, se crea una nueva copia de cada una de las variables locales declaradas dentro de dicho bucle, según lo descrito en la sección 10.9, y la instancia de método incluye entonces esta copia de su variable local en lugar de la copia anterior.

Todas las variables locales se inicializan en el valor predeterminado de su tipo. Los parámetros y las variables locales son siempre públicamente accesibles. Es un error hacer referencia a una variable local en una posición textual anterior a su declaración, como se ilustra en este ejemplo:

Class A  
 Private i As Integer = 0  
  
 Sub F()  
 i = 1  
 Dim i As Integer ' Error, use precedes declaration.  
 i = 2  
 End Sub  
  
 Sub G()  
 Dim a As Integer = 1  
 Dim b As Integer = a ' This is valid.  
 End Sub  
End Class

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 y es un error porque precede textualmente a la declaración de la variable. En el método G, una declaración de variable posterior hace referencia a una variable local declarada en una declaración de variable anterior dentro de la misma declaración de variable local.

Cada bloque de un método crea un espacio de declaración para variables locales. Los nombres se introducen en este espacio de declaración a través de las declaraciones de variables locales en el bloque del método y a través de la lista de parámetros del método, que introduce los nombres en el espacio de declaración del bloque más externo. Los bloques no permiten ocultar nombres a través de la anidación: una vez que el nombre se ha declarado en un bloque, no se puede volver a declarar en ningún bloque anidado.

De este modo, en el siguiente ejemplo, los métodos F y G producen errores 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 ambas i se declaran en bloques independientes no anidados.

Class A  
 Sub F()  
 Dim i As Integer = 0  
 If True Then  
 Dim i As Integer = 1  
 End If  
 End Sub  
  
 Sub G()  
 If True Then  
 Dim i As Integer = 0  
 End If  
 Dim i As Integer = 1  
 End Sub   
  
 Sub H()  
 If True Then  
 Dim i As Integer = 0  
 End If  
 If True Then  
 Dim i As Integer = 1  
 End If  
 End Sub  
  
 Sub I()   
 For i As Integer = 0 To 9  
 H()  
 Next i  
   
 For i As Integer = 0 To 9  
 H()  
 Next i  
 End Sub   
End Class

Cuando el método es una función, una variable local especial que se declara implícitamente en el espacio de declaración del cuerpo del método con el mismo nombre que el método representa el valor devuelto de la función. La variable local tiene una semántica de resolución de nombres especial cuando se usa en expresiones. Si la variable local se usa en un contexto que espera una expresión que se clasifica como un grupo de métodos, como una expresión de invocación, entonces el nombre se resuelve como la función y no como la variable local. Por ejemplo:

Function F(i As Integer) As Integer  
 If i = 0 Then  
 F = 1 ' Sets the return value.  
 Else  
 F = F(i - 1) ' Recursive call.  
 End If  
End Function

El uso de los paréntesis puede dar lugar a situaciones ambiguas (como F(1), donde F es una función cuyo tipo devuelto es una matriz unidimensional); en todas las situaciones ambiguas, los nombres se resuelven como la función, no como la variable local. Por ejemplo:

Function F(i As Integer) As Integer()  
 If i = 0 Then  
 F = new Integer(2) { 1, 2, 3 }  
 Else  
 F = F(i - 1) ' Recursive call, not an index.  
 End If  
End Function

Cuando el flujo de control deja el cuerpo del método, el valor de la variable local vuelve a la expresión de llamada. Si el método es una subrutina, no hay tal variable local implícita y el control simplemente vuelve a la expresión de llamada.

## Instrucciones de declaración local

Una instrucción de declaración local declara una variable local, constante local o variable estática nueva. Las variables locales y las constantes locales son equivalentes a las constantes y variables de instancia cuyo ámbito es el método y se declaran del mismo modo. Las variables estáticas son similares a las variables Shared y se declaran mediante el modificador Static.

Las variables estáticas son locales que conservan su valor en todas las llamadas al método. Las variables estáticas declaradas en métodos no compartidos son por instancia: cada instancia del tipo que contiene el método tiene su propia copia de la variable estática. Las variables estáticas declaradas en métodos Shared son por tipo; solo hay una copia de la variable estática para todas las instancias. Mientras que las variables locales se inicializan en el valor predeterminado de su tipo cuando entran en el método, las variables estáticas solo se inicializan en el valor predeterminado de su tipo cuando se inicializa el tipo o la instancia del tipo. Las variables estáticas no se pueden declarar en estructuras o métodos genéricos.

Las variables locales, las constantes locales y las variables estáticas siempre tienen accesibilidad pública y no pueden especificar modificadores de accesibilidad. Si no se especifica un tipo en una instrucción de declaración local, los pasos siguientes determinan cuál es el tipo de la declaración local:

Si la declaración tiene un carácter de tipo, el tipo del carácter de tipo es el tipo de la declaración local.

Si se utiliza la inferencia de tipos de variable local y la declaración local es una constante local o una variable local con un inicializador, se infiere el tipo de la declaración local del tipo del inicializador. Si el inicializador hace referencia a la declaración local, se produce un error en tiempo de compilación. (Las constantes locales deben tener inicializadores).

Si no se usa semántica estricta, el tipo de la instrucción de la declaración local es implícitamente Object.

En caso contrario, se producirá un error en tiempo de compilación.

Si no se especifica ningún tipo en una instrucción de declaración local que tenga un modificador de tamaño de matriz o de tipo de matriz, entonces el tipo de la declaración local es una matriz con el rango especificado y los pasos anteriores sirven para determinar el tipo de elemento de la matriz. Si se usa la inferencia de tipos de variable local, el tipo del inicializador debe ser un tipo de matriz con la misma forma de matriz que la instrucción de declaración local (es decir, modificadores de tipo de matriz). Obsérvese que es posible que el tipo del elemento inferido siga siendo un tipo de matriz. Por ejemplo:

Option Infer On  
  
Module Test  
 Sub Main()  
 ' Error: initializer is not an array type  
 Dim x() = 1  
  
 ' Type is Integer()  
 Dim y() = New Integer() {}  
  
 ' Type is Integer()()  
 Dim z() = New Integer()() {}  
  
 ' Type is Integer()()()

Dim a()() = New Integer()()() {}  
  
 ' Error: initializer does not have same array shape  
 Dim b()() = New Integer(,)() {}  
 End Sub  
End Module

Si no se especifica ningún tipo en una instrucción de declaración local que tiene un modificador de tipo que admite valores null, el tipo de la declaración local es la versión que admite valores null del tipo inferido o el tipo inferido propiamente si ya es un tipo de valor que admite valores null. Si el tipo inferido no es un tipo de valor que se pueda hacer que admita valores null, se produce un error en tiempo de compilación. Si en la instrucción de declaración local hay tanto un modificador de tipo de matriz o de tamaño de matriz como un modificador de tipo que admite valores null, entonces modificador del tipo que admite valores null se considera que se aplica al tipo de elemento de la matriz y los pasos anteriores sirven para determinar el tipo de elemento.

Los inicializadores de variable en las instrucciones de declaración local equivalen a las instrucciones de asignación situadas en la ubicación textual de la declaración. Por tanto, si la ejecución se bifurca en la instrucción de declaración local, no se ejecuta el inicializador de variable. Si la instrucción de declaración local se ejecuta más de una vez, el inicializador de variable se ejecutará el mismo número de veces. Las variables estáticas solo ejecutan su inicializador la primera vez. Si se produce una excepción durante la inicialización de una variable estática, se considera que está inicializada con el valor predeterminado del tipo de la variable estática.

En el siguiente ejemplo se muestra el uso de los inicializadores.

Module Test  
 Sub F()  
 Static x As Integer = 5  
  
 Console.WriteLine("Static variable x = " & x)  
 x += 1  
 End Sub  
  
 Sub Main()  
 Dim i As Integer  
  
 For i = 1 to 3  
 F()  
 Next i  
  
 i = 3  
label:  
 Dim y As Integer = 8  
  
 If i > 0 Then  
 Console.WriteLine("Local variable y = " & y)  
 y -= 1  
 i -= 1  
 GoTo label  
 End If  
 End Sub  
End Module

Este programa imprime:

Static variable x = 5  
Static variable x = 6  
Static variable x = 7  
Local variable y = 8  
Local variable y = 8  
Local variable y = 8

Los inicializadores en propiedades locales estáticas son seguros para la ejecución de subprocesos y están protegidos contra excepciones durante la inicialización. Si se produce una excepción durante una inicialización de una variable local estática, tendrá el valor predeterminado y no se inicializará. Un inicializador de variable local estática

Module Test  
 Sub F()  
 Static x As Integer = 5  
 End Sub  
End Module

equivale a

Imports System.Threading  
Imports Microsoft.VisualBasic.CompilerServices  
  
Module Test  
 Class InitFlag  
 Public State As Short  
 End Class  
  
 Private xInitFlag As InitFlag = New InitFlag()  
  
 Sub F()  
 Dim x As Integer  
  
 If xInitFlag.State <> 1 Then  
 Monitor.Enter(xInitFlag)  
 Try  
 If xInitFlag.State = 0 Then  
 xInitFlag.State = 2  
 x = 5  
 Else If xInitFlag.State = 2 Then  
 Throw New IncompleteInitialization()  
 End If  
 Finally  
 xInitFlag.State = 1  
 Monitor.Exit(xInitFlag)  
 End Try  
 End If  
 End Sub  
End Module

Las variables locales, las constantes locales y las variables estáticas tienen como ámbito el bloque de la instrucción en el que se declaran. Las variables estáticas son especiales porque su nombre solo se puede usar una vez a lo largo de todo el método. Por ejemplo, no es válido especificar dos declaraciones de variables estáticas con el mismo nombre aunque se encuentren en bloques diferentes.

LocalDeclarationStatement ::= LocalModifier VariableDeclarators StatementTerminator

LocalModifier ::= Static | Dim | Const

### Declaraciones locales implícitas

Además de las instrucciones de declaraciones locales, las variables locales también se pueden declarar de forma implícita mediante el uso. Una expresión de nombre simple que usa un nombre que no se resuelve como algo distinto declara una variable local por ese nombre. Por ejemplo:

Option Explicit Off  
  
Module Test  
 Sub Main()  
 x = 10  
 y = 20  
 Console.WriteLine(x + y)  
 End Sub  
End Module

La declaración local implícita solo tiene lugar en contextos de expresión que pueden aceptar una expresión clasificada como variable. La excepción a esta regla es que una variable local no puede declararse de forma implícita cuando es el destino de una expresión de invocación de función, una expresión de indizado o una expresión de acceso a miembro.

Las variables locales implícitas se tratan como si se hubieran declarado al comienzo del método que las contiene. Por tanto, su ámbito es siempre el cuerpo del método completo, aunque se declaren dentro de una expresión lambda. Por ejemplo, el código siguiente:

Option Explicit Off   
  
Module Test  
 Sub Main()  
 Dim x = Sub()  
 a = 10  
 End Sub  
 Dim y = Sub()  
 Console.WriteLine(a)  
 End Sub  
  
 x()  
 y()  
 End Sub  
End Module

imprimirá el valor 10. Las variables locales implícitas tienen el tipo Object si no se ha adjuntado ningún carácter de tipo al nombre de la variable; de lo contrario, su tipo es el tipo del carácter de tipo. Con las variables locales implícitas no se emplea la inferencia de tipos.

Si la declaración local explícita la especifica el entorno de compilación u Option Explicit, todas las variables locales deben declararse explícitamente y no se permite la declaración implícita.

## Instrucción With

Una instrucción With permite múltiples referencias a los miembros de una expresión sin especificar la expresión múltiples veces. La expresión debe clasificarse como un valor y se evalúa una vez, cuando entra en el bloque. Dentro del bloque de la instrucción With, una expresión de acceso a miembros o una expresión de acceso de diccionario que comienza por un punto o un signo de exclamación se evalúa como si estuviera precedida por la expresión With. Por ejemplo:

Structure Test  
 Public x As Integer  
  
 Function F() As Integer  
 Return 10  
 End Sub  
End Structure  
  
Module TestModule  
 Sub Main()  
 Dim y As Test  
  
 With y  
 .x = 10  
 Console.WriteLine(.x)  
 .x = .F()  
 End With  
 End Sub  
End Module

No es válido hacer una bifurcación a un bloque de instrucciones With desde fuera del bloque.

WithStatement ::=  
 With Expression StatementTerminator  
 [ Block ]  
 End With StatementTerminator

## Instrucción SyncLock

Una instrucción SyncLock permite sincronizar instrucciones en una expresión, lo que garantiza que varios subprocesos no ejecuten las mismas instrucciones al mismo tiempo. La expresión debe clasificarse como un valor y se evalúa una vez, cuando entra en el bloque. Cuando entra en el bloque SyncLock, el método System.Threading.Monitor.Enter (Shared) se invoca en la expresión especificada, que bloquea hasta que el subproceso de ejecución tenga un bloqueo exclusivo en el objeto que la expresión devuelve. El tipo de la expresión de una instrucción SyncLock debe ser un tipo de referencia. Por ejemplo:

Class Test  
 Private count As Integer = 0  
  
 Public Function Add() As Integer  
 SyncLock Me  
 count += 1  
 Add = count  
 End SyncLock  
 End Function  
  
 Public Function Subtract() As Integer  
 SyncLock Me  
 count -= 1  
 Subtract = count  
 End SyncLock  
 End Function  
End Class

En el ejemplo anterior se sincroniza en la instancia específica de la clase Test para asegurar que no se pueda sumar ni restar más de un subproceso de ejecución de la variable de contador cada vez para una instancia concreta.

El bloque SyncLock está contenido implícitamente en una instrucción Try cuyo bloque Finally llama al método System.Threading.Monitor.Exit (Shared) en la expresión. Así se asegura de que se libera el bloqueo incluso si se produce una excepción. Como resultado, no es válido hacer una bifurcación a un bloque SyncLock desde fuera del bloque, y un bloque SyncLock se trata como una instrucción única para los fines de Resume y Resume Next. El ejemplo anterior es equivalente al código siguiente:

Class Test  
 Private count As Integer = 0  
  
 Public Function Add() As Integer  
 Try  
 System.Threading.Monitor.Enter(Me)  
  
 count += 1  
 Add = count  
 Finally  
 System.Threading.Monitor.Exit(Me)  
 End Try  
 End Function  
  
 Public Function Subtract() As Integer  
 Try  
 System.Threading.Monitor.Enter(Me)  
  
 count -= 1  
 Subtract = count  
 Finally  
 System.Threading.Monitor.Exit(Me)  
 End Try  
 End Function  
End Class

SyncLockStatement ::=  
 SyncLock Expression StatementTerminator  
 [ Block ]  
 End SyncLock StatementTerminator

## Instrucciones Event

Las instrucciones RaiseEvent, AddHandler y RemoveHandler generan eventos y los controlan de forma dinámica.

EventStatement ::=  
 RaiseEventStatement |  
 AddHandlerStatement |  
 RemoveHandlerStatement

### Instrucción RaiseEvent

Una instrucción RaiseEvent notifica a los controladores de eventos que se ha generado un evento concreto. La expresión de nombre simple de una instrucción RaiseEvent se interpreta como una búsqueda de miembros en Me. Por consiguiente, RaiseEvent x se interpreta como si fuera RaiseEvent Me.x. El resultado de la expresión debe clasificarse como un acceso de evento para un evento definido en la clase; los eventos definidos en tipos base no se pueden usar como instrucciones RaiseEvent.

La instrucción RaiseEvent se procesa como una llamada al método Invoke del delegado del evento mediante los parámetros proporcionados, si los hay. Si el valor del delegado es Nothing, no se produce ninguna excepción. Si no hay argumentos, pueden omitirse los paréntesis. Por ejemplo:

Class Raiser  
 Public Event E1(Count As Integer)  
  
 Public Sub Raise()  
 Static RaiseCount As Integer = 0  
  
 RaiseCount += 1  
 RaiseEvent E1(RaiseCount)  
 End Sub  
End Class  
  
Module Test  
 Private WithEvents x As Raiser  
  
 Private Sub E1Handler(Count As Integer) Handles x.E1  
 Console.WriteLine("Raise #" & Count)  
 End Sub  
  
 Public Sub Main()  
 x = New Raiser  
 x.Raise() ' Prints "Raise #1".  
 x.Raise() ' Prints "Raise #2".  
 x.Raise() ' Prints "Raise #3".  
 End Sub  
End Module

La clase Raiser anterior es equivalente a:

Class Raiser  
 Public Event E1(Count As Integer)  
  
 Public Sub Raise()  
 Static RaiseCount As Integer = 0  
 Dim TemporaryDelegate As E1EventHandler  
  
 RaiseCount += 1  
  
 ' Use a temporary to avoid a race condition.  
 TemporaryDelegate = E1Event  
 If Not TemporaryDelegate Is Nothing Then  
 TemporaryDelegate.Invoke(RaiseCount)  
 End If  
 End Sub  
End Class

RaiseEventStatement ::= RaiseEvent IdentifierOrKeyword [ OpenParenthesis  
 [ ArgumentList ] CloseParenthesis ] StatementTerminator

### Instrucciones AddHandler y RemoveHandler

Aunque la mayoría de los controladores de eventos se enlazan automáticamente mediante variables WithEvents, puede ser necesario agregar y quitar controladores de eventos dinámicamente en tiempo de ejecución. Las instrucciones AddHandler y RemoveHandler hacen eso.

Cada instrucción toma dos argumentos: el primero debe ser una expresión clasificada como un acceso de evento y el segundo debe ser una expresión clasificada como valor. El tipo del segundo argumento debe ser el tipo del delegado asociado con el acceso del evento. Por ejemplo:

Public Class Form1  
 Public Sub New()  
 ' Add Button1\_Click as an event handler for Button1's Click event.  
 AddHandler Button1.Click, AddressOf Button1\_Click  
 End Sub   
  
 Private Button1 As Button = New Button()  
  
 Sub Button1\_Click(sender As Object, e As EventArgs)  
 Console.WriteLine("Button1 was clicked!")  
 End Sub  
  
 Public Sub Disconnect()  
 RemoveHandler Button1.Click, AddressOf Button1\_Click  
 End Sub  
End Class

Dado un evento E, la instrucción llama al método add\_E o remove\_E pertinente en la instancia para agregar o quitar el delegado como un controlador para el evento. Así, el código anterior es equivalente a:

Public Class Form1  
 Public Sub New()  
 Button1.add\_Click(AddressOf Button1\_Click)  
 End Sub   
  
 Private Button1 As Button = New Button()  
  
 Sub Button1\_Click(sender As Object, e As EventArgs)  
 Console.WriteLine("Button1 was clicked!")  
 End Sub  
  
 Public Sub Disconnect()  
 Button1.remove\_Click(AddressOf Button1\_Click)  
 End Sub  
End Class

AddHandlerStatement ::= AddHandler Expression Comma Expression StatementTerminator

RemoveHandlerStatement ::= RemoveHandler Expression Comma Expression StatementTerminator

## Instrucciones de asignación

Una instrucción de asignación asigna el valor de una expresión a una variable. Hay tres tipos de asignaciones.

AssignmentStatement ::=  
 RegularAssignmentStatement |  
 CompoundAssignmentStatement |  
 MidAssignmentStatement

### Instrucciones de asignación regular

Una instrucción de asignación simple almacena el resultado de una expresión en una variable. La expresión del lado izquierdo del operador de asignación debe clasificarse como una variable o un acceso de propiedad, mientras que la expresión del lado derecho debe clasificarse como valor. El tipo de la expresión debe poder convertirse implícitamente en el tipo de la variable o acceso de propiedad.

Si la variable a la que se está asignando es un elemento de matriz de un tipo de referencia, se hará una comprobación en tiempo de ejecución para garantizar que la expresión es compatible con el tipo de elemento de matriz. En el ejemplo siguiente, 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 una matriz String.

Dim sa(10) As String  
Dim oa As Object() = sa  
oa(0) = Nothing ' This is allowed.  
oa(1) = "Hello" ' This is allowed.  
oa(2) = New ArrayList() ' System.ArrayTypeMismatchException is thrown.

Si la expresión de la izquierda del operador de asignación se clasifica como variable, la instrucción de asignación guarda el valor en la variable. Si la expresión se clasifica como un acceso de propiedad, la instrucción de asignación convierte el acceso de propiedad en una invocación del descriptor de acceso Set de la propiedad con el valor sustituido para el parámetro de valor. Por ejemplo:

Module Test  
 Private PValue As Integer  
  
 Public Property P As Integer  
 Get  
 Return PValue  
 End Get  
  
 Set (Value As Integer)  
 PValue = Value  
 End Set  
 End Property  
  
 Sub Main()  
 ' The following two lines are equivalent.  
 P = 10  
 set\_P(10)  
 End Sub  
End Module

Si el destino de la variable o acceso de propiedad tiene como tipo un tipo de valor pero no se clasifica como variable, se produce un error en tiempo de compilación. Por ejemplo:

Structure S  
 Public F As Integer  
End Structure  
  
Class C  
 Private PValue As S  
  
 Public Property P As S  
 Get  
 Return PValue  
 End Get  
  
 Set (Value As S)  
 PValue = Value  
 End Set  
 End Property  
End Class  
  
Module Test  
 Sub Main()  
 Dim ct As C = New C()  
 Dim rt As Object = new C()  
  
 ' Compile-time error: ct.P not classified as variable.  
 ct.P.F = 10  
  
 ' Run-time exception.  
 rt.P.F = 10  
 End Sub  
End Module

Nota   La semántica de la asignación depende del tipo de la variable o propiedad a la que está siendo asignada. Si la variable a la que se está asignando es un tipo de valor, la asignación copia el valor de la expresión en la variable. Si la variable a la que se está asignando es un tipo de referencia, la asignación copia la referencia, no el valor, en la variable. Si el tipo de la variable es Object, la semántica de asignación se determina en función de si el tipo del valor es un tipo de valor o de referencia en tiempo de ejecución.

Anotación

Para tipos intrínsecos como Integer y Date, la semántica de asignación de valores y referencias es la misma porque los tipos son inmutables. Como resultado, el lenguaje es libre de usar la asignación de referencia en tipos intrínsecos con conversión boxing como una optimización. Desde la perspectiva de los valores, el resultado es el mismo.

Como el carácter igual (=) se usa para la asignación y para la igualdad, hay ambigüedad entre una asignación simple y una instrucción de llamada en una situación como x = y.ToString(). En estos casos, la instrucción de asignación tiene preferencia sobre el operador de igualdad. Esto significa que la expresión de ejemplo se interpreta como x = (y.ToString()) en lugar de como (x = y).ToString().

RegularAssignmentStatement ::= Expression Equals Expression StatementTerminator

### Instrucciones de asignación compuesta

Una instrucción de asignación compuesta adopta la forma V OP= E (donde OP es un operador binario válido). La expresión de la izquierda del operador de asignación debe clasificarse como una variable o un acceso de propiedad, mientras que la expresión de la derecha debe clasificarse como valor. La instrucción de asignación compuesta es equivalente a la instrucción V = V OP E con la diferencia de que la variable de la izquierda del operador de asignación compuesta solo se evalúa una vez. En el siguiente ejemplo se muestra esta diferencia:

Module Test  
 Function GetIndex() As Integer  
 Console.WriteLine("Getting index")  
 Return 1  
 End Function  
  
 Sub Main()  
 Dim a(2) As Integer  
  
 Console.WriteLine("Simple assignment")  
 a(GetIndex()) = a(GetIndex()) + 1  
  
 Console.WriteLine("Compound assignment")  
 a(GetIndex()) += 1  
 End Sub  
End Module

La expresión a(GetIndex()) se evalúa dos veces para una asignación simple pero solo una vez para una asignación compuesta, de manera que el código imprime:

Simple assignment  
Getting index  
Getting index  
Compound assignment  
Getting index

CompoundAssignmentStatement ::= Expression CompoundBinaryOperator [ LineTerminator ]  
 Expression StatementTerminator

CompoundBinaryOperator ::= ^ = | \* = | / = | \ = | + = | - = | & = | < < = | > > =

### Instrucciones de asignación Mid

Una instrucción de asignación Mid asigna una cadena a otra cadena. La parte izquierda de la asignación tiene la misma sintaxis que una llamada a la función Microsoft.VisualBasic.Strings.Mid. El primer argumento es el destino de la asignación y debe clasificarse como una variable o un acceso de propiedad cuyo tipo sea implícitamente convertible de y en String. El segundo parámetro es la posición de inicio de base 1 que corresponde con el lugar en el que debe comenzar la asignación en la cadena de destino y debe clasificarse como un valor cuyo tipo debe ser implícitamente convertible en Integer. El tercer parámetro opcional es el número de caracteres a partir del valor de la derecha que se asignan a la cadena de destino y debe clasificarse como un valor cuyo tipo sea implícitamente convertible en Integer. El lado derecho es la cadena de origen y debe clasificarse como un valor cuyo tipo es implícitamente convertible en String. El lado derecho se trunca de acuerdo con el parámetro de longitud, si se especifica, y reemplaza los caracteres de la cadena de la izquierda, comenzando por la posición inicial. Si la cadena del lado derecho contenía menos caracteres que el tercer parámetro, solo se copiarán los caracteres de la cadena de la derecha.

El ejemplo siguiente muestra ab123fg:

Module Test  
 Sub Main()  
 Dim s1 As String = "abcdefg"  
 Dim s2 As String = "1234567"  
  
 Mid$(s1, 3, 3) = s2  
 Console.WriteLine(s1)  
 End Sub  
End Module

Nota   Mid no es una palabra reservada.

MidAssignmentStatement ::=  
 Mid [ $ ] OpenParenthesis Expression Comma Expression [ Comma Expression ] CloseParenthesis  
 Equals Expression StatementTerminator

## Instrucciones de invocación

Una instrucción de invocación llama a un método precedido por la palabra clave opcional Call. Una instrucción de invocación se procesa igual que una expresión de invocación de función, con algunas diferencias que se indican a continuación. La expresión de invocación debe clasificarse como un valor o como vacío. Se descarta cualquier valor que resulte de evaluar la expresión de invocación.

Si la palabra clave Call se omite, la expresión de invocación debe comenzar por un identificador o palabra clave, o por . dentro de un bloque With. Así, por ejemplo, "Call 1.ToString()" es una instrucción válida, pero "1.ToString()" no lo es. (Asimismo, observe que en un contexto de expresión, las expresiones de invocación no deben comenzar por un identificador. Por ejemplo, "Dim x = 1.ToString()" es una instrucción válida).

Existe otra diferencia entre las instrucciones de invocación y las expresiones de invocación: si una instrucción de invocación incluye una lista de argumentos, esta siempre se toma como la lista de argumentos de la invocación. En el ejemplo siguiente se ilustra la diferencia:

Module Test  
 Sub Main()  
 Call {Function() 15}(0)  
 ' error: (0) is taken as argument list, but array is not invokable  
  
 Call ({Function() 15}(0))  
 ' valid, since the invocation statement has no argument list  
  
 Dim x = {Function() 15}(0)  
 ' valid as an expression, since (0) is taken as an array-indexing  
  
 Call f("a")  
 ' error: ("a") is taken as argument list to the invocation of f  
  
 Call f()("a")  
 ' valid, since () is the argument list for the invocation of f  
  
 Dim y = f("a")  
 ' valid as an expression, since f("a") is interpreted as f()("a")  
 End Sub  
  
 Sub f() As Func(Of String,String)  
 Return Function(x) x  
 End Sub  
End Module

InvocationStatement ::= [ Call ] InvocationExpression StatementTerminator

## Instrucciones condicionales

Las instrucciones condicionales permiten la ejecución condicional de las instrucciones basada en la evaluación de expresiones en tiempo de ejecución.

ConditionalStatement ::= IfStatement | SelectStatement

### Instrucciones If...Then...Else

Una instrucción If...Then...Else es la instrucción condicional básica. Cada expresión de una instrucción If...Then...Else debe ser una expresión booleana, según la sección 11.19. (Nota: esto no precisa que la expresión sea de tipo booleano). Si la expresión de la instrucción If es verdadera, se ejecutan las instrucciones incluidas en el bloque If. Si la expresión es falsa, se evalúa cada expresión ElseIf. Si una de las expresiones ElseIf se evalúa como verdadera, se ejecuta el bloque correspondiente. Si ninguna de las expresiones se evalúa como verdadera y si hay un bloque Else, se ejecuta el bloque Else. Una vez que el bloque termina, la ejecución pasa al final de la instrucción If...Then...Else.

La versión de línea de la instrucción If tiene un único conjunto de instrucciones para ejecutarse si la expresión If es True y un conjunto opcional de instrucciones que se ejecutarán si es False. Por ejemplo:

Module Test  
 Sub Main()  
 Dim a As Integer = 10  
 Dim b As Integer = 20  
  
 ' Block If statement.  
 If a < b Then  
 a = b  
 Else  
 b = a  
 End If  
  
 ' Line If statement  
 If a < b Then a = b Else b = a  
 End Sub  
End Module

IfStatement ::= BlockIfStatement | LineIfThenStatement

BlockIfStatement ::=  
 If BooleanExpression [ Then ] StatementTerminator  
 [ Block ]  
 [ ElseIfStatement+ ]  
 [ ElseStatement ]  
 End If StatementTerminator

ElseIfStatement ::=  
 ElseIf BooleanExpression [ Then ] StatementTerminator  
 [ Block ]

ElseStatement ::=  
 Else StatementTerminator  
 [ Block ]

LineIfThenStatement ::=  
 If BooleanExpression Then Statements [ Else Statements ] StatementTerminator

La versión de línea de la instrucción If enlaza con menos presión que “:”, y su Else enlaza con el If precedente más cercano que permita la sintaxis. Por ejemplo, las dos versiones siguientes son equivalentes:

If True Then \_  
If True Then Console.WriteLine("a") Else Console.WriteLine("b") \_  
Else Console.WriteLine("c") : Console.WriteLine("d")  
  
If True Then  
 If True Then  
 Console.WriteLine("a")  
 Else  
 Console.WriteLine("b")  
 End If  
 Console.WriteLine("c") : Console.WriteLine("d")  
End If

En una instrucción de línea If se permiten todas las instrucciones que no sean instrucciones de declaración de etiqueta, incluidas las instrucciones de bloque. Sin embargo, no pueden usar LineTerminators como StatementTerminators salvo dentro de expresiones lambda multilínea. Por ejemplo:

' Allowed, since it uses : instead of LineTerminator to separate statements  
If b Then With New String("a"(0),5) : Console.WriteLine(.Length) : End With  
  
' Disallowed, since it uses a LineTerminator  
If b then With New String("a"(0), 5)  
 Console.WriteLine(.Length)  
 End With

' Allowed, since it only uses LineTerminator inside a multi-line lambda  
If b Then Call Sub()  
 Console.WriteLine("a")  
 End Sub.Invoke()

### Instrucciones Select...Case

Una instrucción Select Case ejecuta instrucciones basándose en el valor de una expresión. La expresión debe clasificarse como valor. Cuando se ejecuta una instrucción Select Case, la expresión Select se evalúa en primer lugar y la instrucción Case se evalúa en el orden de la declaración textual. La primera instrucción Case que se evalúa como True ejecuta su bloque. Si ninguna de las instrucciones Case se evalúa como True y si hay una instrucción Case Else, se ejecuta ese bloque. Una vez que el bloque termina, la ejecución pasa al final de la instrucción Select.

La ejecución de un bloque Case no tiene permitido el "paso explícito" a la siguiente sección switch. Esto evita que se produzcan una serie de errores comunes en otros lenguajes cuando se omite accidentalmente una instrucción Case de finalización. En el siguiente ejemplo se muestra este comportamiento:

Module Test  
 Sub Main()  
 Dim x As Integer = 10  
  
 Select Case x  
 Case 5  
 Console.WriteLine("x = 5")  
 Case 10  
 Console.WriteLine("x = 10")  
 Case 20 - 10  
 Console.WriteLine("x = 20 - 10")  
 Case 30  
 Console.WriteLine("x = 30")  
 End Select  
 End Sub  
End Module

El código imprime:

x = 10

Aunque Case 10 y Case 20 - 10 seleccionan para el mismo valor, Case 10 se ejecuta porque precede a Case 20 - 10 textualmente. Cuando se alcanza la siguiente Case, continúa la ejecución después de la instrucción Select.

Una cláusula Case puede adoptar dos formas. Una es una palabra clave Is opcional, un operador de comparación, y una expresión. La expresión se convierte en el tipo de la expresión Select; si la expresión no es implícitamente convertible en el tipo de la expresión Select, se produce un error en tiempo de compilación. Si la expresión Select es E, el operador de comparación es Op y la expresión Case es E1, se evalúa como E OP E1. El operador debe ser válido para los tipos de las dos expresiones; de lo contrario se produce un error en tiempo de compilación.

La otra forma es una expresión opcionalmente seguida de la palabra clave To y una segunda expresión. Ambas expresiones se convierten en el tipo de la expresión Select; si alguna expresión no es implícitamente convertible en el tipo de la expresión Select, se produce un error en tiempo de compilación. Si la expresión Select es E, la primera expresión Case es E1 y la segunda expresión Case es E2, Case se evalúa bien como E = E1 (si E2 no se especifica) o (E >= E1) y (E <= E2). Los operadores deben ser válidos para los tipos de las dos expresiones; de lo contrario se produce un error en tiempo de compilación.

SelectStatement ::=  
 Select [ Case ] Expression StatementTerminator  
 [ CaseStatement+ ]  
 [ CaseElseStatement ]  
 End Select StatementTerminator

CaseStatement ::=  
 Case CaseClauses StatementTerminator  
 [ Block ]

CaseClauses ::=  
 CaseClause |  
 CaseClauses Comma CaseClause

CaseClause ::=  
 [ Is [ LineTerminator ] ] ComparisonOperator [ LineTerminator ] Expression |  
 Expression [ To Expression ]

ComparisonOperator ::= = | < > | < | > | > = | < =

CaseElseStatement ::=  
 Case Else StatementTerminator  
 [ Block ]

## Instrucciones de bucle

Las instrucciones de bucle permiten la ejecución repetida de las instrucciones en su cuerpo.

Cada vez que se entra en el cuerpo de un bucle, se crea una copia reciente de todas las variables locales declaradas en dicho cuerpo, inicializada en los valores anteriores de las variables. Cualquier referencia a una variable dentro del cuerpo del bucle usará la copia creada más recientemente. Este código muestra un ejemplo:

Module Test  
 Sub Main()  
 Dim lambdas As New List(Of Action)  
 Dim x = 1

For i = 1 To 3  
 x = i  
 Dim y = x  
 lambdas.Add(Sub() Console.WriteLine(x & y))  
 Next

For Each lambda In lambdas  
 lambda()  
 Next  
 End Sub  
End Module

El código produce el resultado:

31 32 33

Cuando se ejecuta el cuerpo del bucle, usa la copia de la variable que sea actual. Por ejemplo, la instrucción Dim y = x hace referencia a la copia más reciente de y y la copia original de x. Además, cuando se crea una expresión lambda, recuerda la copia de una variable que era actual en el momento de su creación. Por tanto, cada lambda usa la misma copia compartida de x, pero una copia diferente de y. Al final del programa, cuando ejecuta las expresiones lambda, esa copia compartida de x a la que todas hacen referencia está ahora en su valor final 3.

Note that if there are no lambdas or LINQ expressions, then it's impossible to know that a fresh copy is made on loop entry. Indeed, compiler optimizations will avoid making copies in this case. Note too that it's illegal to GoTo into a loop that contains lambdas or LINQ expressions.LoopStatement ::=  
 WhileStatement |  
 DoLoopStatement |  
 ForStatement |  
 ForEachStatement

### Instrucciones While...End While y Do...Loop

Una instrucción de bucle While o Do funciona basándose en una expresión booleana. Una instrucción de bucle While mantiene el bucle siempre que la expresión booleana se evalúe como verdadera; una instrucción de bucle Do puede contener una condición más compleja. Una expresión puede ir después de la palabra clave Do o Loop, pero no después de ambas. Se evalúa la expresión booleana según la Sección 11.19. (Nota: esto no precisa que la expresión sea de tipo booleano). También es válido no especificar ninguna expresión; en ese caso, el bucle no terminará nunca. Si la expresión se sitúa después de Do, será evaluada antes de que el bloque del bucle se ejecute en cada iteración. Si la expresión se sitúa después de Loop, será evaluada después de que el bloque del bucle se haya ejecutado en cada iteración. Situar la expresión después de Loop generará un bucle más que si va después de Do. En el siguiente ejemplo se muestra este comportamiento:

Module Test  
 Sub Main()  
 Dim x As Integer  
  
 x = 3  
 Do While x = 1  
 Console.WriteLine("First loop")  
 Loop  
  
 Do  
 Console.WriteLine("Second loop")  
 Loop While x = 1  
 End Sub  
End Module

El código produce el resultado:

Second Loop

En el caso del primer bucle, la condición se evalúa antes de que se ejecute el bucle. En el caso del segundo bucle, la condición se ejecuta después de que se ejecute el bucle. La expresión condicional debe llevar el prefijo con una palabra clave While o Until. La primera rompe el bucle si la condición se evalúa como falsa, la segunda cuando la condición se evalúa como verdadera.

Nota   Until no es una palabra reservada.

WhileStatement ::=  
 While BooleanExpression StatementTerminator  
 [ Block ]  
 End While StatementTerminator

DoLoopStatement ::= DoTopLoopStatement | DoBottomLoopStatement

DoTopLoopStatement ::=  
 Do [ WhileOrUntil BooleanExpression ] StatementTerminator  
 [ Block ]  
 Loop StatementTerminator

DoBottomLoopStatement ::=  
 Do StatementTerminator  
 [ Block ]  
 Loop WhileOrUntil BooleanExpression StatementTerminator

WhileOrUntil ::= While | Until

### Instrucciones For...Next

1. Una instrucción For...Next se repite según un conjunto de límites. Una instrucción For especifica una variable de control de bucle, una expresión de límite inferior, una expresión de límite superior y una expresión de valor de paso opcional. La variable de control de bucle se especifica bien mediante un identificador seguido de una cláusula As opcional o una expresión. Según lo establecido en las reglas siguientes, la variable de control de bucle hace referencia a una nueva variable local específica de esta instrucción For...Next, a una variable ya existente o a una expresión. Si la variable de control de bucle es un identificador con una cláusula As, el identificador define una nueva variable local del tipo especificado en la cláusula As, que tiene por ámbito todo el bucle For.
2. Si la variable de control de bucle es un identificador sin una cláusula As, el identificador se resuelve primero utilizando las reglas de resolución de nombre simple (vea la Sección 11.4.4), con la excepción de que esta aparición del identificador no provocaría en y por sí misma la creación de una variable local implícita (sección 10.2.1).
   1. Si esta resolución es correcta y el resultado se clasifica como variable, la variable de control de bucle es esa variable ya existente.
   2. Si la resolución da error, o bien se realiza correctamente y el resultado se clasifica como tipo:
      1. Si se usa la inferencia de tipos de variable local, el identificador define una nueva variable local cuyo tipo se infiere de las expresiones de límite y de paso, cuyo ámbito es el bucle For completo;
      2. Si no se usa la inferencia de tipos de variable local, pero sí la declaración local implícita, se crea una variable local implícita cuyo ámbito es el método completo (sección 10.2.1) y la variable de control de bucle hace referencia a esta variable ya existente;
      3. Si no se usan la inferencia de tipos de variable local ni las declaraciones locales implícitas, es un error.
   3. Si la resolución se realiza correctamente pero el resultado no se clasifica como tipo ni como variable, es un error.
3. Si la variable de control de bucle es una expresión, la expresión se debe clasificar como una variable.

Una variable de control de bucle no puede usarse en otra instrucción For...Next envolvente. El tipo de la variable de control de bucle de una instrucción For determina el tipo de la iteración y debe ser uno de los siguientes:

1. Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long, Decimal, Single, Double
2. Un tipo enumerado
3. Object
4. Un tipo T con los operadores siguientes, donde B es un tipo que puede usarse en una expresión booleana:

Public Shared Operator >= (op1 As T, op2 As T) As B  
Public Shared Operator <= (op1 As T, op2 As T) As B  
Public Shared Operator - (op1 As T, op2 As T) As T  
Public Shared Operator + (op1 As T, op2 As T) As T

Las expresiones de límite y de paso deben poder convertirse implícitamente en el tipo de la variable de control de bucle y deben clasificarse como valores. En tiempo de compilación, el tipo de la variable de control de bucle se infiere eligiendo el tipo más amplio de los tipos de la expresión de paso, de límite superior y de límite inferior. Si no hay conversión de ampliación (widening) entre dos de los tipos, se produce un error en tiempo de compilación.

En tiempo de ejecución, si el tipo de la variable de control de bucle es Object, el tipo de la iteración se infiere igual que en tiempo de compilación, con dos excepciones. Primera, si las expresiones de paso y de límite son todas de tipos enteros pero no tienen tipo más amplio, el tipo más amplio que abarca a los tres tipos se inferirá. Y segunda, si el tipo de la variable de control de bucle se infiere para que sea String, se inferirá Double. Si, en tiempo de ejecución, no se puede determinar el tipo de control de bucle o si alguna de las expresiones no se puede convertir en el tipo de control de bucle, se producirá una System.InvalidCastException. Una vez que el tipo de control de bucle se ha elegido al comienzo del bucle, el mismo tipo se utilizará en toda la iteración, con independencia de los cambios que se hagan en el valor de la variable de control de bucle.

Una instrucción For debe estar seguida de cerca por una instrucción Next emparejada. Una instrucción Next sin una variable coincide con la instrucción For abierta más interna, mientras que una instrucción Next con una o más variables de control de bucle coincidirá, de izquierda a derecha, con los bucles For que coinciden con cada variable. Si una variable coincide con un bucle For que no es el más anidado en ese punto, se produce un error en tiempo de compilación.

Al comienzo del bucle, las tres expresiones se evalúan en orden textual y la expresión de límite inferior se asigna a la variable de control de bucle. Si se omite el valor del paso, es implícitamente el literal 1, convertido en el tipo de la variable de control de bucle. Las tres expresiones solo se evalúan al comienzo del bucle.

Al comienzo de cada bucle, la variable de control se compara para ver si es mayor que el extremo si la expresión de paso es positiva, o menor que el extremo si la expresión de paso es negativa. Si lo es, el bucle For finaliza; de lo contrario, se ejecuta su bloque. Si la variable de control de bucle no es un tipo primitivo, el operador de comparación se determina en función de si la expresión step >= step – step es verdadera o falsa. En la instrucción Next, el valor de paso se agrega a la variable de control y la ejecución vuelve al inicio del bucle.

Observe que *no* se crea una nueva copia de la variable de control de bucle en cada una de las iteraciones del bloque del bucle. En este sentido, la instrucción For se diferencia de For Each (sección 10.9.3).

No es válido hacer una bifurcación en un bucle For desde fuera del bucle.

ForStatement ::=  
 For LoopControlVariable Equals Expression To Expression [ Step Expression ] StatementTerminator  
 [ Block ]  
 [ Next [ NextExpressionList ] StatementTerminator ]

LoopControlVariable ::=  
 Identifier [ IdentifierModifiers As TypeName ] |  
 Expression

NextExpressionList ::=  
 Expression |  
 NextExpressionList Comma Expression

### Instrucciones For Each...Next

Una instrucción For Each...Next se repite basándose en los elementos de una expresión. Una instrucción For Each especifica una variable de control de bucle y una expresión de enumerador. La variable de control de bucle se especifica bien mediante un identificador seguido de una cláusula As opcional o una expresión. Si seguimos las mismas reglas que en las instrucciones For...Next (10.9.2), la variable de control de bucle hace referencia a una nueva variable local específica de esta instrucción For Each...Next, a una variable ya existente o a una expresión.

La expresión de enumerador debe clasificarse como un valor y su tipo debe ser un tipo de colección u Object. Si el tipo de la expresión de enumerador es Object, todo el procesamiento se aplaza hasta el tiempo de ejecución. De lo contrario, debe existir una conversión del tipo de elemento de la colección en el tipo de la variable de control de bucle.

Una variable de control de bucle no puede usarse en otra instrucción For Each envolvente. Una instrucción For Each debe estar seguida de cerca por una instrucción Next emparejada. Una instrucción Next sin variable de control de bucle coincide con la instrucción For Each abierta más interna. Una instrucción Next con una o más variables de control de bucle se emparejará con los bucles For Each que tienen la misma variable de control de bucle. Si una variable coincide con un bucle For Each que no es el más anidado en ese punto, se produce un error en tiempo de compilación.

Se dice que un tipo C es un tipo de colección si:

1. Es cierto todo lo siguiente:
   1. C contiene un método de instancia, compartido o de extensión accesible con la signatura GetEnumerator() que devuelve un tipo E.
   2. E contiene un método de instancia, compartido o de extensión accesible con la signatura MoveNext() y el tipo de valor devuelto Boolean.
   3. E contiene una propiedad compartida o de instancia accesible denominada Current que posee un getter. El tipo de esta propiedad es el tipo de elemento del tipo de la colección.
2. Implementa la interfaz System.Collections.Generic.IEnumerable(Of T), en cuyo caso se considera que el tipo de elemento de la colección es T.
3. Implementa la interfaz System.Collections.IEnumerable, en cuyo caso se considera que el tipo de elemento de la colección es Object.

A continuación se ofrece un ejemplo de una clase que se puede enumerar:

Public Class IntegerCollection  
 Private integers(10) As Integer  
  
 Public Class IntegerCollectionEnumerator  
 Private collection As IntegerCollection  
 Private index As Integer = -1  
  
 Friend Sub New(c As IntegerCollection)  
 collection = c  
 End Sub  
  
 Public Function MoveNext() As Boolean  
 index += 1  
  
 Return index <= 10  
 End Function  
  
 Public ReadOnly Property Current As Integer  
 Get  
 If index < 0 OrElse index > 10 Then  
 Throw New System.InvalidOperationException()  
 End If  
  
 Return collection.integers(index)  
 End Get  
 End Property  
 End Class  
  
 Public Sub New()  
 Dim i As Integer  
  
 For i = 0 To 10  
 integers(i) = I  
 Next i  
 End Sub  
  
 Public Function GetEnumerator() As IntegerCollectionEnumerator  
 Return New IntegerCollectionEnumerator(Me)  
 End Function  
End Class

Antes de que el bucle comience, se evalúa la expresión de enumerador. Si el tipo de expresión no satisface el modelo de diseño, la expresión se convierte en System.Collections.IEnumerable o System.Collections.Generic.IEnumerable(Of T). Si el tipo de expresión implementa la interfaz genérica, la interfaz genérica se prefiere en tiempo de compilación, pero la interfaz no genérica se prefiere en tiempo de ejecución. Si el tipo de expresión implementa la interfaz genérica muchas veces, la instrucción se considera ambigua y se produce un error en tiempo de compilación.

Anotación

La interfaz no genérica se prefiere en caso de enlace en tiempo de ejecución porque elegir la interfaz genérica significaría que todas las llamadas a los métodos de interfaz implicarían parámetros de tipo. Como no es posible conocer los argumentos de tipo coincidentes en tiempo de ejecución, todas esas llamadas se tendrían que hacer mediante llamadas enlazadas en tiempo de ejecución. Esto sería más lento que llamar a la interfaz no genérica porque ésta se podría invocar mediante llamadas en tiempo de compilación.

GetEnumerator se invoca en el valor resultante y el valor devuelto de la función se guarda en un temporal. Después, al comienzo de cada iteración, se llama a MoveNext en el temporal. Si devuelve False, el bucle finaliza. De lo contrario, cada una de las iteraciones del bucle se ejecuta de la siguiente forma:

1. Si la variable de control de bucle identificó una nueva variable local (en lugar de una ya existente), se crea una copia reciente de esta variable local. Para la iteración actual, todas las referencias dentro del bloque del bucle harán referencia a esta copia.
2. La propiedad Current se recupera, se fuerza al tipo de la variable de control de bucle (con independencia de si la conversión es implícita o explícita) y se asigna a la variable de control de bucle.
3. Se ejecuta el bloque del bucle.

Anotación

Hay un ligero cambio de comportamiento entre la versión 10.0 y la versión 11.0 del lenguaje. Antes de 11.0, *no* se creaba una variable de iteración reciente para cada iteración del bucle. La diferencia solo es visible si la variable de iteración se captura por una expresión lambda o LINQ, que se invoca después del bucle.

Dim lambdas As New List(Of Action)  
For Each x In {1,2,3}  
 lambdas.Add(Sub() Console.WriteLine(x)  
Next  
lambdas(0).Invoke()  
lambdas(1).Invoke()  
lambdas(2).Invoke()

Hasta Visual Basic 10.0, esto producía una advertencia en tiempo de compilación e imprimía "3" tres veces. Esto se debía a que solo había una variable única "x" compartida por todas las iteraciones del bucle y las tres lambdas capturaban la misma "x"; en el momento en que se ejecutaban las expresiones lambda, se retenía el número 3.

A partir de Visual Basic 11.0, imprime "1, 2, 3". Esto se debe a que cada lambda captura una variable "x" diferente.

Anotación

El elemento actual de la iteración se convierte en el tipo de la variable de control de bucle aunque la conversión sea explícita, porque no hay un lugar conveniente para situar un operador de conversión en la instrucción. Esto es especialmente problemático cuando se trabaja con el tipo ahora obsoleto System.Collections.ArrayList, porque su tipo de elemento es Object. Esto habría requerido conversiones en muchos más bucles, lo que no parecía ser lo ideal.

Irónicamente, los genéricos habilitaron la creación de una colección fuertemente tipada, System.Collections.Generic.List(Of T), lo que podría haber hecho que consideráramos de nuevo este punto del diseño, pero por motivos de compatibilidad, no se puede cambiar ahora.

Cuando se alcanza la instrucción Next, la ejecución vuelve al inicio del bucle. Si una variable se especifica después de la palabra clave Next, debe ser la misma que la primera variable después de For Each. Por ejemplo, considere el siguiente código:

Module Test  
 Sub Main()  
 Dim i As Integer  
 Dim c As IntegerCollection = New IntegerCollection()  
  
 For Each i In c  
 Console.WriteLine(i)  
 Next i  
 End Sub  
End Module

Es equivalente al código siguiente:

Module Test  
 Sub Main()  
 Dim i As Integer  
 Dim c As IntegerCollection = New IntegerCollection()  
  
 Dim e As IntegerCollection.IntegerCollectionEnumerator  
  
 e = c.GetEnumerator()  
 While e.MoveNext()  
 i = e.Current  
  
 Console.WriteLine(i)  
 End While  
 End Sub  
End Module

Si el tipo E del enumerador implementa System.IDisposable, el enumerador se desecha al salir del bucle mediante la llamada al método Dispose. Esto asegura que los recursos controlados por el enumerador se liberan. Si el método que contiene la instrucción For Each no usa el control de errores no estructurado, la instrucción For Each se envuelve en una instrucción Try con la llamada al método Dispose en Finally para asegurar la limpieza.

Nota   El tipo System.Array es un tipo de colección y como todos los tipos de matriz se derivan de System.Array, una expresión de tipo de matriz se permite en una instrucción For Each. En matrices unidimensionales, la instrucción For Each enumera los elementos de una matriz en orden de índice creciente, empezando por el índice 0 y acabando por el índice Length - 1. En matrices multidimensionales, se aumentan primero los índices de las dimensiones más a la derecha.

Por ejemplo, el siguiente código imprime 1 2 3 4:

Module Test  
 Sub Main()  
 Dim x(,) As Integer = { { 1, 2 }, { 3, 4 } }  
 Dim i As Integer  
  
 For Each i In x  
 Console.Write(i & " ")  
 Next i  
 End Sub  
End Module

No es válido hacer una bifurcación a un bloque de instrucciones For Each desde fuera del bloque.

ForEachStatement ::=  
 For Each LoopControlVariable In [ LineTerminator ] Expression StatementTerminator  
 [ Block ]  
 [ Next [ NextExpressionList ] StatementTerminator ]

## Instrucciones de control de excepciones

Visual Basic admite el control de excepciones tanto estructurado como no estructurado. Solo se puede usar un estilo de control de excepciones en un método, pero la instrucción Error se puede usar en el control de excepciones estructurado. Si un método usa ambos estilos de control de excepciones, se produce un error en tiempo de compilación.

ErrorHandlingStatement ::=  
 StructuredErrorStatement |  
 UnstructuredErrorStatement

### Instrucciones de control estructurado de excepciones

El control de excepciones estructurado es un método para controlar errores mediante la declaración de bloques explícitos dentro de los que se controlarán algunas excepciones. El control de excepciones estructurado se hace a través de una instrucción Try.

Por ejemplo:

Module Test  
 Sub ThrowException()  
 Throw New Exception()  
 End Sub  
  
 Sub Main()  
 Try  
 ThrowException()  
 Catch e As Exception  
 Console.WriteLine("Caught exception!")  
 Finally  
 Console.WriteLine("Exiting try.")  
 End Try  
 End Sub  
End Module

Una instrucción Try se compone de tres tipos de bloques: bloques try, bloques catch y bloques finally. Un bloque try contiene las instrucciones que se van a ejecutar. Un bloque catch es un bloque de instrucciones que controla una excepción. Un bloque finally contiene las instrucciones que se van a ejecutar cuando se salga de la instrucción Try, con independencia de que se haya producido una excepción y se haya controlado. Una instrucción Try, que solo puede contener un bloque try y un bloque finally, debe contener por lo menos un bloque catch o un bloque finally. No es válido transferir explícitamente la ejecución a un bloque try excepto desde dentro de un bloque catch en la misma instrucción.

StructuredErrorStatement ::=  
 ThrowStatement |  
 TryStatement

TryStatement ::=  
 Try StatementTerminator  
 [ Block ]  
 [ CatchStatement+ ]  
 [ FinallyStatement ]  
 End Try StatementTerminator

#### Bloques Finally

Siempre se ejecuta un bloque Finally cuando la ejecución sale de cualquier parte de la instrucción Try. No se requiere llevar a cabo ninguna acción explícita para ejecutar el bloque Finally; cuando la ejecución abandona la instrucción Try, el sistema ejecutará automáticamente el bloque Finally y después transferirá la ejecución a su destino previsto. El bloque Finally se ejecuta con independencia del modo en que la ejecución abandone la instrucción Try: a través del final del bloque Try, a través del final de un bloque Catch, a través de una instrucción Exit Try, de una instrucción GoTo o no controlando una excepción producida.

Observe que la expresión Await en un método asincrónico, y la instrucción Yield en un método iterator, pueden causar la suspensión del flujo de control en la instancia del método asincrónico o iterator y su reanudación en alguna otra instancia de método. Sin embargo, esto es simplemente una suspensión de la ejecución y no implica la salida de la instancia de método asincrónico o iterator respectiva, por lo que no da lugar a la ejecución de los bloques Finally.

No es válido transferir explícitamente la ejecución a un bloque Finally; tampoco lo es transferirla fuera de un bloque Finally salvo a través de una excepción.

FinallyStatement ::=  
 Finally StatementTerminator  
 [ Block ]

#### Bloques Catch

Si se produce una excepción durante el procesamiento del bloque Try, cada instrucción Catch se examina en el orden en que aparece para determinar si controla la excepción. El identificador especificado en una cláusula Catch representa la excepción que se ha producido. Si el identificador contiene una cláusula As, se considera que el identificador se declara dentro del espacio de declaración local del bloque Catch. De lo contrario, el identificador debe ser una variable local (no una variable estática) que se definió en el bloque contenedor.

Una cláusula Catch sin identificador detectará todas las excepciones derivadas de System.Exception. Una cláusula Catch con un identificador solo detectará las excepciones cuyos tipos sean los mismos o derivados del tipo del identificador. El tipo debe ser System.Exception o un tipo derivado de System.Exception. Cuando se detecta una excepción que se deriva de System.Exception, se almacena una referencia al objeto de la excepción en el objeto devuelto por la función Microsoft.VisualBasic.Information.Err.

Una cláusula Catch con una cláusula When solo detecta las excepciones cuando la expresión se evalúa como True; el tipo de expresión debe ser una expresión booleana según la Sección 11.19. Una cláusula When solo se aplica después de comprobar el tipo de la excepción y la expresión puede hacer referencia al identificador que representa la excepción, como se muestra en este ejemplo:

Module Test  
 Sub Main()  
 Dim i As Integer = 5  
  
 Try  
 Throw New ArgumentException()  
 Catch e As OverflowException When i = 5  
 Console.WriteLine("First handler")  
 Catch e As ArgumentException When i = 4  
 Console.WriteLine("Second handler")  
 Catch When i = 5  
 Console.WriteLine("Third handler")  
 End Try  
  
 End Sub  
End Module

Este ejemplo imprime:

Third handler

Si una cláusula Catch controla la excepción, la ejecución se transfiere al bloque Catch. Al final del bloque Catch, la ejecución pasa a la siguiente instrucción después de la instrucción Try. La instrucción Try no controlará ninguna excepción producida en un bloque Catch. Si ninguna cláusula Catch controla la excepción, la ejecución se transfiere a una ubicación determinada por el sistema.

No es válido transferir la ejecución explícitamente a un bloque Catch.

Anotación

Los filtros de las cláusulas When se evalúan normalmente antes de producirse la excepción. Por ejemplo, el código siguiente imprimirá "Filter, Finally, Catch".

Sub Main()  
 Try  
 Foo()  
 Catch ex As Exception When F()  
 Console.WriteLine("Catch")  
 End Try  
 End Sub

Sub Foo()  
 Try  
 Throw New Exception  
 Finally  
 Console.WriteLine("Finally")  
 End Try  
 End Sub

Function F() As Boolean  
 Console.WriteLine("Filter")  
 Return True  
 End Function

Sin embargo, los métodos Async e Iterator dan lugar a la ejecución de todos los bloques finally dentro de los mismos antes de cualquier filtro externo. Por ejemplo, si el código anterior tenía "Async Sub Foo()", el resultado sería "Finally, Filter, Catch".

CatchStatement ::=  
 Catch [ Identifier [ As NonArrayTypeName ] ] [ When BooleanExpression ] StatementTerminator  
 [ Block ]

#### Instrucción Throw

La instrucción Throw genera una excepción, que es representada por una instancia de un tipo derivado de System.Exception. Si la expresión no se clasifica como un valor o no es un tipo derivado de System.Exception, se produce un error en tiempo de compilación. Si la expresión se evalúa como un valor null en tiempo de ejecución, la excepción generada en su lugar es System.NullReferenceException.

Una instrucción Throw puede omitir la expresión dentro de un bloque catch de una instrucción Try siempre que no intervenga un bloque finally. En ese caso, la instrucción vuelve a producir la excepción que se está controlando dentro del bloque catch. Por ejemplo:

Sub Test(x As Integer)  
 Try  
 Throw New Exception()  
 Catch  
 If x = 0 Then  
 Throw ' OK, rethrows exception from above.  
 Else  
 Try  
 If x = 1 Then  
 Throw ' OK, rethrows exception from above.  
 End If  
 Finally  
 Throw ' Invalid, inside of a Finally.  
 End Try  
 End If  
 End Try  
End Sub

ThrowStatement ::= Throw [ Expression ] StatementTerminator

### Instrucciones de control no estructurado de excepciones

El control de excepciones no estructurado es un método de control de errores mediante instrucciones de bifurcación cuando se produce una excepción. El control de excepciones no estructurado se implementa mediante tres instrucciones: Error, On Error y Resume. Por ejemplo:

Module Test  
 Sub ThrowException()  
 Error 5  
 End Sub  
  
 Sub Main()  
 On Error GoTo GotException  
  
 ThrowException()  
 Exit Sub  
  
GotException:  
 Console.WriteLine("Caught exception!")  
 Resume Next  
 End Sub  
End Module

Cuando un método emplea el control de excepciones no estructurado, se establece un controlador de excepciones estructurado único para todo el método que detecta todas las excepciones. (Observe que en los constructores este controlador no se extiende por la llamada a la llamada a New en el inicio del constructor). El método hace el seguimiento de la ubicación del controlador de excepciones más reciente y la última excepción que se ha producido. A la entrada del método, la ubicación del controlador de excepciones y la excepción se han establecido en Nothing. Cuando se produce una excepción en un método que emplea el control de excepciones no estructurado, se almacena una referencia al objeto de la excepción en el objeto devuelto por la función Microsoft.VisualBasic.Information.Err.

No se permiten instrucciones de control de errores no estructuradas en los métodos async o iterator.

UnstructuredErrorStatement ::=  
 ErrorStatement |  
 OnErrorStatement |  
 ResumeStatement

#### Instrucción Error

Una instrucción Error produce una excepción System.Exception que contiene un número de excepción de Visual Basic 6. La expresión debe clasificarse como un valor y su tipo debe poder convertirse implícitamente en Integer.

ErrorStatement ::= Error Expression StatementTerminator

#### Instrucción On Error

Una instrucción On Error modifica el estado del control de excepciones más reciente. Puede usarse de uno de estos cuatro modos:

On Error GoTo -1 restablece la excepción más reciente en Nothing.

On Error GoTo 0 restablece la ubicación del controlador de excepciones más reciente en Nothing.

On Error GoTo LabelName establece la etiqueta como la ubicación del controlador de excepciones más reciente. Esta instrucción no se puede utilizar en un método que contenga una expresión lambda o de consulta.

On Error Resume Next establece el comportamiento Resume Next como la ubicación del controlador de excepciones más reciente.

OnErrorStatement ::= On Error ErrorClause StatementTerminator

ErrorClause ::=  
 GoTo - 1 |  
 GoTo 0 |  
 GoToStatement |  
 Resume Next

#### Instrucción Resume

Una instrucción Resume devuelve la ejecución a la instrucción que produjo la excepción más reciente. Si se especifica el modificador Next, la ejecución vuelve a la instrucción que se habría ejecutado después de la instrucción que produjo la excepción más reciente. Si se especifica un nombre de etiqueta, la ejecución vuelve a la etiqueta.

Como la instrucción SyncLock contiene un bloque de control de errores estructurado implícito, Resume y Resume Next muestran comportamientos especiales para las excepciones que se producen en las instrucciones SyncLock. Resume devuelve la ejecución al inicio de la instrucción SyncLock, mientras que Resume Next la devuelve a la instrucción siguiente a la instrucción SyncLock. Por ejemplo, considere el siguiente código:

Class LockClass  
End Class  
  
Module Test  
 Sub Main()  
 Dim FirstTime As Boolean = True  
 Dim Lock As LockClass = New LockClass()  
  
 On Error GoTo Handler  
  
 SyncLock Lock  
 Console.WriteLine("Before exception")  
 Throw New Exception()  
 Console.WriteLine("After exception")  
 End SyncLock  
  
 Console.WriteLine("After SyncLock")  
 Exit Sub  
  
Handler:  
 If FirstTime Then  
 FirstTime = False  
 Resume  
 Else  
 Resume Next  
 End If  
 End Sub  
End Module

Imprime el resultado siguiente.

Before exception  
Before exception  
After SyncLock

La primera vez que se recorre la instrucción SyncLock, Resume devuelve la ejecución al inicio de la instrucción SyncLock. La segunda vez que se recorre la instrucción SyncLock, Resume Next devuelve la ejecución al final de la instrucción SyncLock. Resume y Resume Next no se permiten en una instrucción SyncLock.

En todos los casos, cuando se ejecuta una instrucción Resume, la excepción más reciente se establece en Nothing. Si se ejecuta una instrucción Resume sin la excepción más reciente, la instrucción produce una excepción System.Exception con el número de error de Visual Basic 20 (Reanudar sin error).

ResumeStatement ::= Resume [ ResumeClause ] StatementTerminator

ResumeClause ::= Next | LabelName

## Instrucciones de bifurcación

Las instrucciones de bifurcación modifican el flujo de la ejecución en un método. Existen seis instrucciones de bifurcación:

1. Una instrucción GoTo hace que la ejecución se transfiera a la etiqueta especificada en el método. No se permite el uso de GoTo en un bloque Try, Using, SyncLock, With, For o For Each, ni en ningún bloque del bucle si una variable local de dicho bloque se captura en una expresión lambda o LINQ.
2. Una instrucción Exit transfiere la ejecución a la instrucción siguiente tras el final de la instrucción de bloque contenedor inmediato del tipo especificado. Si el bloque es de método, el flujo de control sale del método como se describe en las secciones 10.1, 10.2 y 10.3. Si la instrucción Exit no se encuentra dentro del tipo de bloque especificado en la instrucción, se produce un error en tiempo de compilación.
3. Una instrucción Continue transfiere la ejecución al final de la instrucción de bucle de bloque contenedor inmediato del tipo especificado. Si la instrucción Continue no se encuentra dentro del tipo de bloque especificado en la instrucción, se produce un error en tiempo de compilación.
4. Una instrucción Stop hace que el depurador produzca una excepción.
5. Una instrucción End termina el programa. Los finalizadores se ejecutan antes del cierre, pero los bloques finally de cualquier instrucción Try que se esté ejecutando no se ejecutan. Esta instrucción no se puede usar en programas que no sean ejecutables (por ejemplo, las DLL).
6. Una instrucción Return sin expresión equivale a la instrucción Exit Sub o Exit Function. Una instrucción Return con una expresión solo se permite en un método normal que sea una función, o en un método asincrónico que sea una función con el tipo de valor devuelto Task(Of T) para T. La expresión debe clasificarse como un valor que se puede convertir implícitamente en la *variable devuelta de función* (en el caso de los métodos normales) o en la *variable devuelta de tarea* (en el caso de los métodos asincrónicos). Su comportamiento consiste en evaluar su expresión, almacenarla en la variable devuelta y ejecutar una instrucción Exit Function implícita.

BranchStatement ::=  
 GoToStatement |  
 ExitStatement |  
 ContinueStatement |  
 StopStatement |  
 EndStatement |  
 ReturnStatement

GoToStatement ::= GoTo LabelName StatementTerminator

ExitStatement ::= Exit ExitKind StatementTerminator

ExitKind ::= Do | For | While | Select | Sub | Function | Property | Try

ContinueStatement ::= Continue ContinueKind StatementTerminator

ContinueKind ::= Do | For | While

StopStatement ::= Stop StatementTerminator

EndStatement ::= End StatementTerminator

ReturnStatement ::= Return [ Expression ] StatementTerminator

## Instrucciones de control de matrices

Dos instrucciones simplifican el trabajo con matrices: ReDim y Erase.

ArrayHandlingStatement ::=  
 RedimStatement |  
 EraseStatement

### Instrucción ReDim

Una instrucción ReDim crea instancias de matrices nuevas. Cada cláusula de la instrucción debe clasificarse como una variable o un acceso de propiedad cuyo tipo es un tipo de matriz u Object, y debe ir seguida de una lista de límites de matriz. El número de límites debe ser coherente con el tipo de la variable; para Object se permite cualquier cantidad de límites. En tiempo de ejecución, se crea una instancia de una matriz por cada expresión de izquierda a derecha con los límites especificados y después se asigna a la variable o propiedad. Si el tipo de variable es Object, el número de dimensiones es el especificado y el tipo de elemento de la matriz es Object. Si el número de dimensiones es incompatible con la variable o propiedad en tiempo de ejecución se produce un error en tiempo de compilación. Por ejemplo:

Module Test  
 Sub Main()  
 Dim o As Object  
 Dim b() As Byte  
 Dim i(,) As Integer  
  
 ' The next two statements are equivalent.  
 ReDim o(10,30)  
 o = New Object(10, 30) {}  
  
 ' The next two statements are equivalent.  
 ReDim b(10)  
 b = New Byte(10) {}  
  
 ' Error: Incorrect number of dimensions.  
 ReDim i(10, 30, 40)  
 End Sub  
End Module

Si se especifica la palabra clave Preserve, entonces las expresiones también deben ser clasificables como un valor, y el nuevo tamaño de cada dimensión, excepto para la última de la derecha, debe ser igual que el tamaño de la matriz existente. Los valores de la matriz existente se copian en la nueva matriz: si la nueva matriz es menor, los valores existentes se descartan; si es mayor, los elementos extra se inicializarán en sus valores predeterminados del tipo de elemento de la matriz. Por ejemplo, considere el siguiente código:

Module Test  
 Sub Main()  
 Dim x(5, 5) As Integer  
  
 x(3, 3) = 3  
  
 ReDim Preserve x(5, 6)  
 Console.WriteLine(x(3, 3) & ", " & x(3, 6))  
 End Sub  
End Module

Imprime el resultado siguiente:

3, 0

Si la referencia de matriz existente es un valor null en tiempo de ejecución, no dará error. Si el tamaño de una dimensión cambia, a excepción de la situada más a la derecha, se producirá una excepción System.ArrayTypeMismatchException.

Nota   Preserve no es una palabra reservada.

RedimStatement ::= ReDim [ Preserve ] RedimClauses StatementTerminator

RedimClauses ::=  
 RedimClause |  
 RedimClauses Comma RedimClause

RedimClause ::= Expression ArraySizeInitializationModifier

### Instrucción Erase

Una instrucción Erase establece cada variable o propiedad de la matriz especificada en la instrucción en Nothing. Cada expresión de la instrucción debe clasificarse como una variable o acceso de propiedad cuyo tipo es un tipo de matriz u Object. Por ejemplo:

Module Test  
 Sub Main()  
 Dim x() As Integer = New Integer(5) {}  
  
 ' The following two statements are equivalent.  
 Erase x  
 x = Nothing  
 End Sub  
End Module

EraseStatement ::= Erase EraseExpressions StatementTerminator

EraseExpressions ::=  
 Expression |  
 EraseExpressions Comma Expression

## Instrucción Using

El recolector de elementos no utilizados libera automáticamente las instancias de los tipos cuando se ejecuta una recolección y no se encuentran referencias activas a la instancia. Si un tipo se mantiene en un recurso escaso y valioso (como conexiones de bases de datos o identificadores de archivos), tal vez no sea lo más idóneo esperar hasta que la siguiente recolección de elementos no utilizados limpie una instancia concreta del tipo que ya no se utiliza. Para proporcionar un modo ligero de liberar los recursos antes de una recolección, un tipo puede implementar la interfaz System.IDisposable. Un tipo que haga esto expone un método Dispose al que se puede llamar para forzar la liberación inmediata de los recursos valiosos, por ejemplo:

Module Test  
 Sub Main()  
 Dim x As DBConnection = New DBConnection("...")  
  
 ' Do some work  
 ...  
  
 x.Dispose() ' Free the connection  
 End Sub  
End Module

La instrucción Using automatiza el proceso de adquirir un recurso, ejecutar un grupo de instrucciones y después desechar el recurso. La instrucción puede adoptar dos formas: en una, el recurso es una variable local declarada como parte de la instrucción y tratada como una instrucción de declaración de variable local normal; en la otra, el recurso es el resultado de una expresión.

Si el recurso es una instrucción de declaración de variable local, el tipo de esta última debe ser un tipo que se pueda convertir implícitamente en System.IDisposable. Las variables locales declaradas son de solo lectura, su ámbito es el bloque de instrucciones Using y deben incluir un inicializador. Si el recurso es el resultado de una expresión, la expresión debe clasificarse como un valor y deber ser de un tipo que se pueda convertir implícitamente en System.IDisposable. La expresión solo se evalúa una vez, al comienzo de la instrucción.

El bloque Using está contenido implícitamente en una instrucción Try cuyo bloque finally llama al método IDisposable.Dispose en el recurso. Así se asegura que se desecha el recurso aun cuando se produzca una excepción. Como resultado, no es válido hacer una bifurcación a un bloque Using desde fuera del bloque, y un bloque Using se trata como una instrucción única para los fines de Resume y Resume Next. Si el recurso es Nothing, no se hace ninguna llamada a Dispose. Por lo tanto, el siguiente ejemplo:

Using f As C = New C()  
 ...  
End Using

equivale a:

Dim f As C = New C()  
Try  
 ...  
Finally  
 If f IsNot Nothing Then  
 f.Dispose()  
 End If  
End Try

Una instrucción Using que tiene una instrucción de declaración de variable local puede adquirir múltiples recursos a la vez, lo que equivale a instrucciones Using anidadas. Por ejemplo, una instrucción Using con la siguiente estructura:

Using r1 As R = New R(), r2 As R = New R()  
 r1.F()  
 r2.F()  
End Using

equivale a:

Using r1 As R = New R()  
 Using r2 As R = New R()  
 r1.F()  
 r2.F()  
 End Using  
End Using

UsingStatement ::=  
 Using UsingResources StatementTerminator  
 [ Block ]  
 End Using StatementTerminator

UsingResources ::= VariableDeclarators | Expression

## Instrucciones Await

Una instrucción await tiene la misma sintaxis que una expresión de operador await (11.25), solo se permite en métodos que también admiten expresiones await y tiene el mismo comportamiento que una expresión de operador await.

Sin embargo, puede clasificarse como un valor o como vacío. Se descarta cualquier valor que resulte de evaluar la expresión de operador await.

AwaitStatement ::= AwaitOperatorExpression StatementTerminator

## Instrucciones yield

Las instrucciones yield están relacionadas con los métodos iterator, que se describen en la sección 10.1.2.

Yield es una palabra reservada si el método o la expresión lambda envolvente inmediato donde aparece tiene un modificador Iterator y si Yield aparece después de dicho modificador Iterator; de lo contrario, no es reservada. Tampoco es reservada en las directivas de preprocesador. La instrucción yield solamente se permite en el cuerpo de un método o de una expresión lambda donde es una palabra reservada. Dentro del método o la expresión lambda envolvente inmediato, la instrucción yield no puede aparecer en el cuerpo de un bloque Catch o Finally, ni tampoco en el cuerpo de una instrucción SyncLock.

La instrucción yield toma una única expresión que debe clasificarse como valor y cuyo tipo se puede convertir implícitamente en el tipo de la *variable de iterador actual* (Sección 10.1.2) del método iterador envolvente.

El flujo de control solamente llega a una instrucción Yield cuando se invoca el método MoveNext en un objeto iterador. (Esto se debe a que una instancia de método iterator solamente ejecuta sus instrucciones cuando se llama a los métodos MoveNext o Dispose en un objeto iterador; además, el método Dispose solamente ejecutará código en los bloques Finally,donde no se permite Yield).

Cuando se ejecuta una instrucción Yield, su expresión se evalúa y almacena en la *variable de iterador actual* de la instancia del método iterator asociado a dicho objeto iterador. Se devuelve el valor True al invocador de MoveNext y el punto de control de esta instancia deja de avanzar hasta la siguiente invocación de MoveNext en el objeto iterador.

YieldStatement ::= Yield *Expression* StatementTerminator

# Expresiones

Una expresión es una secuencia de operadores y operandos que especifican el cálculo de un valor o que designan una variable o constante. En este capítulo se definen la sintaxis, el orden de evaluación de los operandos y los operadores, así como el significado de las expresiones.

Expression ::=  
 SimpleExpression |  
 TypeExpression |  
 MemberAccessExpression |  
 DictionaryAccessExpression |  
 InvocationExpression |  
 IndexExpression |  
 NewExpression |  
 CastExpression |  
 OperatorExpression |  
 ConditionalExpression |  
 LambdaExpression |  
 QueryExpression |  
 XMLLiteralExpression |  
 XMLMemberAccessExpression

## Clasificación de expresiones

Todas las expresiones 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 esta clasificación solo puede aparecer como el lado izquierdo de un acceso a miembro. 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 solo puede aparecer como el lado izquierdo de un acceso a miembro. 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 en el mismo nombre. Un grupo de métodos puede tener una expresión de destino y una lista de argumentos de tipo asociadas.

Un puntero a método, que representa la ubicación de un método. Un puntero a método puede tener una expresión de destino y una lista de argumentos de tipo asociadas.

Un método lambda, que es un método anónimo.

Un grupo de propiedades, que es un conjunto de propiedades sobrecargadas en el mismo nombre. Un grupo de propiedades puede tener asociada una expresión de destino.

Un acceso de propiedad. Todos los accesos de propiedad tienen un tipo asociado, esto es, el tipo declarado de la propiedad. Un acceso de propiedad puede tener asociada una expresión de destino.

Un acceso enlazado en tiempo de ejecución, que representa un método o acceso de propiedad diferido hasta el tiempo de ejecución. Un acceso enlazado en tiempo de ejecución puede tener una expresión de destino y una lista de argumentos de tipo asociadas. El tipo de un acceso enlazado en tiempo de ejecución siempre es Object.

Un acceso de evento. Todos los accesos de evento tienen un tipo asociado, el tipo declarado del evento. Un acceso de evento puede tener asociada una expresión de destino. Un acceso de evento puede aparecer como el primer argumento de las instrucciones RaiseEvent, AddHandler y RemoveHandler. En cualquier otro contexto, una expresión que se clasifica como un acceso de evento produce un error en tiempo de compilación.

Un literal de matriz, que representa los valores iniciales de una matriz cuyo tipo no se ha determinado todavía.

Vacío. Esto ocurre cuando la expresión es una invocación de una subrutina o una expresión de operador await sin ningún resultado. Una expresión clasificada como void solo es válida en el contexto de una instrucción de invocación o una instrucción await.

Un valor predeterminado. Solo el literal Nothing produce esta clasificación.

El resultado final de una expresión suele ser un valor o una variable, y las demás categorías de las expresiones funcionan como valores intermedios que solo se permiten en algunos contextos.

Observe que las expresiones cuyo tipo es un parámetro de tipo pueden usarse en instrucciones y expresiones que requieren que el tipo de una expresión tenga ciertas características (como ser un tipo de referencia, un tipo de valor, derivar de algún tipo, etc.) si las restricciones impuestas en el parámetro de tipo satisface esas características.

### Reclasificación de expresiones

Normalmente, cuando una expresión se usa en un contexto que requiere una clasificación diferente de la de la expresión, se produce un error en tiempo de compilación, por ejemplo al intentar asignar un valor a un literal. Sin embargo, en muchos casos es posible cambiar la clasificación de una expresión mediante el proceso de reclasificación.

Si la reclasificación se realiza correctamente, se considera si es widening o narrowing. A menos que se indique lo contrario, todas las reclasificaciones de esta lista son widening.

Se pueden reclasificar los tipos de expresiones siguientes:

1. Una variable se puede reclasificar como valor. Se obtiene el valor guardado en la variable.
2. Un grupo de métodos se puede reclasificar como valor. La expresión del grupo de métodos se interpreta como una expresión de invocación con la expresión de destino y una lista de parámetros de tipo asociadas, y un paréntesis vacío (es decir, f se interpreta como f() y f(Of Integer) se interpreta como f(Of Integer)()). Esta reclasificación puede dar como resultado la expresión que después se reclasifica como vacía.
3. Un puntero a método se puede reclasificar como valor. Esta reclasificación solo puede tener lugar en el contexto de una conversión donde se conoce el tipo de destino. La expresión de puntero a método se interpreta como el argumento a una expresión de creación de instancias de delegado del tipo adecuado con la lista de argumentos de tipo asociada. Por ejemplo:

Delegate Sub D(i As Integer)  
  
Module Test  
 Sub F(i As Integer)  
 End Sub  
  
 Sub Main()  
 Dim del As D  
  
 ' The next two lines are equivalent.  
 del = AddressOf F  
 del = New D(AddressOf F)  
 End Sub  
End Module

1. Un método lambda se puede reclasificar como valor. Si la reclasificación tiene lugar en el contexto de una conversión en la que se conoce el tipo de destino, se produce una de estas dos reclasificaciones:
2. Si el tipo de destino es un tipo de delegado, el método lambda se interpreta como el argumento de una expresión de construcción de delegados del tipo adecuado.
3. Si el tipo de destino es System.Linq.Expressions.Expression(Of T) y T es un tipo de delegado, el método lambda se interpreta como si se usara en una expresión de construcción de delegado para T y después se convirtiera en un árbol de expresión.

Un método lambda async o iterator solamente puede interpretarse como el argumento de una expresión de construcción de delegados, si el delegado no tiene parámetros ByRef.

Si la conversión de cualquiera de los tipos de parámetros del delegado en los tipos de parámetros lambda correspondientes es de tipo narrowing, la reclasificación se considera como narrowing; de lo contrario, es widening.

Anotación

La traducción exacta entre los métodos lambda y los árboles de expresión puede no ser fija entre versiones del compilador y sobrepasa el alcance de esta especificación. En Microsoft Visual Basic 11.0, todas las expresiones lambda pueden convertirse en árboles de expresión sujetas a las restricciones siguientes:

1. Solo pueden convertirse en árboles de expresión las expresiones lambda de línea única sin parámetros ByRef. De las expresiones lambda de línea única Sub, solo pueden convertirse en árboles de expresión las instrucciones de invocación.
2. Las expresiones de tipo anónimo no se pueden convertir en árboles de expresión si se utiliza un inicializador de campo anterior para inicializar un inicializador de campo posterior, por ejemplo New With {.a=1, .b=.a}
3. Las expresiones de inicializador de objeto no se pueden convertir en árboles de expresión si el objeto que se inicializa se usa en uno de los inicializadores de campo, por ejemplo New C1 With {.a=1, .b=.Method1()}
4. Las expresiones de creación de matrices multidimensionales solo se pueden convertir en árboles de expresión si declaran su tipo de elemento de forma explicita.
5. Las expresiones de enlace en tiempo de ejecución no se pueden convertir en árboles de expresión
6. Cuando una variable o un campo se pasa ByRef a una expresión de invocación pero no tiene exactamente el mismo tipo que el parámetro ByRef, o cuando se pasa una propiedad ByRef, la semántica normal de VB es que se pasa una copia del argumento ByRef y su valor final se vuelve a copiar en la propiedad, campo o variable. En los árboles de expresión, no se vuelve a copiar el valor final.

Todas estas restricciones se aplican también a las expresiones lambda anidadas

Si el tipo de destino no se conoce, el método lambda se interpreta como el argumento a una expresión de creación de instancias de delegado de un tipo de delegado anónimo con la misma firma del método lambda. Si se omite el tipo de alguno de los parámetros y se usa semántica estricta, se produce un error en tiempo de compilación; de lo contrario, Object se sustituye para los tipos de parámetro que falten. Por ejemplo:

Module Test  
 Sub Main()  
 ' Type of x will be equivalent to Func(Of Object, Object, Object)  
 Dim x = Function(a, b) a + b  
  
 ' Type of y will be equivalent to Action(Of Object, Object)  
 Dim y = Sub(a, b) Console.WriteLine(a + b)  
 End Sub  
End Module

1. Un grupo de propiedades se puede reclasificar como un acceso de propiedad. La expresión del grupo de propiedades se interpreta como una expresión de índice con un paréntesis vacío (es decir, f se interpreta como f()).
2. Un acceso de propiedad se puede reclasificar como valor. La expresión de acceso a propiedad se interpreta como una expresión de invocación del descriptor de acceso Get de la propiedad. Si la propiedad no tiene un getter, se produce un error durante la compilación.
3. Un acceso enlazado en tiempo de ejecución se puede reclasificar como un método o un acceso de propiedad enlazado en tiempo de ejecución. En una situación donde un acceso enlazado en tiempo de ejecución puede reclasificarse como un acceso a método y un acceso de propiedad; se prefiere la reclasificación como acceso de propiedad.
4. Un acceso enlazado en tiempo de ejecución se puede reclasificar como valor.
5. Un literal de matriz se puede reclasificar como valor. El tipo del valor se determina como sigue:
6. Si la reclasificación tiene lugar en el contexto de una conversión en la que se conoce el tipo de destino y se sabe que es un tipo de matriz, el literal de matriz se reclasifica como valor de tipo T(). Si el tipo de destino es System.Collections.Generic.IList(Of T), IReadOnlyList(Of T), ICollection(Of T), IReadOnlyCollection(Of T) o IEnumerable(Of T) y el literal de matriz tiene un nivel de anidamiento, dicho literal se reclasifica como valor de tipo T().
7. De lo contrario, el literal de matriz se reclasifica en un valor cuyo tipo es una matriz de igual rango que el nivel de anidamiento, donde el tipo de elemento lo determina el tipo dominante de los elementos del inicializador; si no se puede determinar un tipo dominante, se emplea Object. Por ejemplo:

' x Is GetType(Double(,,))  
Dim x = { { { 1, 2.0 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } }.GetType()  
  
' y Is GetType(Integer())  
Dim y = { 1, 2, 3 }.GetType()  
  
' z Is GetType(Object())  
Dim z = { 1, "2" }.GetType()  
  
' Error: Inconsistent nesting  
Dim a = { { 10 }, { 20, 30 } }.GetType()

Anotación

Hay un ligero cambio de comportamiento entre la versión 9.0 y 10.0 del lenguaje. Antes de la versión 10.0, los inicializadores de elemento de matriz no afectaban a la inferencia de tipos de variable local y ahora sí. Por tanto Dim a() = { 1, 2, 3 } habría inferido Object() como el tipo de a en la versión 9.0 del lenguaje e Integer() en la versión 10.0.

La reclasificación reinterpreta el literal de matriz como una expresión de creación de matrices. Por tanto, los ejemplos:

Dim x As Double = { 1, 2, 3, 4 }  
Dim y = { "a", "b" }

equivalen a:

Dim x As Double = New Double() { 1, 2, 3, 4 }  
Dim y = New String() { "a", "b" }

La reclasificación se considera como narrowing si cualquier conversión de una expresión de elemento en el tipo de elemento de matriz es narrowing; de lo contrario, se considera como widening.

1. El valor predeterminado Nothing se puede reclasificar como valor. En un contexto donde se conoce el tipo de destino, el resultado es el valor predeterminado del tipo de destino. En un contexto donde no se conoce el tipo de destino, el resultado es un valor null de tipo Object.

Las expresiones de espacio de nombres, de tipo, de acceso de evento o vacía no se pueden reclasificar. Se pueden efectuar varias reclasificaciones al mismo tiempo. Por ejemplo:

Module Test  
 Sub F(i As Integer)  
 End Sub  
  
 ReadOnly Property P() As Integer  
 Get  
 End Get  
 End Sub  
  
 Sub Main()  
 F(P)  
 End Property  
End Module

En este caso, la expresión de grupo de propiedades P se reclasifica primero de grupo de propiedades en un acceso de propiedad y después, de un acceso de propiedad en un valor. Se realiza el menor número de reclasificaciones para alcanzar una clasificación válida en el contexto.

## Expresiones constantes

Una expresión constante es una expresión cuyo valor se puede evaluar totalmente en tiempo de compilación. El tipo de una expresión constante puede ser uno de los siguientes: Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long, Char, Single, Double, Decimal, Date, Boolean, String, Object cualquier tipo de enumeración. Se permiten las siguientes construcciones en expresiones de constantes:

Literales (incluido Nothing).

Referencias a miembros de tipo constante o variables locales constantes.

Referencias a miembros de tipos de enumeración.

Subexpresiones entre paréntesis.

Las expresiones de conversión, siempre que el tipo de destino sea uno de los antes indicados. Las conversiones en y de String son una excepción a esta regla y solo se permiten en valores null porque las conversiones String siempre se hacen en la referencia cultural del entorno de ejecución en tiempo de ejecución. Observe que las expresiones de conversión de constantes solo pueden utilizar conversiones intrínsecas.

Los operadores unarios +, – y Not, siempre que el operando y el resultado sea de un tipo arriba enumerado.

Los operadores binarios +, –, \*, ^, Mod, /, \, <<, >>, &, And, Or, Xor, AndAlso, OrElse, =, <, >, <>, <= y =>, siempre que cada operando y resultado sea de un tipo arriba enumerado.

El operador condicional If, siempre que cada operando y resultado sea de un tipo arriba enumerado.

Las siguientes funciones en tiempo de ejecución:

Microsoft.VisualBasic.Strings.ChrW

Microsoft.VisualBasic.Strings.Chr, si el valor constante está entre 0 y 128

Microsoft.VisualBasic.Strings.AscW, si la cadena constante no está vacía

Microsoft.VisualBasic.Strings.Asc, si la cadena constante no está vacía

No se permiten las siguientes construcciones en expresiones constantes:

Enlace implícito a través de un contexto With.

Las expresiones constantes de un tipo entero (ULong, Long, UInteger, Integer, UShort, Short, SByte o Byte) se pueden convertir implícitamente en un tipo entero más restringido, y las de tipo Double se pueden convertir implícitamente en Single, siempre que el valor de la expresión constante esté dentro del intervalo del tipo de destino. Estas conversiones de restricción (narrowing) se permiten con independencia de si se usa semántica estricta o permisiva.

ConstantExpression ::= Expression

## Expresiones enlazadas en tiempo de ejecución

Cuando el destino de una expresión de acceso a miembro o una expresión de índice es del tipo Object, el proceso de la expresión se puede diferir hasta el tiempo de ejecución. Diferir el procesamiento de esta manera se denomina enlace en tiempo de ejecución. El enlace en tiempo de ejecución permite que las variables Object se usen en modo sin tipo, donde toda la resolución de miembros se basa en el tipo en tiempo de ejecución del valor de la variable. Si el entorno de compilación u Option Strict especifica semántica estricta, el enlace en tiempo de ejecución causa un error en tiempo de compilación. No se pasa por alto ningún miembro que no sea público durante el enlace en tiempo de ejecución, incluidos con fines de resolución de sobrecarga. Observe que, a diferencia del enlace en tiempo de compilación, invocar o tener acceso a un enlace en tiempo de ejecución de un miembro Shared provocará que el destino de la invocación se evalúe en tiempo de ejecución. Si la expresión es de invocación para un miembro definido en System.Object, no se producirá el enlace en tiempo de ejecución.

En general, los accesos enlazados en tiempo de ejecución se resuelven en tiempo de ejecución buscando el identificador en el tipo en tiempo de ejecución de la expresión. Si la búsqueda de miembros enlazados en tiempo de ejecución da error en tiempo de ejecución, se produce una excepción System.MissingMemberException. Como la búsqueda de miembros enlazados en tiempo de ejecución solo se lleva a cabo fuera del tipo en tiempo de ejecución de la expresión de destino asociada, un tipo en tiempo de ejecución de un objeto nunca puede ser una interfaz. Por tanto, no es posible el acceso a miembros de interfaz en una expresión de acceso a miembro enlazado en tiempo de ejecución.

The arguments to a late-bound member access are evaluated in the order they appear in the member access expression: not the order in which parameters are declared in the late-bound member. The following example illustrates this difference:Class C

Public Sub f(ByVal x As Integer, ByVal y As Integer)

End Sub

End Class

Module Module1  
 Sub Main()

Console.Write("Early-bound: ")

Dim c As C = New C

c.f(y:=t("y"), x:=t("x"))

Console.Write(vbCrLf & "Late-bound: ")

Dim o As Object = New C

o.f(y:=t("y"), x:=t("x"))

End Sub

Function t(ByVal s As String) As Integer

Console.Write(s)

Return 0

End Function

End Module

Este código muestra:

Early-bound: xy

Late-bound: yx

Como la resolución de sobrecargas en el enlace en tiempo de ejecución se hace en el tipo en tiempo de ejecución de los argumentos, es posible que una expresión pueda producir resultados diferentes según se evalúen durante la compilación o la ejecución. En el ejemplo siguiente se ilustra esta diferencia:

Class Base  
End Class  
  
Class Derived  
 Inherits Base  
End Class  
  
Module Test  
 Sub F(b As Base)  
 Console.WriteLine("F(Base)")  
 End Sub  
  
 Sub F(d As Derived)  
 Console.WriteLine("F(Derived)")  
 End Sub  
  
 Sub Main()  
 Dim b As Base = New Derived()  
 Dim o As Object = b  
  
 F(b)  
 F(o)  
 End Sub  
End Module

Este código muestra:

F(Base)  
F(Derived)

## Expresiones simples

Las expresiones simples son literales, expresiones de paréntesis, expresiones de instancia o expresiones de nombre simples.

SimpleExpression ::=  
 LiteralExpression |  
 ParenthesizedExpression |  
 InstanceExpression |  
 SimpleNameExpression |  
 AddressOfExpression

### Expresiones literales

Las expresiones literales se evalúan como el valor que representa el literal. Una expresión literal se clasifica como valor, excepto el literal Nothing, que se clasifica como valor predeterminado.

LiteralExpression ::= Literal

### Expresiones entre paréntesis

Una expresión entre paréntesis es una expresión encerrada entre paréntesis. Se clasifica como valor y la expresión encerrada entre paréntesis debe clasificarse como valor. Una expresión entre paréntesis se evalúa como el valor de la expresión entre los paréntesis.

ParenthesizedExpression ::= OpenParenthesis Expression CloseParenthesis

### Expresiones de instancia

Una expresión de instancia es la palabra clave Me. Solamente puede usarse dentro del cuerpo de un descriptor de acceso de propiedad, constructor o método no compartido. Se clasifica como un valor. . La palabra clave Me representa la instancia del tipo que contiene el descriptor de acceso de la propiedad o el método que se está ejecutando. Si un constructor invoca explícitamente otro constructor (Sección 9.3), Me no se puede usar hasta después de la llamada de ese constructor, ya que la instancia aún no se ha construido.

InstanceExpression ::= Me

### Expresiones de nombre simple

Una expresión de nombre simple consiste en un identificador único seguido de una lista de argumentos de tipo opcionales. El nombre se resuelve y clasifica según las siguientes “reglas de resolución de nombres simples”:

1. Comenzando con el bloque inmediatamente contenedor y siguiendo con cada bloque contenedor más externo (si lo hay), si el identificador coincide con el nombre de una variable local, variable estática, variable local constante, parámetro de tipo de método o parámetro, entonces el identificador hace referencia a la entidad coincidente.

Si el identificador coincide con una variable local, con una variable estática o con una variable local constante y se proporcionó una lista de argumentos de tipo, se produce un error en tiempo de compilación. Si el identificador coincide con un parámetro de tipo de método y se proporcionó una lista de argumentos de tipo, no se produce ninguna coincidencia y la resolución continúa. Si el identificador coincide con una variable local, dicha variable es la función implícita o la variable local devuelta del descriptor de acceso Get, y si la expresión es parte de una expresión de invocación, instrucción de invocación o una expresión AddressOf, no se produce ninguna coincidencia y la resolución continúa.

La expresión se clasifica como variable si es una variable local, estática o un parámetro. La expresión se clasifica como tipo si es un parámetro de tipo de método. La expresión se clasifica como valor si es una constante local.

1. Por cada tipo anidado que contenga la expresión, comenzando por la más interna y yendo a la más externa, si una búsqueda del identificador en el tipo encuentra una coincidencia con un miembro accesible:
2. Si el miembro de tipo coincidente es un parámetro de tipo, el resultado se clasifica como tipo y es el parámetro de tipo coincidente. Si se proporcionó una lista de argumentos de tipo, no se produce ninguna coincidencia y la resolución continúa.
3. De lo contrario, si el tipo es el tipo contenedor inmediato y la búsqueda identifica un miembro de tipo no compartido, el resultado es el mismo que un acceso a miembro de la forma Me.E(Of A), donde E es el identificador y A es la lista de argumentos de tipo, si hay.
4. De lo contrario, el resultado es exactamente el mismo que un acceso a miembro de la forma T.E(Of A), donde T es el tipo que contiene el miembro coincidente, E es el identificador y A es la lista de argumentos de tipo, si hay. En este caso, es un error que el identificador haga referencia a un miembro no compartido.
5. Por cada espacio de nombres anidado, comenzando por el más interno y terminando en el más externo, haga lo siguiente:
6. Si el espacio de nombres contiene un tipo accesible con el nombre dado y tiene el mismo número de parámetros de tipo que se proporcionaron en la lista de argumentos de tipo, si hay una, el identificador hace referencia a ese tipo y se clasifica como tipo.
7. De otro modo, si no se proporcionó la lista de argumentos de tipo y ningún espacio de nombres contiene un miembro de espacio de nombres con el nombre dado, el identificador hace referencia a ese espacio de nombres y se clasifica como espacio de nombres.
8. De otro modo, si el espacio de nombres contiene uno o más módulos estándar accesibles y una búsqueda de nombres de miembros del identificador arroja una coincidencia accesible exactamente en un módulo estándar, el resultado es el mismo que un acceso a miembro con la forma M.E(Of A), donde M es el módulo estándar que contiene el miembro coincidente, E es el identificador y A es la lista de argumentos de tipo, si hay una. Si el identificador coincide con miembros de tipo accesibles en más de un módulo estándar, se produce un error en tiempo de compilación.
9. Si el archivo de origen tiene uno o más alias de importación y el identificador coincide con el nombre de uno de ellos, el identificador hace referencia a ese espacio de nombres o tipo. Si se proporciona una lista de argumentos de tipo, se produce un error en tiempo de compilación.
10. Si el archivo de origen que contiene la referencia de nombre tiene una o más importaciones:
11. Si el identificador coincide con el nombre de un tipo accesible con el mismo nombre y número de parámetros de tipo que se proporcionaron en la lista de argumentos de tipo, si hay una, o un miembro de tipo exactamente en una importación, el identificador hace referencia a ese tipo o miembro de tipo. Si el identificador coincide en más de una importación con el nombre de un tipo accesible con el mismo nombre y número de parámetros de tipo que se proporcionaron en la lista de argumentos de tipo, si hay una, o un miembro de tipo accesible, se produce un error en tiempo de compilación.
12. De lo contrario, si no hay lista de argumentos de tipo y el identificador coincide exactamente en una importación con el nombre de un espacio de nombres con tipos accesibles, el identificador hace referencia a ese espacio de nombres. Si no se proporcionó una lista de argumentos de tipo y el identificador coincide en más de una importación con el nombre de un espacio de nombres con tipos accesibles, se produce un error en tiempo de compilación.
13. De lo contrario, si la importación contiene uno o más módulos estándar accesibles y una búsqueda de nombres de miembros del identificador arroja una coincidencia accesible exactamente en un módulo estándar, el resultado es el mismo que un acceso a miembro con la forma M.E(Of A), donde M es el módulo estándar que contiene el miembro coincidente, E es el identificador y A es la lista de argumentos de tipo, si hay una. Si el identificador coincide con miembros de tipo accesibles en más de un módulo estándar, se produce un error en tiempo de compilación.
14. Si el entorno de compilación define uno o más alias de importación y el identificador coincide con el nombre de uno de ellos, el identificador hace referencia a ese espacio de nombres o tipo. Si se proporciona una lista de argumentos de tipo, se produce un error en tiempo de compilación.
15. Si el entorno de compilación define una o más importaciones:
16. Si el identificador coincide con el nombre de un tipo accesible con el mismo nombre y número de parámetros de tipo que se proporcionaron en la lista de argumentos de tipo, si hay una, o un miembro de tipo exactamente en una importación, el identificador hace referencia a ese tipo o miembro de tipo. Si el identificador coincide en más de una importación con el nombre de un tipo accesible con el mismo nombre y número de parámetros de tipo que se proporcionaron en la lista de argumentos de tipo, si hay una, o un miembro de tipo, se produce un error en tiempo de compilación.
17. De lo contrario, si no hay lista de argumentos de tipo y el identificador coincide exactamente en una importación con el nombre de un espacio de nombres con tipos accesibles, el identificador hace referencia a ese espacio de nombres. Si no se proporcionó una lista de argumentos de tipo y el identificador coincide en más de una importación con el nombre de un espacio de nombres con tipos accesibles, se produce un error en tiempo de compilación.
18. De lo contrario, si la importación contiene uno o más módulos estándar accesibles y una búsqueda de nombres de miembros del identificador arroja una coincidencia accesible exactamente en un módulo estándar, el resultado es el mismo que un acceso a miembro con la forma M.E(Of A), donde M es el módulo estándar que contiene el miembro coincidente, E es el identificador y A es la lista de argumentos de tipo, si hay una. Si el identificador coincide con miembros de tipo accesibles en más de un módulo estándar, se produce un error en tiempo de compilación.
19. De lo contrario, el nombre dado por el identificador no está definido.

Una expresión de nombre simple que no está definida produce un error en tiempo de compilación.

Normalmente un nombre solo aparece una vez en un espacio de nombres concreto. Sin embargo, como los espacios de nombres se pueden declarar entre múltiples ensamblados de .NET, es posible que se dé una situación en la que dos ensamblados definen un tipo con el mismo nombre completo. En ese caso, se prefiere un tipo declarado en el grupo de archivos de origen actual a un tipo declarado en un ensamblado de .NET externo. De otro modo, el nombre es ambiguo y no hay forma de romper esa ambigüedad.

SimpleNameExpression ::= Identifier [ OpenParenthesis Of TypeArgumentList CloseParenthesis ]

### Expresiones AddressOf

Una expresión AddressOf se usa para generar un puntero a método. La expresión consiste en una palabra clave AddressOf y una expresión que debe clasificarse como un grupo de métodos o acceso enlazado en tiempo de ejecución. El grupo de métodos no puede hacer referencia a constructores.

El resultado se clasifica como un puntero a método, con la misma expresión de destino asociada y lista de argumentos de tipo (si la hay) que el grupo de métodos.

AddressOfExpression ::= AddressOf Expression

## Expresiones de tipo

Una expresión de tipo es una de las siguientes expresiones: GetType, TypeOf...Is, Is o GetXmlNamespace.

TypeExpression ::=  
 GetTypeExpression |  
 TypeOfIsExpression |  
 IsExpression |  
 GetXmlNamespaceExpression

### Expresiones GetType

Una expresión GetType consiste en la palabra clave GetType y el nombre de un tipo. Una expresión de tipo se clasifica como valor y el valor de la expresión es la clase de reflexión (System.Type) que representa ese tipo. Si la expresión es un parámetro de tipo, devolverá el objeto System.Type que corresponde al argumento de tipo proporcionado por el parámetro de tipo en tiempo de ejecución.

El nombre de tipo en una expresión GetType es especial por dos razones:

El nombre de tipo se permite que sea System.Void, el único lugar en el lenguaje donde se puede hacer referencia a este nombre de tipo.

El nombre de tipo puede ser un tipo genérico construido con los argumentos de tipo omitidos. Esto permite que la expresión GetType devuelva el objeto System.Type que se corresponde con el propio tipo genérico.

En el siguiente ejemplo se muestra la expresión GetType:

Module Test  
 Sub Main()  
 Dim t As Type() = { GetType(Integer), GetType(System.Int32), \_  
 GetType(String), GetType(Double()) }  
 Dim i As Integer  
  
 For i = 0 To t.Length – 1  
 Console.WriteLine(t(i).Name)  
 Next i  
 End Sub  
End Module

El resultado que se obtiene es:

Int32  
Int32  
String  
Double[]

GetTypeExpression ::= GetType OpenParenthesis GetTypeTypeName CloseParenthesis

GetTypeTypeName ::=  
 TypeName |  
 *Qualified*OpenTypeName

QualifiedOpenTypeName ::=  
 Identifier [ TypeArityList ] |  
 Global Period IdentifierOrKeyword [ TypeArityList ] |  
 QualifiedOpenTypeName Period IdentifierOrKeyword [ TypeArityList ]

*TypeArityList* ::= ( Of [ *CommaList* ] )

### CommaList ::= Comma | CommaList CommaExpresiones TypeOf...Is

Una expresión TypeOf...Is se utiliza para comprobar si el tipo en tiempo de ejecución de un valor es compatible con un tipo especificado. El primer operando debe clasificarse como valor, no puede ser un método reclasificado y debe ser de un tipo de referencia o un tipo de parámetro de tipo sin restricciones. El tipo del segundo operando debe ser un nombre de tipo. El resultado de la expresión se clasifica como valor y es un valor Boolean. La expresión se evalúa como True si el tipo en tiempo de ejecución del operando tiene una identidad, valor predeterminado, referencia, matriz, tipo de valor o conversión de parámetros de tipo en el tipo; de lo contrario, es False. Se produce un error durante la compilación si no existe una conversión entre el tipo de la expresión y el tipo específico.

TypeOfIsExpression ::= TypeOf Expression Is [ LineTerminator ] TypeName

### Expresiones Is

Una expresión Is o IsNot se usa para hacer una comparación de igualdad de referencia. Cada expresión debe clasificarse como un valor y el tipo de cada expresión debe ser un tipo de referencia, un tipo de parámetros de tipo sin restricciones o un tipo de valor que admite valores null. Si el tipo de una expresión es un parámetro de tipo sin restricciones o un tipo de valor que admite valores null, sin embargo, la otra expresión debe ser el literal Nothing.

El resultado se clasifica como valor y su tipo es Boolean. Una operación Is se evalúa como True si ambos valores hacen referencia a la misma instancia o ambos valores son Nothing; de lo contrario, False. Una operación IsNot se evalúa como False si ambos valores hacen referencia a la misma instancia o ambos valores son Nothing; de lo contrario, True.

IsExpression ::=  
 Expression Is [ LineTerminator ] Expression |  
 Expression IsNot [ LineTerminator ] Expression

### Expresiones GetXmlNamespace

Una expresión GetXmlNamespace consiste en la palabra clave GetXmlNamespace y el nombre de un espacio de nombres XML declarada por el archivo de origen o entorno de compilación. Una expresión de espacio de nombres XML se clasifica como valor y el valor de la expresión es una instancia de System.Xml.Linq.XNamespace que representa ese espacio de nombres XML. Si ese tipo no está disponible, se produce un error en tiempo de compilación.

Por ejemplo:

Imports <xmlns:db="http://example.org/database">  
  
Module Test  
 Sub Main()  
 Dim db = GetXmlNamespace(db)  
  
 ' The following are equivalent  
 Dim customer1 = \_  
 New System.Xml.Linq.XElement(db + "customer", "Bob")  
 Dim customer2 = <db:customer>Bob</>  
 End Sub  
End Module

Todo lo que está entre paréntesis se considera parte del nombre del espacio de nombres, de manera que las reglas XML de aspectos como los espacios en blanco son aplicables. Por ejemplo:

Imports <xmlns:db-ns="http://example.org/database">  
  
Module Test  
 Sub Main()  
  
 ' Error, XML name expected  
 Dim db1 = GetXmlNamespace( db-ns )  
  
 ' Error, ')' expected  
 Dim db2 = GetXmlNamespace(db \_  
 )  
  
 ' OK.  
 Dim db3 = GetXmlNamespace(db-ns)  
 End Sub  
End Module

La expresión de espacio de nombres XML también se puede omitir, en cuyo caso la expresión devuelve el objeto que representa el espacio de nombres XML predeterminado.

GetXmlNamespaceExpression ::= GetXmlNamespace OpenParenthesis [ XMLNamespaceName ]  
 CloseParenthesis

## Expresiones de acceso a miembros

Una expresión de acceso a miembro se usa para tener acceso a un miembro de una entidad. Un acceso a miembro con la forma E.I(Of A), donde E es una expresión, un nombre de tipo que no es matriz, la palabra clave Global o se omite, e I es un identificador con una lista de argumentos de tipo opcional A, se evalúa y clasifica como sigue:

1. Si E se omite, la siguiente expresión que contiene la instrucción With se sustituye por E y se lleva a cabo el acceso a miembro. Si no hay ninguna instrucción contenedora With, se produce un error en tiempo de compilación.
2. Si E se clasifica como un espacio de nombres o E es la palabra clave Global, la búsqueda de miembros se efectúa en el contexto del espacio de nombres especificado. Si I es el nombre de miembro accesible de ese espacio de nombres con el mismo número de parámetros de tipo que se proporcionaron en la lista de argumentos de tipo, si hay una, el resultado es ese miembro. El resultado se clasifica como espacio de nombres o como tipo en función del miembro. En caso contrario, se producirá un error en tiempo de compilación.
3. Si E es un tipo o una expresión clasificada como tipo, la búsqueda de miembros se efectúa en el contexto del tipo especificado. Si I es el nombre de un miembro accesible de E, E.I se evalúa y clasifica como sigue:
4. Si I es la palabra clave New y E no es una enumeración, se produce un error en tiempo de compilación.
5. Si I identifica un tipo con el mismo número de parámetros de tipo que se proporcionaron en la lista de argumentos de tipo, si hay una, el resultado es ese tipo.
6. Si I identifica uno o varios métodos, el resultado es un grupo de métodos con la lista de argumentos de tipo asociada y sin expresión de destino asociada.
7. Si I identifica una o varias propiedades y no se proporcionó lista de argumentos de tipo, el resultado es un grupo de propiedades sin expresión de destino asociada.
8. Si I identifica una variable compartida y no se proporcionó lista de argumentos de tipo, el resultado es una variable o un valor. Si la variable es de solo lectura y la referencia tiene lugar fuera del constructor compartido del tipo donde está declarada la variable, el resultado es el valor de la variable compartida I de E. En caso contrario, el resultado es la variable compartida I en E.
9. Si I identifica un evento compartido y no se proporcionó lista de argumentos de tipo, el resultado es un acceso de eventos sin expresión de destino asociada.
10. Si I identifica una constante y no se proporcionó lista de argumentos de tipo, el resultado es el valor de esa constante.
11. Si I identifica un miembro de enumeración y no se proporcionó lista de argumentos de tipo, el resultado es el valor de ese miembro de enumeración.
12. De lo contrario, E.I será una referencia de miembro no válida y se producirá un error en tiempo de compilación.
13. Si E se clasifica como una variable o valor, cuyo tipo es T, la búsqueda de miembros se efectúa en el contexto de T. Si I es el nombre de un miembro accesible de T, E.I se evalúa y clasifica como sigue:
14. Si I es la palabra clave New, E es Me, MyBase o MyClass y no se proporcionaron argumentos de tipo, el resultado es un grupo de métodos que representa los constructores de instancia del tipo de E con una expresión de destino asociada de E y sin lista de argumentos de tipo. En caso contrario, se producirá un error en tiempo de compilación.
15. Si I identifica uno o varios métodos, incluidos los métodos de extensión si T no es Object, el resultado es un grupo de métodos con la lista de argumentos de tipo asociada y una expresión de destino asociada de E.
16. Si I identifica una o varias propiedades y no se proporcionaron argumentos de tipo, el resultado es un grupo de propiedades sin expresión de destino asociada de E.
17. Si I identifica una variable compartida o una variable de instancia y no se proporcionó lista de argumentos de tipo, el resultado es una variable o un valor. Si la variable es de solo lectura y la referencia tiene lugar fuera de un constructor de la clase donde la variable se declara adecuada para la modalidad de variable (compartida o de instancia), el resultado es el valor de la variable I en el objeto al que hace referencia E. Si T es un tipo de referencia, el resultado es la variable I en el objeto al que hace referencia E. En caso contrario, si T es un tipo de valor y la expresión E se clasifica como una variable, el resultado es una variable; en caso contrario, el resultado es un valor.
18. Si I identifica un evento compartido y no se proporcionaron argumentos de tipo, el resultado es un acceso de evento con una expresión de destino asociada de E.
19. Si I identifica una constante y no se proporcionaron argumentos de tipo, el resultado es el valor de esa constante.
20. Si I identifica un miembro de enumeración y no se proporcionaron argumentos de tipo, el resultado es el valor de ese miembro de enumeración.
21. Si T es Object, el resultado es una búsqueda de miembros enlazados en tiempo de ejecución clasificada como un acceso enlazado en tiempo de ejecución con la lista de argumentos de tipo asociada y una expresión de destino asociada de E.
22. De lo contrario, E.I será una referencia de miembro no válida y se producirá un error en tiempo de compilación.

Un acceso a miembro con la forma MyClass.I(Of A) equivale a Me.I(Of A), pero todos los miembros a los que se tiene acceso aquí se tratan como si fueran no invalidables. Así, el miembro al que se tiene acceso no se verá afectado por el tipo en tiempo de ejecución del valor en que se está obteniendo acceso al miembro.

Un acceso a miembro con la forma MyBase.I(Of A) equivale a CType(Me, T).I(Of A) donde T es el tipo base directo del tipo que contiene la expresión de acceso a miembro. Todas las invocaciones de métodos que tenga se tratan como si el método al que se llama es no invalidable. La forma de acceso a miembro también se denomina *acceso base*.

En el ejemplo siguiente se muestra cómo se relacionan Me, MyBase y MyClass:

Class Base  
 Public Overridable Sub F()  
 Console.WriteLine("Base.F")  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Public Overrides Sub F()  
 Console.WriteLine("Derived.F")  
 End Sub  
  
 Public Sub G()  
 MyClass.F()  
 End Sub  
End Class  
  
Class MoreDerived  
 Inherits Derived  
  
 Public Overrides Sub F()  
 Console.WriteLine("MoreDerived.F")  
 End Sub  
  
 Public Sub H()  
 MyBase.F()  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As MoreDerived = new MoreDerived()  
  
 x.F()  
 x.G()  
 x.H()  
 End Sub  
  
End Module

Este código imprime:

MoreDerived.F  
Derived.F  
Derived.F

Cuando una expresión de acceso a miembro comienza con la palabra clave Global, la palabra clave representa el espacio de nombres sin denominación más externo, lo que es útil en los casos en que una declaración prevalece sobre un espacio de nombres envolvente. En esa situación, la palabra clave Global permite “escapar” del espacio de nombres más externo. Por ejemplo:

Class System  
End Class  
  
Module Test  
 Sub Main()  
 ' Error: Class System does not contain Console  
 System.Console.WriteLine("Hello, world!")   
  
  
 ' Legal, binds to System in outermost namespace  
 Global.System.Console.WriteLine("Hello, world!")   
 End Sub  
End Module

En el ejemplo anterior, la primera llamada al método no es válida porque el identificador System enlaza con la clase System, no con el espacio de nombres System. El único modo de acceso al espacio de nombres System es el uso de Global para escapar del espacio de nombres más externo.

Si el miembro al que se está teniendo acceso es compartido, cualquier expresión situada a la izquierda del punto es superflua y no se evalúa salvo que el acceso a miembro se haga en tiempo de ejecución. Por ejemplo, considere el siguiente código:

Class C  
 Public Shared F As Integer = 10  
End Class  
  
Module Test  
 Public Function ReturnC() As C  
 Console.WriteLine("Returning a new instance of C.")  
 Return New C()  
 End Function  
  
 Public Sub Main()  
 Console.WriteLine("The value of F is: " & ReturnC().F)  
 End Sub  
End Module

Imprime The value of F is: 10 porque no es preciso llamar a la función ReturnC para que proporcione una instancia de C y tener acceso al miembro compartido F.

MemberAccessExpression ::=  
 [ MemberAccessBase ] Period IdentifierOrKeyword  
 [ OpenParenthesis Of TypeArgumentList CloseParenthesis ]

MemberAccessBase ::=  
 Expression |  
 NonArrayTypeName |  
 Global |  
 MyClass |  
 MyBase

### Nombres de miembros y de tipos idénticos

No es infrecuente que los miembros tengan los mismos nombres que su tipo. En esa situación, sin embargo, puede producirse un ocultamiento del nombre nada conveniente:

Enum Color  
 Red  
 Green  
 Yellow  
End Enum  
  
Class Test  
 ReadOnly Property Color() As Color  
 Get  
 Return Color.Red  
 End Get  
 End Property  
  
 Shared Function DefaultColor() As Color  
 Return Color.Green ' Binds to the instance property!  
 End Function  
End Class

En el ejemplo, el nombre simple Color en DefaultColor se enlaza con la propiedad de instancia en lugar de con el tipo. Como no se puede hacer referencia a un miembro de instancia en un método compartido, normalmente daría un error.

Sin embargo, una regla especial permite el acceso al tipo en este caso. Si la expresión base de una expresión de acceso a miembro es un nombre simple que se enlaza a una constante, campo, propiedad, variable local o parámetro cuyo tipo tiene el mismo nombre, entonces la expresión base puede hacer referencia bien al miembro, bien al tipo. Esto nunca produce ambigüedad porque los miembros a los que se puede tener acceso fuera de ambos son los mismos.

### Instancias predeterminadas

En algunas situaciones, las clases derivadas de una clase base común casi siempre o siempre tienen solo una instancia única. Por ejemplo, la mayoría de las ventanas que se muestran en una interfaz de usuario solo tienen una instancia en pantalla cada vez. Para simplificar el trabajo con estos tipos de clases, Visual Basic pueden generar automáticamente instancias predeterminadas de las clases que proporcionan una instancia única, a la que se puede hacer referencia con facilidad para cada clase.

Las instancias predeterminadas siempre se crean para una familia de tipos más que para un tipo concreto. Por consiguiente, en lugar de crear una instancia predeterminada para una clase Form1 que deriva de Form, se crean instancias predeterminadas para todas las clases derivadas de Form. Esto significa que cada clase que deriva de la clase base no tiene que marcarse de modo especial para tener una instancia predeterminada.

La instancia predeterminada de una clase la representa una propiedad generada por el compilador que devuelve la instancia predeterminada de esa clase. La propiedad se generó como un miembro de una clase denominada clase de grupo que administra la asignación y la destrucción de las instancias predeterminadas de todas las clases derivadas de la clase base concreta. Por ejemplo, todas las propiedades de instancia predeterminada de las clases derivadas de Form se pueden recopilar en la clase MyForms. Si la expresión My.Forms devuelve una instancia de la clase de grupo, el código siguiente tiene acceso a las instancias predeterminadas de las clases derivadas Form1 y Form2:

Class Form1  
 Inherits Form  
 Public x As Integer  
End Class  
  
Class Form2  
 Inherits Form  
 Public y As Integer  
End Class  
  
Module Main  
 Sub Main()  
 My.Forms.Form1.x = 10  
 Console.WriteLine(My.Forms.Form2.y)  
 End Sub  
End Module

Las instancias predeterminadas no se crearán hasta que se les haga referencia por primera vez; ir a buscar la propiedad que representa la instancia predeterminada hace que esta se cree si aún no se ha creado o no se ha establecido en Nothing. Para permitir probar la existencia de una instancia predeterminada, cuando una instancia predeterminada es el destino de un operador Is o IsNot, no se creará la instancia predeterminada. Por tanto, es posible probar si una instancia predeterminada es Nothing o alguna otra referencia sin hacer que se cree la instancia predeterminada.

Las instancias predeterminadas pretenden facilitar el establecimiento de referencias desde fuera de la clase que tiene la instancia predeterminada. El uso de una instancia predeterminada desde dentro de la clase que la define puede provocar confusión sobre la instancia a la que se hace referencia, es decir, la instancia predeterminada o la instancia actual. Por ejemplo, el código siguiente modifica solo el valor x en la instancia predeterminada, incluso aunque se le llame desde otra instancia. Por tanto, el código imprime el valor 5 en lugar de 10:

Class Form1  
 Inherits Form  
  
 Public x As Integer = 5  
  
 Public Sub ChangeX()  
 Form1.x = 10  
 End Sub  
End Class  
  
Module Main  
 Sub Main()  
 Dim f As Form1 = New Form1()  
 f.ChangeX()  
 Console.WriteLine(f.x)  
 End Sub  
End Module

Para evitar este tipo de confusión, no es válido hacer referencia a una instancia predeterminada desde dentro de un método de instancia del tipo de la instancia predeterminada.

#### Instancias predeterminadas y nombres de tipos

También se puede tener acceso a una instancia predeterminada a través del nombre de su tipo. En este caso, en cualquier contexto de expresión donde el nombre de tipo no permite la expresión E, donde E representa el nombre completo de la clase con una instancia predeterminada, se cambia por E’, donde E’ representa una expresión que va en busca de la propiedad de instancia predeterminada. Por ejemplo, si las instancias predeterminadas de las clases derivadas de Form permiten el acceso a la instancia predeterminada a través del nombre de tipo, el código siguiente es equivalente al código del ejemplo anterior:

Module Main  
 Sub Main()  
 Form1.x = 10  
 Console.WriteLine(Form2.y)  
 End Sub  
End Module

Esto significa que una instancia predeterminada accesible a través del nombre de tipo también es asignable a través del nombre de tipo. Por ejemplo, el código siguiente establece la instancia predeterminada de Form1 en Nothing:

Module Main  
 Sub Main()  
 Form1 = Nothing  
 End Sub  
End Module

Observe que el significado de E.I donde E representa una clase e I representa un miembro compartido que no cambia. Una expresión así sigue teniendo acceso al miembro compartido directamente fuera de la instancia de clase y no hace referencia a la instancia predeterminada.

#### Clases de grupo

El atributo Microsoft.VisualBasic.MyGroupCollectionAttribute indica la clase de grupo para una familia de instancias predeterminadas. El atributo tiene cuatro parámetros:

El parámetro TypeToCollect especifica la clase base del grupo. Todas las clases de las que se pueden crear instancias sin parámetros de tipo abierto que derivan de un tipo con este nombre (con independencia de los parámetros de tipo) tendrán automáticamente una instancia predeterminada.

El parámetro CreateInstanceMethodName especifica el método al que se llama en la clase de grupo para crear una nueva instancia en una propiedad de instancia predeterminada.

El parámetro DisposeInstanceMethodName especifica el método al que se llama en la clase de grupo para eliminar una propiedad de instancia predeterminada si a la propiedad de instancia predeterminada se le asigna el valor Nothing.

El parámetro DefaultInstanceAlias especifica la expresión E’ que sustituye el nombre de la clase si se tiene acceso directamente a las instancias predeterminadas por su nombre de tipo. Si este parámetro es Nothing o una cadena vacía, no se tiene acceso directamente a las instancias predeterminadas por su nombre de tipo.

Anotación

En todas las implementaciones actuales del lenguaje Visual Basic, se omite el parámetro DefaultInstanceAlias, salvo en el código proporcionado por el compilador.

Es posible recopilar múltiples tipos en el mismo grupo separando los nombres de los tipos y los métodos de los primeros tres parámetros con comas. Debe haber el mismo número de elementos en cada parámetro, y los elementos de la lista se emparejan en orden. Por ejemplo, la siguiente declaración de atributo recopila los tipos que derivan de C1, C2 o C3 en un único grupo:

<Microsoft.VisualBasic.MyGroupCollection("C1, C2, C3", \_  
 "CreateC1, CreateC2, CreateC3", \_  
 "DisposeC1, DisposeC2, DisposeC3", “My.Cs")>  
Public NotInheritable Class MyCs  
 …  
End Class

La signatura del método creado debe tener la forma Shared Function <Name>(Of T As {New, <Type>})(Instance Of T) As T. El método Dispose debe tener la forma Shared Sub <Name>(Of T As <Type>)(ByRef Instance Of T). Por tanto, la clase de grupo del ejemplo de la sección anterior podría declararse así:

<Microsoft.VisualBasic.MyGroupCollection("Form", "Create", \_  
 "Dispose", "My.Forms")> \_  
Public NotInheritable Class MyForms  
 Private Shared Function Create(Of T As {New, Form}) \_  
 (Instance As T) As T  
 If Instance Is Nothing Then  
 Return New T()  
 Else  
 Return Instance  
 End If  
 End Function  
  
 Private Shared Sub Dispose(Of T As Form)(ByRef Instance As T)  
 Instance.Close()  
 Instance = Nothing  
 End Sub  
End Class

Si un archivo de origen declaró una clase derivada Form1, la clase de grupo generada equivaldría a:

<Microsoft.VisualBasic.MyGroupCollection("Form", "Create", \_  
 "Dispose", "My.Forms")> \_  
Public NotInheritable Class MyForms  
 Private Shared Function Create(Of T As {New, Form}) \_  
 (Instance As T) As T  
 If Instance Is Nothing Then  
 Return New T()  
 Else  
 Return Instance  
 End If  
 End Function  
  
 Private Shared Sub Dispose(Of T As Form)(ByRef Instance As T)  
 Instance.Close()  
 Instance = Nothing  
 End Sub  
  
 Private m\_Form1 As Form1  
  
 Public Property Form1() As Form1  
 Get  
 Return Create(m\_Form1)  
 End Get  
 Set (Value As Form1)  
 If Value IsNot Nothing AndAlso Value IsNot m\_Form1 Then  
 Throw New ArgumentException( \_  
 "Property can only be set to Nothing.")  
 End If  
 Dispose(m\_Form1)  
 End Set  
 End Property  
End Class

### Recopilación de métodos de extensión

Los métodos de extensión de la expresión de acceso a miembro E.I se recopilan juntando todos los métodos de extensión de nombre I disponibles en el contexto:

1. Primero se comprueba cada tipo anidado que contiene la expresión, comenzando en el más interno y yendo hacia el más externo.
2. Después se comprueba cada espacio de nombres anidado, comenzando por el más interno y terminando en el más externo.
3. A continuación se comprueban las importaciones en el archivo de origen.
4. Por último se comprueban las importaciones definidas por el entorno de compilación.

Un método de extensión solo se recopila si hay una conversión nativa de ampliación (widening) del tipo de la expresión de destino en el tipo del primer parámetro del método de extensión. Y a diferencia del enlace normal de expresiones de nombres simples, la búsqueda recopila todos los métodos de extensión; la recolección no se detiene cuando se encuentra uno. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Class C1  
End Class

Namespace N1  
 Module N1C1Extensions  
 <Extension> \_  
 Sub M1(c As C1, x As Integer)  
 End Sub  
 End Module  
End Namespace  
  
Namespace N1.N2  
 Module N2C1Extensions  
 <Extension> \_  
 Sub M1(c As C1, y As Double)  
 End Sub  
 End Module  
End Namespace  
  
Namespace N1.N2.N3  
 Module Test  
 Sub Main()  
 Dim x As New C1()  
  
 ' Calls N1C1Extensions.M1  
 x.M1(10)  
 End Sub  
 End Module  
End Namespace

En este ejemplo, aunque N2C1Extensions.M1 está antes que N1C1Extensions.M1, ambos se consideran métodos de extensión. Una vez recopilados todos los métodos de extensión, se currifican. Al currificar, se toma el destino de la llamada al método de extensión y se aplica a la llamada al método de extensión, lo que da como resultado una nueva firma de método a la que se ha quitado el primer parámetro (porque se ha especificado). Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Module Ext1  
 <Extension> \_  
 Sub M(x As Integer, y As Integer)  
 End Sub  
End Module  
  
Module Ext2  
 <Extension> \_  
 Sub M(x As Integer, y As Double)  
 End Sub  
End Module  
  
Module Main  
 Sub Test()  
 Dim v As Integer = 10  
  
 ' The curried method signatures considered are:  
 ' Ext1.M(y As Integer)  
 ' Ext2.M(y As Double)  
 v.M(10)  
 End Sub  
End Module

En ese ejemplo, el resultado currificado de aplicar v a Ext1.M es la signatura del método Sub M(y As Integer).

Además de quitar el primer parámetro del método de extensión, la currificación también quita todos los parámetros de tipo del método que sean parte del tipo del primer parámetro. Cuando se currifica un método de extensión con parámetros de tipo de método, se aplica la inferencia de tipos al primer parámetro y el resultado se fija para todos los parámetros de tipo que se infieren. Si la inferencia de tipos da error, el método se pasa por alto. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Module Ext1  
 <Extension> \_  
 Sub M(Of T, U)(x As T, y As U)  
 End Sub  
End Module  
  
Module Ext2  
 <Extension> \_  
 Sub M(Of T)(x As T, y As T)  
 End Sub  
End Module  
  
Module Main  
 Sub Test()  
 Dim v As Integer = 10  
  
 ' The curried method signatures considered are:  
 ' Ext1.M(Of U)(y As U)  
 ' Ext2.M(y As Integer)  
 v.M(10)  
 End Sub  
End Module

En el ejemplo, el resultado currificado de aplicar v a Ext1.M es la signatura del método Sub M(Of U)(y As U), porque el parámetro de tipo T se infiere como un resultado de la currificación y ahora es fijo. Como el parámetro de tipo U no fue inferido como parte de la currificación, continúa siendo un parámetro abierto. De igual modo, como el parámetro de tipo T se infiere como un resultado de aplicar v a Ext2.M, el tipo del parámetro y se fija como Integer. No se inferirá que es de ningún otro tipo. Cuando se currifica la signatura, también se aplican todas las restricciones excepto New. Si las restricciones no se satisfacen o dependen de un tipo que no fue inferido como parte de la currificación, el método de extensión se pasa por alto. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Module Ext1  
 <Extension> \_  
 Sub M1(Of T As Structure)(x As T, y As Integer)  
 End Sub  
  
 <Extension> \_  
 Sub M2(Of T As U, U)(x As T, y As U)  
 End Sub  
End Module  
  
Module Main  
 Sub Test()  
 Dim s As String = "abc"  
  
 ' Error: String does not satisfy the Structure constraint  
 s.M1(10)  
  
 ' Error: T depends on U, which cannot be inferred  
 s.M2(10)  
 End Sub  
End Module

Anotación

Una de las razones principales para currificar los métodos de extensión es que permite que las expresiones de consulta infieran el tipo de la iteración antes de evaluar los argumentos de un método de modelo de consultas. Ya que la mayoría de los métodos de modelo de consultas adoptan expresiones lambda, las cuales requieren la inferencia de tipos, se simplifica ampliamente el proceso de evaluar una expresión de consulta.

A diferencia de la herencia de interfaz normal, los métodos de extensión que amplían dos interfaces que no están relacionadas entre sí están disponibles, siempre que no tengan la misma firma currificada:

Imports System.Runtime.CompilerServices  
  
Interface I1  
End Interface  
  
Interface I2  
End Interface  
  
Class C1  
 Implements I1, I2  
End Class  
  
Module I1Ext  
 <Extension> \_  
 Sub M1(i As I1, x As Integer)  
 End Sub  
  
 <Extension> \_  
 Sub M2(i As I1, x As Integer)  
 End Sub  
End Module  
  
Module I2Ext  
 <Extension> \_  
 Sub M1(i As I2, x As Integer)  
 End Sub  
  
 <Extension> \_  
 Sub M2(I As I2, x As Double)  
 End Sub  
End Module  
  
Module Main  
 Sub Test()  
 Dim c As New C1()  
  
 ' Error: M is ambiguous between I1Ext.M1 and I2Ext.M1.  
 c.M1(10)  
  
 ' Calls I1Ext.M2  
 c.M2(10)  
 End Sub  
End Module

Por último es importante recordar que los métodos de extensión no se tienen en cuenta cuando se efectúa el enlace en tiempo de ejecución:

Module Test  
 Sub Main()  
 Dim o As Object = …  
  
 ' Ignores extension methods  
 o.M1()  
 End Sub  
End Module

## Expresiones de acceso a miembros de diccionario

Una expresión de acceso a miembros de diccionario se usa para buscar un miembro de una colección. Un acceso a miembro de diccionario adopta la forma E!I, donde E es una expresión que se clasifica como valor e I como identificador. El tipo de la expresión debe tener una propiedad predeterminada indizada por un parámetro String único. La expresión de acceso a miembros de diccionario E!I se transforma en la expresión E.D("I"), donde D es la propiedad predeterminada de E. Por ejemplo:

Class Keys  
 Public ReadOnly Default Property Item(s As String) As Integer  
 Get  
 Return 10  
 End Get  
 End Property   
End Class  
  
Module Test  
 Sub Main()  
 Dim x As Keys = new Keys()  
 Dim y As Integer  
 ' The two statements are equivalent.  
 y = x!abc  
 y = x("abc")  
 End Sub  
End Module

Si se especifica un signo de exclamación sin ninguna expresión, se supone que es la expresión de la instrucción inmediata que contiene la instrucción With. Si no hay ninguna instrucción contenedora With, se produce un error en tiempo de compilación.

DictionaryAccessExpression ::= [ Expression ] ! IdentifierOrKeyword

## Expresiones de invocación

Una expresión de invocación consiste en un destino de invocación y una lista de argumentos opcional. La expresión de destino debe clasificarse como un grupo de métodos o un valor cuyo tipo sea un tipo delegado. Si la expresión de destino es un valor cuyo tipo es un tipo de delegado, el destino de la expresión de invocación pasa a ser el grupo de métodos para el miembro Invoke del tipo de delegado y la expresión de destino pasa a ser la expresión de destino asociada del grupo de métodos.

Una lista de argumentos tiene dos secciones: argumentos de posición y argumentos con nombre. Los argumentos de posición (o posicionales) son expresiones y deben preceder a los argumentos con nombre. Los argumentos con nombre comienzan por un identificador que puede coincidir con palabras clave seguido de := y una expresión.

Si el grupo de métodos solo contiene un método accesible, incluidos los métodos de extensión y de instancia, y ese método no toma ningún argumento y es una función, entonces el grupo de métodos se interpreta como una expresión de invocación con una lista de argumentos vacía y el resultado se usa como el destino de una expresión de invocación con la lista(s) de argumentos proporcionada. Por ejemplo:

Class C1  
 Function M1() As Integer()  
 Return New Integer() { 1, 2, 3 }  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim c As New C1()  
  
 ' Prints 3  
 Console.WriteLine(c.M1(2))  
 End Sub  
End Module

De lo contrario, se aplica la resolución de sobrecarga a los métodos para elegir el más aplicable de la lista o listas de argumentos dada. Si el método más aplicable es una función, el resultado de la expresión de invocación se clasifica como un tipo de valor como el tipo devuelto de la función. Si el método más aplicable es una subrutina, el resultado se clasifica como vacío. Si el método más aplicable es un método parcial que no tiene cuerpo, la expresión de invocación se pasa por alto y el resultado se clasifica como vacío.

En expresiones de invocación enlazada en tiempo de compilación, los argumentos se evalúan en el orden en que se declaran los parámetros correspondientes en el método de destino. En expresiones de invocación enlazada en tiempo de ejecución, se evalúan en el orden en que aparecen en las expresiones de acceso a miembros: vea la Sección 11.3, Expresiones en tiempo de ejecución.

InvocationExpression ::= Expression [ OpenParenthesis [ ArgumentList ] CloseParenthesis ]

ArgumentList ::=  
 PositionalArgumentList |  
 PositionalArgumentList Comma NamedArgumentList | NamedArgumentList

PositionalArgumentList ::=  
 [ Expression ] |  
 PositionalArgumentList Comma [ Expression ]

NamedArgumentList ::=  
 IdentifierOrKeyword ColonEquals Expression |  
 NamedArgumentList Comma IdentifierOrKeyword ColonEquals Expression

### Resolución de métodos sobrecargados

En la práctica, las reglas para determinar la resolución de sobrecarga intentan buscar la sobrecarga “más cercana” a los argumentos suministrados. Si hay un método cuyos tipos de parámetro coinciden con los tipos de argumento, entonces el método es obviamente el más cercano. Exceptuando eso, un método es más cercano que otro si todos sus tipos de parámetro son más restringidos (o iguales) que los tipos de parámetro del otro método. Si los parámetros de ninguno de los métodos son más restringidos que el otro, no hay modo de determinar qué método está más cercano a los argumentos.

Nota   La resolución de sobrecarga no tiene en cuenta el tipo de valor devuelto esperado del método.

Obsérvese también que debido a la sintaxis de parámetro con nombre, el orden de los parámetros formales y reales puede no ser el mismo.

Dado un grupo de métodos, el método más aplicable del grupo para una lista de argumentos se determina con los pasos siguientes. Si después de un paso concreto, no queda ningún miembro en el grupo, se produce un error en tiempo de compilación. Si solo queda un miembro en el conjunto, ese es el más aplicable. Los pasos son:

1. Primero, si no se han proporcionado argumentos de tipo, se aplica la inferencia de tipos a todos los métodos que tengan parámetros de tipo. Si la inferencia de tipos se produce correctamente, los argumentos de tipo inferidos se usan para ese método concreto. Si la inferencia de tipos da error para un método, ese método se elimina del conjunto.
2. Después, se eliminan todos los miembros del conjunto que sean inaccesibles o no aplicables (11.8.2) a la lista de argumentos.
3. Después, si uno o más argumentos son expresiones lambda o AddressOf, calcule los *niveles de relajación de delegados* de cada uno de estos argumentos como se indica a continuación. Si el menor nivel de relajación de delegados en N es inferior al nivel de relajación de delegados más bajo en M, elimine N del conjunto.  
     
   Los niveles de relajación de delegados son los siguientes:
4. *(1) Error de* *nivel de relajación de delegados*: si la expresión AddressOf o lambda no se puede convertir en el tipo del delegado.  
     
   *(2) Relajación de delegados restrictiva del tipo de valor devuelto o los parámetros*: si el argumento es AddressOf o lambda con un tipo declarado y la conversión del tipo de valor devuelto en el tipo de valor devuelto del delegado es restrictiva; o bien, si el argumento es una expresión lambda regular y la conversión de cualquiera de sus expresiones return en el tipo de valor devuelto del delegado es restrictiva, o si el argumento es una expresión lambda asincrónica y el tipo de valor devuelto del delegado es Task(Of T) y la conversión de cualquiera de sus expresiones return en T es restrictiva; o si el argumento es una expresión lambda de iteración y el tipo de valor devuelto del delegado es IEnumerator(Of T) o IEnumerable(Of T) y la conversión de cualquiera de sus operandos yield en T es restrictiva.  
      
   *(3) Relajación de delegados de ampliación en delegado sin signatura*: si el tipo de delegado es System.Delegate, System.MultiCastDelegate o System.Object.  
     
   *(4) Relajación de delegados de colocación de valor devuelto o argumentos*: si el argumento es AddressOf o lambda con un tipo de valor devuelto declarado y el tipo de delegado no tiene un tipo de valor devuelto; o bien, si el argumento es una expresión lambda con una o más expresiones return y el tipo de delegado no tiene un tipo de valor devuelto; o bien si el argumento es AddressOf o lambda sin parámetros y el tipo de delegado tiene parámetros.  
     
   *(5) Relajación de delegados de ampliación del tipo de valor devuelto*: si el argumento es AddressOf o lambda con un tipo de valor devuelto declarado y hay una conversión de ampliación de su tipo de valor devuelto en el del delegado; o bien, si el argumento es una expresión lambda regular donde la conversión de todas las expresiones return en el tipo de valor devuelto del delegado es de ampliación o una identidad con al menos una ampliación; o bien, si el argumento es una expresión lambda asincrónica y el delegado es Task(Of T) o Task y la conversión de todas las expresiones return en T/Object respectivamente es de ampliación o una identidad con al menos una ampliación; o bien, si el argumento es una expresión lambda de iteración y el delegado es IEnumerator(Of T), IEnumerable(Of T), IEnumerator o IEnumerable y la conversión de todas las expresiones return en T/Object es de ampliación o una identidad con al menos una ampliación.  
     
   *(6) Relajación de delegados de identidad*: si el argumento es de tipo AddressOf o lambda que coincide con el delegado exactamente, sin aplicar una ampliación, restricción o colocación de parámetros, valores devueltos u operandos yield. A continuación, si algunos miembros del conjunto no requieren que se apliquen conversiones narrowing a cualquiera de los argumentos, elimine todos los miembros que lo hacen. Por ejemplo:

Sub f(x As Object)  
End Sub

Sub f(x As Short)  
End Sub

Sub f(x As Short())  
End Sub

f("5") ' picks the Object overload, since String->Short is narrowing  
f(5) ' picks the Object overload, since Integer->Short is narrowing  
f({5}) ' picks the Object overload, since Integer->Short is narrowing  
f({}) ' a tie-breaker rule subsequent to [3] picks the Short() overload

1. A continuación, la eliminación se realiza en función de la restricción de la siguiente forma. (Observe que, si el valor de Option Strict es On, todos los miembros que requieren una conversión narrowing ya se han considerado no aplicables (11.8.2) y se eliminaron en el paso 2).
   1. Si algunos miembros de instancia del conjunto solamente requieren conversiones narrowing (de restricción) donde el tipo de argumento es Object, elimine todos los miembros restantes.
   2. Si el conjunto contiene más de un miembro que requiere la conversión narrowing solo de Object, la expresión de destino de invocación se reclasifica como un acceso a método enlazado en tiempo de ejecución (y se obtiene un error si el tipo que contiene el grupo de métodos es una interfaz o si cualquiera de los miembros aplicables fuera miembro de extensión).
   3. Si hay algún candidato que solamente requiera conversiones narrowing de literales numéricos, elija el candidato más específico entre todos los restantes mediante los pasos siguientes. Si el elemento elegido solamente requiere conversiones narrowing de literales numéricos, se selecciona como resultado de una resolución de sobrecarga; de lo contrario, es un error.

Anotación

La justificación de esta regla es que un programa débilmente tipado (es decir, la mayoría de las variables se declaran Object), la resolución de sobrecarga puede ser difícil cuando muchas conversiones de Object son de restricción (narrowing). En lugar de que la resolución de sobrecarga dé error en muchas situaciones (que requiere que los argumentos a la invocación del método estén fuertemente tipados), la resolución del método sobrecargado adecuado que hay que invocar se retrasa hasta el tiempo de ejecución. Esto permite que la llamada débilmente tipada sea correcta sin ninguna conversión adicional.

Un efecto secundario infortunado, sin embargo, es que efectuar la llamada enlazada en tiempo de ejecución exige convertir el destino de la llamada en Object. En el caso de un valor de estructura, esto significa que el valor debe sufrir una conversión boxing en un temporal. Si el método finalmente llamado intenta cambiar un campo de la estructura, este cambio se perderá cuando el método vuelva.

Las interfaces están excluidas de esta regla especial porque el enlace en tiempo de ejecución siempre se resuelve con los miembros de la clase o tipo de estructura en tiempo de ejecución, que pueden tener nombres distintos de los nombres de los miembros de las interfaces que implementan.

1. Después, si queda en el conjunto algún método de instancia que no requiere la conversión narrowing, elimine todos los métodos de extensión del conjunto. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Class C3  
 Sub M1(d As Integer)  
 End Sub  
End Class  
  
Module C3Extensions  
 <Extension> \_  
 Sub M1(c3 As C3, c As Long)  
 End Sub  
  
 <Extension> \_  
 Sub M1(c3 As C3, c As Short)  
 End Sub  
End Module  
  
Module Test  
 Sub Main()  
 Dim c As New C3()  
 Dim sVal As Short = 10  
 Dim lVal As Long = 20  
  
 ' Calls C3.M1, since C3.M1 is applicable.  
 c.M1(sVal)  
  
 ' Calls C3Extensions.M1 since C3.M1 requires a narrowing conversion  
 c.M1(lVal)  
 End Sub  
End Module

Anotación

Los métodos de extensión se omiten si existen métodos de instancia aplicables para garantizar que agregar una importación (que puede traer nuevos métodos de extensión al ámbito) no provocará una llamada a ningún método de instancia existente para volver a enlazar con un método de extensión. Dado el amplio ámbito de algunos métodos de extensión (es decir, los definidos en interfaces o parámetros de tipo), este enfoque es más seguro que enlazar con métodos de extensión.

1. Después, si dados dos miembros cualesquiera del conjunto, M y N, M es más *específico* (Sección 11.8.1.1) que N dada la lista de argumentos, se elimina N del conjunto. Si queda más de un miembro en el conjunto y no todos son igualmente específicos dada la lista de argumentos, se produce un error en tiempo de compilación.
2. De lo contrario, dados dos miembros cualesquiera del conjunto, M y N, se aplican las siguientes reglas de desempate, en orden:

If M does not have a ParamArray parameter but N does, or if both do but M passes fewer arguments into the ParamArray parameter than N does, then eliminate N from the set. For example:Module Test  
 Sub F(a As Object, ParamArray b As Object())  
 Console.WriteLine("F(Object, Object())")  
 End Sub  
  
 Sub F(a As Object, b As Object, ParamArray c As Object())  
 Console.WriteLine("F(Object, Object, Object())")  
 End Sub  
  
 Sub G(Optional a As Object = Nothing)  
 Console.WriteLine("G(Object)")  
 End Sub  
  
 Sub G(ParamArray a As Object())  
 Console.WriteLine("G(Object())")  
 End Sub Sub Main()  
 F(1)  
 F(1, 2)  
 F(1, 2, 3)  
 G()  
 End Sub  
End Module

El ejemplo produce el siguiente resultado:

F(Object, Object())  
F(Object, Object, Object())  
F(Object, Object, Object())  
G(Object)

Anotación

Cuando una clase declara un método con un parámetro paramarray, se suelen incluir también algunas de las formas expandidas como métodos normales. Así es posible evitar la asignación de una instancia de matriz que se produce cuando se invoca una forma expandida de un método con parámetro paramarray.

1. Si M se define en más de un tipo derivado que N, se elimina N del conjunto. Por ejemplo:

Class Base  
 Sub F(Of T, U)(x As T, y As U)  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Overloads Sub F(Of T, U)(x As U, y As T)  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim d As New Derived()  
  
 ' Calls Derived.F  
 d.F(10, 10)  
 End Sub  
End Module

Esta regla también se aplica a los tipos en los que se definen los métodos de extensión. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Class Base  
End Class  
  
Class Derived  
 Inherits Base  
End Class  
  
Module BaseExt  
 <Extension> \_  
 Sub M(b As Base, x As Integer)  
 End Sub  
End Module  
  
Module DerivedExt  
 <Extension> \_  
 Sub M(d As Derived, x As Integer)  
 End Sub  
End Module  
  
Module Test  
 Sub Main()  
 Dim b As New Base()  
 Dim d As New Derived()  
  
 ' Calls BaseExt.M  
 b.M(10)  
  
 ' Calls DerivedExt.M   
 d.M(10)  
 End Sub  
End Module

1. Si M y N son métodos de extensión y el tipo de destino de M es una clase o estructura y el de N es una interfaz, se elimina N del conjunto. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Interface I1  
End Interface  
  
Class C1  
 Implements I1  
End Class  
  
Module Ext1  
 <Extension> \_  
 Sub M(i As I1, x As Integer)  
 End Sub  
End Module  
  
Module Ext2  
 <Extension> \_  
 Sub M(c As C1, y As Integer)  
 End Sub  
End Module  
  
Module Test  
 Sub Main()  
 Dim c As New C1()  
  
 ' Calls Ext2.M, because Ext1.M is hidden since it extends  
 ' an interface.  
 c.M(10)  
  
 ' Calls Ext1.M  
 CType(c, I1).M(10)  
 End Sub  
End Module

1. Si M y N son métodos de extensión, el tipo de destino de M y N es idéntico tras la sustitución de los parámetros de tipo y el tipo de destino de M antes de dicha sustitución no contiene parámetros de tipo pero el tipo de destino de N sí contiene, y después tiene menos parámetros de tipo que el tipo de destino de N, se elimina N del conjunto. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Module Module1  
 Sub Main()  
 Dim x As Integer = 1  
 x.f(1) ' Calls first "f" extension method

Dim y As New Dictionary(Of Integer, Integer)  
 y.g(1) ' Ambiguity error  
 End Sub

<Extension()> Sub f(x As Integer, z As Integer)  
 End Sub

<Extension()> Sub f(Of T)(x As T, z As T)  
 End Sub  
  
 <Extension()> Sub g(Of T)(y As Dictionary(Of T, Integer), z As T)  
 End Sub

<Extension()> Sub g(Of T)(y As Dictionary(Of T, T), z As T)  
 End Sub  
End Module

1. Antes de haber sustituido los argumentos de tipo, si M es *menos genérico* (Sección 11.8.1.2) que N, se elimina N del conjunto.
2. Si M no es un método de extensión y N sí, se elimina N del conjunto.
3. Si M y N son métodos de extensión y M se encuentra antes que N (Sección 11.6.3), se elimina N del conjunto. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Class C1  
End Class  
  
Namespace N1  
 Module N1C1Extensions  
 <Extension> \_  
 Sub M1(c As C1, x As Integer)  
 End Sub  
 End Module  
End Namespace  
  
Namespace N1.N2  
 Module N2C1Extensions  
 <Extension> \_  
 Sub M1(c As C1, y As Integer)  
 End Sub  
 End Module  
End Namespace  
  
Namespace N1.N2.N3  
 Module Test  
 Sub Main()  
 Dim x As New C1()  
  
 ' Calls N2C1Extensions.M1  
 x.M1(10)  
 End Sub  
 End Module  
End Namespace

Si los métodos de extensión se encuentran en el mismo paso, entonces esos métodos son ambiguos. Siempre se puede romper la ambigüedad de la llamada utilizando el nombre del módulo estándar que contiene el método de extensión y llamando al método de extensión como si fuera un miembro normal. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Class C1  
End Class  
  
Module C1ExtA  
 <Extension> \_  
 Sub M(c As C1)  
 End Sub  
End Module  
  
Module C1ExtB  
 <Extension> \_  
 Sub M(c As C1)  
 End Sub  
End Module  
  
Module Main  
 Sub Test()  
 Dim c As New C1()  
  
 C1.M() ' Ambiguous between C1ExtA.M and BExtB.M  
 C1ExtA.M(c) ' Calls C1ExtA.M  
 C1ExtB.M(c) ' Calls C1ExtB.M  
 End Sub  
End Module

1. Si tanto M como N requieren la inferencia de tipos para generar argumentos de tipo y M no precisa determinar el tipo dominante de ninguno de sus argumentos de tipo (es decir, cada argumento de tipo inferido como un único tipo), pero N sí, se elimina N del conjunto.

Anotación

Esta regla asegura que la resolución de sobrecarga que era correcta en versiones anteriores (cuando la inferencia de tipos múltiples para un argumento de tipo provoca un error), siga dando los mismos resultados.

1. Si se efectúa una resolución de sobrecarga para solucionar el destino de una expresión de creación de delegado a partir de una expresión AddressOf, y tanto el delegado como M son funciones, mientra que N es una subrutina, se elimina N del conjunto. Igualmente, si tanto el delegado como M son subrutinas y N es una función, se elimina N del conjunto.
2. Si M no usó ningún valor predeterminado de parámetro opcional en lugar de argumentos explícitos pero N sí, se elimina N del conjunto.
3. Antes de haber sustituido los argumentos de tipo, si M es *más genérico* (Sección 11.8.1.3) que N, elimine N del conjunto.
4. De lo contrario, la llamada es ambigua y se genera un error en tiempo de compilación.

#### Especificidad de los miembros o los tipos dada una lista de argumentos

Un miembro M se considera igualmente específico que N, dada una lista de argumentos A, si sus signaturas son las mismas o si cada parámetro de tipo de M es el mismo que el tipo de parámetro correspondiente en N.

Anotación

Dos miembros pueden terminar en un grupo de métodos con la misma firma debido a los métodos de extensión. Dos miembros pueden ser igualmente específicos pero no tener la misma signatura debido a los parámetros de tipo o a la expansión paramarray.

Un miembro M se considera más específico que N si sus signaturas son diferentes y al menos un tipo de parámetro de M es más específico que un tipo de parámetro de N, y ningún tipo de parámetro de N es más específico que un tipo de parámetro de M. Dado un par de parámetros Mj y Nj que coinciden con un argumento Aj, el tipo de Mj se considera *más específico* que el de Nj si se cumple alguna de las condiciones siguientes:

1. Hay una conversión widening del tipo Mj en el tipo Nj,

Anotación

Observe que como los tipos de parámetros se están comparando con independencia del argumento real en este caso, no se tiene en cuenta la conversión widening de expresiones constantes en un tipo numérico en el que encaje el valor.

1. Aj es el literal 0, Mj es un tipo numérico y Nj es un tipo enumerado

Anotación

Esta regla es necesaria porque el literal 0 se amplía a cualquier tipo enumerado. Dado que un tipo enumerado se amplia hasta su tipo subyacente, esto significa que la resolución de sobrecarga en 0 preferirá, de manera predeterminada, los tipos enumerados a los tipos numéricos. Recibimos muchos comentarios que indicaban que este comportamiento era contrario a la intuición.

1. Mj y Nj son ambos tipos numéricos y Mj precede a Nj en la lista de valores  
   Byte, SByte, Short, UShort, Integer, UInteger, Long, ULong, Decimal, Single, Double

Anotación

La regla sobre tipos numéricos es útil porque los tipos numéricos firmados y sin firmar de un tamaño concreto solo tienen conversiones narrowing entre ellos. La regla anterior desempata los dos tipos en favor del tipo numérico más "natural". Esto es especialmente importante cuando se efectúan resoluciones de sobrecarga en un tipo que se amplía a los tipos numéricos firmados y no firmados de un tamaño concreto (por ejemplo, un literal numérico que encaja en ambos).

1. Mj y Nj son tipos de función de delegado y el tipo devuelto de Mj es más específico que el de Nj. Si Aj se clasifica como método lambda, y Mj o Nj es System.Linq.Expressions.Expression(Of T), entonces el argumento de tipo del tipo (suponiendo que sea un tipo de delegado) se sustituye por el tipo que se está comparando.
2. Mj es idéntico al tipo de Aj y Nj no lo es.

Anotación

Es interesante observar que la regla anterior es ligeramente distinta en C#, porque C# requiere que los tipos de función de delegado tengan listas de parámetros idénticas antes de comparar los tipos devueltos, y Visual Basic no.

#### Calidad de genéricos

Se determina que un miembro M es menos genérico que un miembro N del modo siguiente:

1. Si por cada par de parámetros coincidentes Mj y Nj, Mj es menos o igualmente genérico que Nj en relación con los parámetros de tipo del método, y por lo menos un Mj es menos genérico en relación con los parámetros de tipo en el método.
2. De lo contrario, si por cada par de parámetros coincidentes Mj y Nj, Mj es menos o igualmente genérico que Nj en relación con los parámetros de tipo del tipo, y por lo menos un Mj es menos genérico en relación con los parámetros de tipo del tipo, entonces M es menos genérico que N.

Se considera que un parámetro M es tan genérico como un parámetro N si sus tipos Mt y Nt hacen ambos referencia a parámetros de tipo o ninguno de ellos hace referencia a parámetros de tipo. Se considera que M es menos genérico que N si Mt no hace referencia a un parámetro de tipo y Nt sí.

Por ejemplo:

Class C1(Of T)  
 Sub S1(Of U)(x As U, y As T)  
 End Sub  
  
 Sub S1(Of U)(x As U, y As U)  
 End Sub  
  
 Sub S2(x As Integer, y As T)  
 End Sub  
  
 Sub S2(x As T, y As T)  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As C1(Of Integer) = New C1(Of Integer)  
  
 x.S1(10, 10) ' Calls S1(U, T)  
 x.S2(10, 10) ' Calls S2(Integer, T)  
 End Sub  
End Module

Los parámetros de tipo de los métodos de extensión que se fijaron durante la currificación se consideran parámetros de tipo en el tipo, no parámetros de tipo en el método. Por ejemplo:

Imports System.Runtime.CompilerServices  
  
Module Ext1  
 <Extension> \_  
 Sub M1(Of T, U)(x As T, y As U, z As U)  
 End Sub  
End Module  
  
Module Ext2  
 <Extension> \_  
 Sub M1(Of T, U)(x As T, y As U, z As T)  
 End Sub  
End Module  
  
Module Test  
 Sub Main()  
 Dim i As Integer = 10  
  
 i.M1(10, 10)  
 End Sub  
End Module

#### Nivel de generalidad

Un miembro M está determinado para ser más genérico que un miembro N si, por cada par de parámetros coincidentes Mj y Nj, Mj es más o igualmente *genérico* que Nj y al menos un Mj es más genérico. El nivel de generalidad se define de la siguiente forma:

1. Cualquier elemento que no sea un parámetro de tipo es más genérico que un parámetro de tipo;
2. De manera recursiva, un tipo construido es más genérico que otro tipo construido (con el mismo número de argumentos de tipo) si al menos un argumento de tipo es más genérico y no hay argumentos de tipo menos genéricos que el argumento de tipo correspondiente.
3. Un tipo de matriz es más genérico que otro tipo de matriz (con el mismo número de dimensiones) si el tipo de elemento del primero es más genérico que el tipo de elemento del segundo.

Por ejemplo:

Module Test

Sub f(Of T)(x As Task(Of T))  
 End Sub

Sub f(Of T)(x As T)  
 End Sub

Sub Main()  
 Dim x As Task(Of Integer) = Nothing  
 f(x) ' Calls the first overload  
 End Sub  
End Module

### Aplicabilidad a lista de argumentos

Un método es aplicable a un conjunto de argumentos de tipo, argumentos de posición y argumentos con nombre si se puede invocar usando las dos listas de argumentos. Las listas de argumentos se emparejan con las listas de parámetros del modo siguiente:

1. Primero, se empareja cada argumento de posición para hacer la lista de los parámetros de método. Si hay más argumentos de posición que parámetros y el último parámetro no es un paramarray, el método no es aplicable. De lo contrario, el parámetro paramarray se expande con parámetros del elemento de tipo de paramarray para emparejar el número de argumentos de posición. Si se omite un argumento de posición que aparecería en paramarray, el método no es aplicable.
2. Después, se empareja cada argumento con nombre con un parámetro con el nombre dado. Si uno de los argumentos con nombre no coincide, coincide con un parámetro paramarray o con un argumento que ya ha coincidido con otro argumento de posición o con nombre, el método no es aplicable.
3. A continuación, si los argumentos de tipo se han especificado, se emparejan con la lista de parámetros de tipo. Si las dos listas no tienen la misma cantidad de elementos, el método no es aplicable, salvo que la lista de argumentos de tipo esté vacía. Si la lista de argumentos de tipo está vacía, se emplea la inferencia de tipos para tratar de inferir la lista de argumentos de tipo. Si la inferencia de tipos da error, el método no es aplicable. De lo contrario, los argumentos de tipo se completan en el lugar de los parámetros de tipo en la signatura. Si los parámetros que no se han emparejado no son opcionales, el método no es aplicable.
4. Si las expresiones de argumentos no son implícitamente convertibles en los tipos de parámetros con los que coinciden, el método no es aplicable.
5. Si un parámetro es ByRef y no hay una conversión implícita del tipo de parámetro en el tipo de argumento, el método no es aplicable.
6. Si los argumentos de tipo incumplen las restricciones del método (incluidos los argumentos de tipo inferidos del paso 3), el método no es aplicable. Por ejemplo:

Module Module1  
 Sub Main()  
 f(Of Integer)(New Exception)  
 ' picks the first overload (narrowing),  
 ' since the second overload (widening) violates constraints   
 End Sub  
  
 Sub f(Of T)(x As IComparable)  
 End Sub

Sub f(Of T As Class)(x As Object)  
 End Sub  
End Module

Si una expresión de argumento única coincide con un parámetro paramarray y el tipo de la expresión de argumento es convertible en el tipo del parámetro paramarray y en el tipo del elemento paramarray, el método es aplicable en la forma expandida y no expandida, con dos excepciones. Si la conversión desde el tipo de la expresión de argumentos en el tipo paramarray es narrowing (de restricción), el método solo es aplicable en la forma expandida. Si la expresión de argumento es el literal Nothing, el método solo es aplicable en la forma no expandida. Por ejemplo:

Module Test  
 Sub F(ParamArray a As Object())  
 Dim o As Object  
  
 For Each o In a  
 Console.Write(o.GetType().FullName)  
 Console.Write(" ")  
 Next o  
 Console.WriteLine()  
 End Sub  
  
 Sub Main()  
 Dim a As Object() = { 1, "Hello", 123.456 }  
 Dim o As Object = a  
  
 F(a)  
 F(CType(a, Object))  
 F(o)  
 F(CType(o, Object()))  
 End Sub  
End Module

El ejemplo produce el siguiente 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 widening del tipo de argumento en el tipo de parámetro (ambos son de tipo Object()), y el argumento se pasa como un parámetro de valor normal. En la segunda y tercera invocación, la forma normal de F no es aplicable porque no existe una conversión widening del tipo del argumento en el tipo del parámetro (las conversiones de Object en Object() son narrowing). Sin embargo, la forma expandida de F es aplicable y la invocación crea un Object() de un elemento. El elemento único de la matriz se inicializa con el valor del argumento dado (que es una referencia a un Object()).

### Pasar y elegir argumentos para parámetros opcionales

Si un parámetro es un parámetro de valor, la expresión de argumento coincidente debe clasificarse como valor. El valor se convierte en el tipo del parámetro y se pasa como el parámetro en tiempo de ejecución. Si el parámetro es un parámetro de referencia y la expresión de argumento coincidente se clasifica como una variable cuyo tipo es el mismo que el parámetro, se pasa una referencia a la variable con el parámetro en tiempo de ejecución.

De lo contrario, si la expresión de argumento coincidente se clasifica como variable, valor o propiedad de acceso, entonces se asigna una variable temporal del tipo del parámetro. Antes de la invocación del método en tiempo de ejecución, la expresión de argumento se reclasifica como valor, convertido en el tipo del parámetro y asignado a la variable temporal. Luego se pasa una referencia a la variable temporal como el parámetro. Después de evaluar la invocación del método, si la expresión de argumento se clasifica como variable o como acceso de propiedad, la variable temporal se asigna a la expresión de variable o la expresión de acceso de propiedad. Si la expresión de acceso de propiedad no tiene descriptor de acceso Set, no se lleva a cabo la asignación.

Para los parámetros opcionales donde no se ha proporcionado un argumento, el compilador elige argumentos como se explica a continuación. En todos los casos, lo comprueba con el tipo de parámetro tras la sustitución de tipos genéricos.

* 1. Si el parámetro opcional tiene el atributo System.Runtime.CompilerServices.CallerLineNumber, la invocación se realiza desde una ubicación del código fuente y un literal numérico que representa el número de línea de dicha ubicación tiene una conversión intrínseca en el tipo de parámetro, se usa el literal numérico. Si la invocación ocupa varias líneas, la elección de la línea que se va a usar depende de la implementación.
  2. Si el parámetro opcional tiene el atributo System.Runtime.CompilerServices.CallerFilePath, la invocación se realiza desde una ubicación del código fuente y un literal de cadena que representa la ruta de acceso al archivo de dicha ubicación tiene una conversión intrínseca en el tipo de parámetro, se usa el literal de cadena. El formato de la ruta de acceso al archivo depende de la implementación.
  3. Si el parámetro opcional tiene el atributo System.Runtime.CompilerServices.CallerMemberName, la invocación se realiza en el cuerpo de un miembro de tipo o en un atributo aplicado a cualquier parte de dicho miembro y un literal de cadena que representa el nombre de ese miembro tiene una conversión intrínseca en el tipo de parámetro, se usa el literal de cadena. Para las invocaciones que forman parte de descriptores de acceso a propiedad o controladores de eventos personalizados, el nombre de miembro que se usa es el de la propiedad o evento en sí. Para las invocaciones que forman parte de un operador o constructor, se usa un nombre específico de la implementación.

Si ninguna de las opciones anteriores es aplicable, se usa el valor predeterminado del parámetro opcional (o Nothing si no se ha proporcionado un valor predeterminado). Si más de una de las opciones anteriores es aplicable, la elección de aquella que se va a usar dependerá de la implementación.

Anotación

Los atributos CallerLineNumber y CallerFilePath son útiles para el registro. CallerMemberName es útil para implementar INotifyPropertyChanged. A continuación se describen algunos ejemplos.

Sub Log(msg As String,  
 <CallerFilePath> Optional file As String = Nothing,  
 <CallerLineNumber> Optional line As Integer? = Nothing)  
 Console.WriteLine("{0}:{1} - {2}", file, line, msg)  
End Sub

WriteOnly Property p As Integer  
 Set(value As Integer)  
 Notify(\_p, value)  
 End Set  
End Property

Private \_p As Integer

Sub Notify(Of T As IEquatable(Of T))(ByRef v1 As T, v2 As T,  
 <CallerMemberName> Optional prop As String = Nothing)  
 If v1 IsNot Nothing AndAlso v1.Equals(v2) Then Return  
 If v1 Is Nothing AndAlso v2 Is Nothing Then Return  
 v1 = v2  
 RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(prop))  
End Sub

Anotación

Además de los parámetros opcionales anteriores, Microsoft Visual Basic también reconoce algunos parámetros opcionales más si se importan de los metadatos (es decir, de una referencia de DLL). Tras importar de los metadatos, Visual Basic también considera el parámetro <Optional> como indicativo de que el parámetro es opcional: de esta forma, es posible importar una declaración que tiene un parámetro opcional pero no un valor predeterminado, aunque esto no se puede expresar con la palabra clave Optional.

1. Si el parámetro opcional tiene el atributo Microsoft.VisualBasic.CompilerServices.OptionCompareAttribute y el literal numérico 1 o 0 tiene una conversión en el tipo de parámetro, el compilador usa como argumento el literal 1 si está en vigor Option Compare Text o el literal 0 si lo está Optional Compare Binary.
2. Si el parámetro opcional tiene el atributo System.Runtime.CompilerServices.IDispatchConstantAttribute, es de tipo Object y no especifica un valor predeterminado, el compilador usa el argumento New System.Runtime.InteropServices.DispatchWrapper(Nothing).
3. Si el parámetro opcional tiene el atributo System.Runtime.CompilerServices.IUnknownConstantAttribute, es de tipo Object y no especifica un valor predeterminado, el compilador usa el argumento New System.Runtime.InteropServices.UnknownWrapper(Nothing).
4. Si el parámetro opcional es de tipo Object y no especifica un valor predeterminado, el compilador usa el argumento System.Reflection.Missing.Value.

### Métodos condicionales

Si el método de destino al que hace referencia la expresión de invocación es una subrutina que no es miembro de una interfaz y si el método tiene uno o más atributos System.Diagnostics.ConditionalAttribute, la evaluación de la expresión depende de las constantes de compilación condicional definidas en ese punto en el archivo de origen. Cada instancia del atributo especifica una cadena, que denomina una constante de compilación condicional. Cada constante de compilación condicional se evalúa como si formara parte de una instrucción de compilación condicional. Si la constante se evalúa como True, la expresión se evalúa normalmente en tiempo de ejecución. Si la constante se evalúa como False, la expresión no se evalúa en absoluto.

Cuando se busca el atributo, se comprueba la declaración más derivada de un método invalidable.

Nota   El atributo no es válido en métodos de interfaz o de función, y se pasa por alto si se especifica en estos métodos. Por consiguiente, los métodos condicionales solo aparecerán en instrucciones de invocación.

### Inferencia de argumentos de tipo

Cuando se llama a un método con parámetros de tipo sin especificar los argumentos de tipo, se usa la inferencia de argumentos de tipo para intentar inferir los argumentos de tipo para la llamada. Esto permite el uso de una sintaxis más natural para llamar a un método con parámetros de tipo cuando estos se pueden inferir trivialmente. Por ejemplo, dada la declaración de método:

Module Util  
 Function Choose(Of T)(b As Boolean, first As T, second As T) As T  
 If b Then  
 Return first  
 Else  
 Return second  
 End If  
 End Function  
End Class

es posible invocar el método Choose sin especificar explícitamente un argumento de tipo:

' calls Choose(Of Integer)  
Dim i As Integer = Util.Choose(True, 5, 213)  
' calls Choose(Of String)  
Dim s As String = Util.Choose(False, "a", "b")

Mediante la inferencia, los argumentos de tipo Integer y String se determinan desde los argumentos al método.

La inferencia de los argumentos de tipo tiene lugar antes de efectuar la reclasificación de las expresiones en los métodos o punteros a métodos de la lista de argumentos, porque la reclasificación de esas dos formas de expresión puede requerir que se conozca el tipo del parámetro. Dado un grupo de argumentos A1, A2, …, AN, un grupo de parámetros coincidentes P1, P2, …, PN y un grupo de parámetros de tipo de método T1, T2, …, TN, las dependencias entre los argumentos y parámetros de tipo de método se recopilan primero del modo siguiente:

Si AN es el literal Nothing, no se generan dependencias.

Si AN es un método lambda y el tipo de PN es un tipo de delegado construido o System.Linq.Expressions.Expression(Of T), donde T es un tipo de delegado construido,

Si el tipo de un parámetro de método lambda se inferirá a partir del tipo de parámetro PN correspondiente y el tipo del parámetro depende de un parámetro de tipo de método TN, entonces AN tiene una dependencia en TN.

Si el tipo de un parámetro de método lambda se especifica y el tipo de parámetro PN correspondiente depende de un parámetro de tipo de método TN, entonces TN tiene una dependencia en AN.

Si el tipo devuelto de PN depende de un parámetro de tipo de método TN, entonces TN tiene una dependencia en AN.

Si AN es un puntero a método y el tipo de PN es un tipo de delegado construido,

Si el tipo devuelto de PN depende de un parámetro de tipo de método TN, entonces TN tiene una dependencia en AN.

Si PN es un tipo construido y el tipo de PN depende de un parámetro de tipo de método TN, entonces TN tiene una dependencia en AN.

En otro caso, no se genera ninguna dependencia.

Después de recopilar las dependencias, se eliminan todos los argumentos que no tengan dependencias. Si algún parámetro de tipo de método no tiene dependencias salientes (es decir, no dependen de un argumento), la inferencia de tipos da error. De otro modo, los argumentos restantes y los parámetros de tipo de método se agrupan en componentes fuertemente conectados. Un componente fuertemente conectado es un conjunto de argumentos y parámetros de tipo de método donde cualquier elemento del componente se puede alcanzar a través de las dependencias en otros elementos.

Los componente fuertemente conectados se ordenan y se procesan topológicamente:

Si el componente fuertemente tipado contiene solo un elemento,

Si el componente ya se ha marcado completo, se pasa por alto.

Si el elemento es un argumento, agregue sugerencias de tipo del argumento a los parámetros de tipo de método que dependen de él y marque el elemento como completo. Si el argumento es un método lambda con parámetros que siguen necesitando tipos inferidos, infiera Object para los tipos de esos parámetros.

Si el elemento es un parámetro de tipo de método, infiera el parámetro de tipo de método para que sea el tipo dominante entre las sugerencias de tipo de argumento y marque el elemento como completo. Si una sugerencia de tipo tiene una restricción de elemento de matriz, solo se consideran las conversiones que son válidas entre las matrices del tipo determinado (es decir, las conversiones de matriz intrínseca y de covariante). Si una sugerencia de tipo tiene una restricción de argumento genérico, solo se consideran las conversiones de identidad. Si no se puede elegir un tipo dominante, la inferencia da error. Si los tipos de argumento del método lambda dependen de este parámetro de tipo de método, el tipo se propaga al método lambda.

Si el componente fuertemente tipado contiene más de un elemento, el componente contiene un ciclo.

Por cada parámetro de tipo de método que es un elemento del componente, si el parámetro de tipo de método depende de un argumento no marcado como completo, convierta esa dependencia en una aserción que se comprobará al final del proceso de inferencia.

Reinicie el proceso de inferencia en el punto en el que se determinaron los componentes fuertemente tipados.

Si la inferencia de tipos es correcta para todos los parámetros de tipo de método, se comprueban todas las dependencias que se convirtieron en aserciones. Una aserción es correcta si tipo del argumento es implícitamente convertible en el tipo inferido del parámetro de tipo de método. Si la aserción da error, la inferencia de argumentos de tipo da error.

Dado un tipo de argumento TA para un argumento A y un tipo de parámetro TP para un parámetro P, las sugerencias de tipo se generan del modo siguiente:

Si TP no implica ningún parámetro de tipo de método, no se genera ninguna sugerencia.

Si TP y TA son tipos de matriz del mismo rango, reemplace TA y TP por los tipos de elemento de TA y TP y reinicie este proceso con una restricción de elemento de matriz.

Si TP es un parámetro de tipo de método, TA se agrega como una sugerencia de tipo con la restricción actual, si la hubiera.

Si A es un método lambda y TP es un tipo de delegado construido o System.Linq.Expressions.Expression(Of T), donde T es un tipo de delegado construido, por cada tipo de parámetro de método lambda TL y el tipo de parámetro de delegado correspondiente TD, reemplace TA por TL y TP por TD y reinicie el proceso sin ninguna restricción. Después reemplace TA por el tipo de valor devuelto del método lambda y:

si A es un método lambda regular, reemplace TP por el tipo devuelto del tipo de delegado;

si A es un método lambda asincrónico y el tipo devuelto del tipo de delegado tiene la forma Task(Of T) para algunos elementos T, reemplace TP por el elemento T;

si A es un método lambda iterator y el tipo devuelto del tipo de delegado tiene la forma IEnumerator(Of T) o IEnumerable(Of T) para algunos elementos T, reemplace TP por el elemento T.

A continuación, reinicie el proceso sin ninguna restricción.

Si A es un puntero a método y TP es un tipo de delegado construido, uso los tipos de parámetro de TP para determinar qué método es más aplicable a TP. Si hay un método más aplicable, reemplace TA por el tipo devuelto del método y TP por el tipo devuelto del tipo de delegado y reinicie el proceso sin ninguna restricción.

De lo contrario, TP debe ser un tipo construido. Dado TG, el tipo genérico de TP,

Si TA es TG, hereda de TG o implementa el tipo TG exactamente una vez, por cada argumento de tipo coincidente TAX de TA y TPX de TP, reemplace TA por TAX y TP por TPX y reinicie el proceso con una restricción de argumento genérico.

En caso contrario, la inferencia de tipos da error para el método genérico.

La inferencia de tipos correcta no garantiza por si sola que el método sea aplicable.

## Expresiones de índice

Una expresión de índice da como resultado un elemento de matriz o reclasifica un grupo de propiedades como un acceso de propiedad. Una expresión de índice consiste, en este orden, en una expresión, un paréntesis de apertura, una lista de argumentos de índice y un paréntesis de cierre. El destino de la expresión de índice debe clasificarse como un grupo de propiedades o un valor. Una expresión de índice se procesa como se explica a continuación:

Si la expresión de destino se clasifica como un valor y su tipo no es un tipo de matriz, Object o System.Array, el tipo debe tener una propiedad predeterminada. El índice se efectúa en un grupo de propiedades que representa todas las propiedades predeterminadas del tipo. Aunque no es válido declarar una propiedad predeterminada sin parámetros en Visual Basic, tal vez otros lenguajes sí lo permitan. En consecuencia, se permite indizar una propiedad sin argumentos.

Si la expresión da como resultado un valor de un tipo de matriz, el número de argumentos de la lista debe ser el mismo que el rango del tipo de matriz y no puede incluir argumentos con nombre. Si alguno de los índices no es válido en tiempo de ejecución, se produce una excepción System.IndexOutOfRangeException. Cada expresión debe poder convertirse implícitamente en el tipo Integer. El resultado de una expresión de índice es la variable del índice especificado y se clasifica como variable.

Si la expresión se clasifica como un grupo de propiedades, se usa la resolución de sobrecarga para determinar si una de las propiedades es aplicable a la lista de argumentos de índice. Si el grupo de propiedades solo contiene una propiedad con el descriptor de acceso Get y si el descriptor de acceso no toma argumentos, el grupo de propiedades se interpreta como una expresión de índice con una lista de argumentos vacía. El resultado se usa como el destino de la expresión actual. Si no hay ninguna propiedad aplicable, se produce un error durante la compilación. De lo contrario, la expresión da como resultado un acceso a propiedad con la expresión de destino asociada (si hay alguna) del grupo de propiedades.

Si la expresión se clasifica como un grupo de propiedades enlazadas en tiempo de ejecución o un valor cuyo tipo es Object o System.Array, el procesamiento de la expresión de índice se puede diferir hasta el tiempo de ejecución y la indización se enlaza en tiempo de ejecución. La expresión da como resultado un acceso de propiedad enlazada en tiempo de ejecución de tipo Object. La expresión de destino asociada es la expresión de destino, si es un valor, o la expresión de destino asociada del grupo de propiedades. En tiempo de ejecución la expresión se procesa como se explica a continuación:

Si la expresión se clasifica como grupo de propiedades enlazadas en tiempo de ejecución, puede dar como resultado un grupo de métodos, un grupo de propiedades o un valor (si el miembro es una variable de compartida o de instancia). Si el resultado es un grupo de métodos o de propiedades, se aplica la resolución de sobrecarga al grupo para determinar el método correcto para la lista de argumentos. Si la resolución de sobrecarga da error, se produce una excepción System.Reflection.AmbiguousMatchException. Después se procesa el resultado bien como un acceso de propiedad o una invocación y se devuelve el resultado. Si la invocación es a una subrutina, el resultado es Nothing.

Si el tipo en tiempo de ejecución de la expresión de destino es un tipo de valor o System.Array, el resultado de la expresión de índice es el valor de la variable en el índice especificado.

De lo contrario, el tipo en tiempo de ejecución de la expresión debe tener una propiedad predeterminada y el índice se efectúa en el grupo de propiedades que representa todas las propiedades predeterminadas en el tipo. Si el tipo no tiene una propiedad predeterminada, se produce una excepción System.MissingMemberException.

IndexExpression ::= Expression OpenParenthesis [ ArgumentList ] CloseParenthesis

## Expresiones New

El operador New se utiliza para crear nuevas instancias de tipos. Existen cuatro formas de expresiones New:

Las expresiones de creación de objetos se utilizan para crear nuevas instancias de tipos de clase y tipos de valor.

Las expresiones de creación de matrices se utilizan para crear nuevas instancias de tipos de matriz.

Las expresiones de creación de delegados (que no se distinguen de las expresiones de creación de objetos) se usan para crear nuevas instancias de tipos de delegado.

Las expresiones de creación de objetos anónimos se utilizan para crear nuevas instancias de tipos de clases anónimas.

Una expresión New se clasifica como valor y el resultado es la nueva instancia del tipo.

NewExpression ::=  
 ObjectCreationExpression |  
 ArrayExpression |  
 AnonymousObjectCreationExpression

### Expresiones de creación de objetos

Una expresión de creación de objetos se utiliza para crear una nueva instancia de un tipo de clase o un tipo de estructura. El tipo de una expresión de creación de objetos debe ser un tipo de clase, un tipo de estructura o un tipo de parámetros con una restricción New y no puede ser una clase MustInherit. Dada una expresión de creación de objetos de la forma New T(A), donde T es un tipo de clase o un tipo de estructura y A es una lista de argumentos opcional, la resolución de sobrecarga determina el constructor correcto de T al que llamar. Cualquier parámetro de tipo con una restricción New se considera que tiene un constructor sin parámetros único. Si no hay ningún constructor invocable, se produce un error en tiempo de compilación; de lo contrario, la expresión da como resultado la creación de una nueva instancia de T usando el constructor elegido. Si no hay argumentos, pueden omitirse los paréntesis.

Dónde se asigne una instancia depende de si la instancia es un tipo de clase o un tipo de valor. Las instancias New de tipos de clase se crean en el montón del sistema, mientras que las de los tipos de valor se crean directamente en la pila.

Una expresión de creación de objetos puede especificar opcionalmente una lista de inicializadores de miembros después de los argumentos del constructor. Estos inicializadores de miembros llevan como prefijo la palabra clave With y la lista de inicializadores se interpreta como si estuviera en el contexto de una instrucción With. Por ejemplo, dada la clase:

Class Customer  
 Dim Name As String  
 Dim Address As String  
End Class

El código:

Module Test  
 Sub Main()  
 Dim x As New Customer() With { .Name = "Bob Smith", \_  
 .Address = "123 Main St." }  
 End Sub  
End Module

es más o menos equivalente a:

Module Test  
 Sub Main()  
 Dim x, \_t1 As Customer  
  
 \_t1 = New Customer()  
 With \_t1  
 .Name = "Bob Smith"  
 .Address = "123 Main St."  
 End With  
  
 x = \_t1  
 End Sub  
End Module

Cada inicializador debe especificar un nombre para asignar, que debe ser una variable de instancia o propiedad que no sea ReadOnly del tipo que se está construyendo; el acceso a miembro no se enlazará en tiempo de ejecución si el tipo que se está construyendo es Object. Los inicializadores no pueden usar la palabra clave Key. Cada miembro de un tipo solo se puede inicializar una vez. Las expresiones del inicializador, sin embargo, se pueden hacer referencia mutuamente. Por ejemplo:

Module Test  
 Sub Main()  
 Dim x As New Customer() With { .Name = "Bob Smith", \_  
 .Address = .Name & " St." }  
 End Sub  
End Module

Los inicializadores se asignan de izquierda a derecha, así que si un inicializador hace referencia a un miembro que todavía no se ha inicializado, verá el valor que ejecutó la variable de instancia después del constructor:

Module Test  
 Sub Main()  
 ' The value of Address will be " St." since Name has not been  
 ' assigned yet.  
 Dim x As New Customer() With { .Address = .Name & " St." }  
 End Sub  
End Module

Los inicializadores pueden estar anidados:

Class Customer  
 Dim Name As String  
 Dim Address As Address  
 Dim Age As Integer  
End Class  
  
Class Address  
 Dim Street As String  
 Dim City As String  
 Dim State As String  
 Dim ZIP As String  
End Class  
  
Module Test  
 Sub Main()  
 Dim c As New Customer() With { \_  
 .Name = "John Smith", \_  
 .Address = New Address() With { \_  
 .Street = "23 Main St.", \_  
 .City = "Peoria", \_  
 .State = "IL", \_  
 .ZIP = "13934" }, \_  
 .Age = 34 }  
 End Sub  
End Module

Si el tipo que se está creando es un tipo de colección y tiene una instancia de método denominada Add (incluidos los métodos de extensión y los métodos compartidos), entonces la expresión de creación de objetos puede especificar un inicializador de colección con la palabra clave From como prefijo. Una expresión de creación de objetos no puede especificar tanto un inicializador de miembros como un inicializador de colecciones. Cada elemento del inicializador de colecciones se pasa como argumento a una invocación de la función Add. Por ejemplo:

Dim list = New List(Of Integer)() From { 1, 2, 3, 4 }

equivale a:

Dim list = New List(Of Integer)()  
list.Add(1)  
list.Add(2)  
list.Add(3)

Si un elemento es un inicializador de colecciones, cada elemento del inicializador de la subcolección se pasará como argumento individual a la función Add. Por ejemplo, el código siguiente:

Dim dict = Dictionary(Of Integer, String) From { { 1, "One" },{ 2, "Two" } }

equivale a:

Dim dict = New Dictionary(Of Integer, String)  
dict.Add(1, "One")  
dict.Add(2, "Two")

Esta expansión se hace siempre y solo en un nivel de profundidad; después de eso, los sub-inicializadores se consideran literales de matriz. Por ejemplo:

' Error: List(Of T) does not have an Add method that takes two parameters.  
Dim list = New List(Of Integer())() From { { 1, 2 }, { 3, 4 } }  
  
' OK, this initializes the dictionary with (Integer, Integer()) pairs.  
Dim dict = New Dictionary(Of Integer, Integer())() From \_  
 { { 1, { 2, 3 } }, { 3, { 4, 5 } } }

ObjectCreationExpression ::=  
 New NonArrayTypeName [ OpenParenthesis [ ArgumentList ] CloseParenthesis ]   
 [ ObjectCreationExpressionInitializer ]

ObjectCreationExpressionInitializer ::= ObjectMemberInitializer | ObjectCollectionInitializer

ObjectMemberInitializer ::=  
 With OpenCurlyBrace FieldInitializerList CloseCurlyBrace

FieldInitializerList ::=  
 FieldInitializer |  
 FieldInitializerList Comma FieldInitializer

FieldInitializer ::= [ [ Key ] . IdentifierOrKeyword Equals ] Expression

ObjectCollectionInitializer ::= From CollectionInitializer

CollectionInitializer ::= OpenCurlyBrace [ CollectionElementList ] CloseCurlyBrace

CollectionElementList ::=  
 CollectionElement |  
 CollectionElementList Comma CollectionElement

CollectionElement ::=  
 *Expression* |  
 *CollectionInitializer*

### Expresiones de matriz

Para crear una nueva instancia de un tipo matricial, se utiliza una expresión de matriz. Hay dos tipos de expresiones de matriz: expresiones de creación de matrices y literales de matrices.

#### Expresiones de creación de matrices

Si se proporciona el modificador de inicialización de tamaño de matriz, el tipo de matriz resultante se deriva eliminando cada argumento individual de la lista de argumentos de inicialización de tamaño de matrices. El valor de cada argumento determina el límite superior de la dimensión correspondiente en la instancia de matriz recién asignada. Si la expresión tiene un inicializador de colección no vacía, cada argumento de la lista de argumentos debe ser una constante, y las longitudes de rango y dimensión especificadas en la lista de expresiones deben coincidir con las de la colección.

Dim a() As Integer = New Integer(2) {}  
Dim b() As Integer = New Integer(2) { 1, 2, 3 }  
Dim c(,) As Integer = New Integer(1, 2) { { 1, 2, 3 } , { 4, 5, 6 } }  
  
' Error, length/initializer mismatch.  
Dim d() As Integer = New Integer(2) { 0, 1, 2, 3 }

Si no se especifica ningún modificador de inicialización de tamaño de matriz, el nombre de tipo debe ser un tipo de matriz y el inicializador de colección debe estar vacío o tener el mismo número de niveles de anidación que el rango del tipo de matriz especificado. Todos los elementos del nivel de anidación más interno deben poder convertirse implícitamente en el tipo de elemento de la matriz y deben clasificarse como valores. El número de elementos de cada inicializador de colección anidado siempre debe ser coherente con el tamaño de otras colecciones situadas en el mismo nivel. Las longitudes de las dimensiones individuales se infieren del número de elementos en cada uno de los niveles de anidamiento correspondientes del inicializador de colección. Si el inicializador de colección está vacío, la longitud de cada dimensión es cero.

Dim e() As Integer = New Integer() { 1, 2, 3 }  
Dim f(,) As Integer = New Integer(,) { { 1, 2, 3 } , { 4, 5, 6 } }  
  
' Error: Inconsistent numbers of elements!  
Dim g(,) As Integer = New Integer(,) { { 1, 2 }, { 4, 5, 6 } }  
  
' Error: Inconsistent levels of nesting!  
Dim h(,) As Integer = New Integer(,) { 1, 2, { 3, 4 } }

El nivel de anidamiento superior de un inicializador de colección se corresponde con la dimensión situada en el extremo izquierdo de una matriz y el nivel de anidamiento inferior se corresponde con la dimensión situada en el extremo derecho. El ejemplo:

Dim array As Integer(,) = \_  
 { { 0, 1 }, { 2, 3 }, { 4, 5 }, { 6, 7 }, { 8, 9 } }

equivale a lo siguiente:

Dim array(4, 1) As Integer  
  
array(0, 0) = 0: array(0, 1) = 1  
array(1, 0) = 2: array(1, 1) = 3  
array(2, 0) = 4: array(2, 1) = 5  
array(3, 0) = 6: array(3, 1) = 7  
array(4, 0) = 8: array(4, 1) = 9

Si el inicializador de colección está vacío (es decir, contiene llaves pero ninguna lista) y los límites de las dimensiones de la matriz que se está inicializando son conocidos, el inicializador de colección vacía representa una instancia de matriz del tamaño especificado donde todos los elementos se han inicializado en el valor predeterminado del tipo de elemento. Si los límites de las dimensiones de la matriz que se está inicializando no se conocen, el inicializador de colección vacía representa una instancia de matriz en la que todas las dimensiones tienen tamaño cero.

La longitud y el rango de la instancia de matriz de cada dimensión son constantes para el período de 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.

#### Literales de matriz

Un literal de matriz denota una matriz cuyo tipo de elemento, rango y límites se infieren a partir de una combinación del contexto de la expresión y un inicializador de colección. Esto se explica en la Sección 11.1.1, Reclasificación de expresiones. Por ejemplo:

' array of integers  
Dim a = {1, 2, 3}

' array of shorts  
Dim b = {1S, 2S, 3S}

' array of shorts whose type is taken from the context  
Dim c As Short() = {1, 2, 3}

' array of type Integer(,)  
Dim d = {{1, 0}, {0, 1}}

' jagged array of rank ()()  
Dim e = {({1, 0}), ({0, 1})}

' error: inconsistent rank  
Dim f = {{1}, {2, 3}}

' error: inconsistent rank  
Dim g = {1, {2}}

El formato y los requisitos del inicializador de colección en un literal de matriz son exactamente iguales que los del inicializador de colección de una expresión de creación de matrices.

Anotación

Un literal de matriz no crea la matriz en y de sí misma; más bien es la reclasificación de la expresión en un valor lo que hace que se cree la matriz. Por ejemplo, la conversión CType(new Integer() {1,2,3}, Short()) no es posible porque no hay conversión de Integer() en Short(); pero la expresión CType({1,2,3},Short()) es posible porque primero reclasifica el literal de matriz en la expresión de creación de matriz New Short() {1,2,3}.

ArrayExpression ::= *ArrayCreationExpression* | *ArrayLiteralExpression*

*ArrayCreationExpression*  ::=  
 New NonArrayTypeName ArrayNameModifier CollectionInitializer

ArrayLiteralExpression ::=  
 CollectionInitializer

### Expresiones de creación de delegados

Una expresión de creación de delegados se usa para crear una nueva instancia de un tipo de delegado. El argumento de una expresión de creación de delegados debe ser una expresión clasificada como puntero a método o como método lambda.

Si el argumento es un puntero a método, uno de los métodos a los que hace referencia el puntero debe ser aplicable a la firma del tipo de delegado. Un método M es aplicable a un tipo de delegado D si:

M no es Partial o tiene cuerpo.

Tanto M como D son funciones o D es una subrutina.

M y D tienen el mismo número de parámetros.

Los tipos de parámetro de M tienen una conversión del tipo de parámetro correspondiente de D y sus modificadores (es decir, ByRef, ByVal) coinciden.

El tipo devuelto de M, si hay alguno, tiene una conversión en el tipo devuelto de D.

Si el puntero a método hace referencia a un acceso enlazado en tiempo de ejecución, se supone que este acceso es a una función que tiene el mismo número de parámetros que el tipo de delegado.

Si no se utiliza semántica estricta y solo hay un método al que hace referencia el puntero a método, pero no es aplicable debido a que no tiene parámetros y el tipo de delegado sí, se considera que el método es aplicable y los parámetros o el tipo devuelto sencillamente se pasan por alto. Por ejemplo:

Delegate Sub F(x As Integer)  
  
Module Test  
 Sub M()  
 End Sub  
  
 Sub Main()  
 ' Valid  
 Dim x As F = AddressOf M  
 End Sub  
End Module

Anotación

Esta relajación solo se permite cuando no se emplea semántica estricta debido a los métodos de extensión. Como los métodos de extensión solo se tienen en cuenta cuando un método normal no es aplicable, es posible que un método de instancia sin parámetros oculte un método de extensión con parámetros con el fin de construir el delegado.

Si más de uno de los métodos a los que hace referencia el puntero a método es aplicable al tipo de delegado, se usa la resolución de sobrecarga para elegir entre los métodos candidatos. Los tipos de los parámetros al delegado se usan como los tipos de los argumentos para fines de resolución de sobrecarga. Si ningún método candidato es más aplicable, se produce un error en tiempo de compilación. En el ejemplo siguiente, la variable local se inicializa con un delegado que hace referencia al segundo método Square debido a que ese método es más aplicable a la signatura y tipo de valor devuelto de DoubleFunc.

Delegate Function DoubleFunc(x As Double) As Double  
  
Module Test  
 Function Square(x As Single) As Single  
 Return x \* x  
 End Function   
  
 Function Square(x As Double) As Double  
 Return x \* x  
 End Function  
  
 Sub Main()  
 Dim a As New DoubleFunc(AddressOf Square)  
 End Sub  
End Module

Si no hubiera existido el segundo método Square, se habría elegido el primero. Si el entorno de compilación u Option Strict especifica semántica estricta, se produce un error en tiempo de compilación si el método más específico al que hace referencia el puntero a método es más restringido que la signatura de delegado. Un método M se considera más restringido que un tipo de delegado D si:

Un parámetro de tipo de M tiene una conversión de ampliación (widening) en el parámetro de tipo de D.

O bien, el tipo devuelto de M, si hay alguno, tiene una conversión de restricción (narrowing) en el tipo devuelto de D.

Si los argumentos de tipo están asociados con el puntero a método, solo los métodos con el mismo número de argumentos de tipo se tienen en cuenta. Si no hay argumentos de tipo asociados al puntero a método, se emplea la inferencia de tipos cuando se busca la coincidencia de firmas con un método genérico. A diferencia de otra inferencia de tipos normal, el tipo devuelto del delegado se usa para inferir argumentos de tipo, pero los tipos devueltos siguen sin tenerse en cuenta cuando se determina la sobrecarga menos genérica. En el siguiente ejemplo se muestran dos maneras de suministrar un argumento de tipo a una expresión de creación de delegados.

Delegate Function D(s As String, i As Integer) As Integer  
Delegate Function E() As Integer  
  
Module Test  
 Public Function F(Of T)(s As String, t1 As T) As T  
 End Function  
   
 Public Function G(Of T)() As T  
 End Function  
  
 Sub Main()  
 Dim d1 As D = AddressOf f(Of Integer) ' OK, type arg explicit  
 Dim d2 As D = AddressOf f ' OK, type arg inferred  
  
 Dim e1 As E = AddressOf g(Of Integer) ' OK, type arg explicit  
 Dim e2 As E = AddressOf g ' OK, infer from return  
 End Sub  
End Module

En el ejemplo anterior, se crean instancias de un tipo de delegado no genérico utilizando un método genérico. También es posible crear una instancia de un tipo de delegado construido utilizando un método genérico. Por ejemplo:

Delegate Function Predicate(Of U)(u1 As U, u2 As U) As Boolean  
  
Module Test  
 Function Compare(Of T)(t1 As List(of T), t2 As List(of T)) As Boolean  
 ...  
 End Function  
   
 Sub Main()  
 Dim p As Predicate(Of List(Of Integer))  
 p = AddressOf Compare(Of Integer)  
 End Sub  
End Module

Si el argumento para la expresión de creación de delegado es un método lambda, el método lambda debe ser aplicable a la firma del tipo de delegado. Un método lambda L es aplicable a un tipo de delegado D si:

Si L tiene parámetros, D tiene el mismo número de parámetros. (Si L no tiene parámetros, los parámetros de D se pasan por alto).

Los tipos de parámetro de L tienen una conversión en el tipo del tipo de parámetro correspondiente de D y sus modificadores (es decir, ByRef, ByVal) coinciden.

Si D es una función, el tipo devuelto de L tiene una conversión en el tipo devuelto de D. (Si D es a una subrutina, el valor devuelto de L se pasa por alto).

Si el tipo de parámetro de un parámetro de L se omite, el tipo del parámetro correspondiente en D se infiere; si el parámetro de L tiene modificadores de matriz o de nombre que admiten valores null, se produce un error en tiempo de compilación. Una vez que todos los tipos de parámetro de L están disponibles, se infiere el tipo de la expresión del método lambda. Por ejemplo:

Delegate Function F(x As Integer, y As Long) As Long  
  
Module Test  
 Sub Main()  
 ' b inferred to Integer, c and return type inferred to Long  
 Dim a As F = Function(b, c) b + c  
  
 ' e and return type inferred to Integer, f inferred to Long  
 Dim d As F = Function(e, f) e + CInt(f)  
 End Sub  
End Module

En las situaciones en las que la firma de delegado no coincide exactamente con el método lambda o la firma de método, .NET Framework no puede admitir la creación de delegados de forma nativa. En esa situación, una expresión de método lambda se usa para que emparejar ambos métodos. Por ejemplo:

Delegate Function IntFunc(x As Integer) As Integer  
  
Module Test  
 Function SquareString(x As String) As String  
 Return CInt(x) \* CInt(x)  
 End Function   
  
 Sub Main()  
 ' The following two lines are equivalent  
 Dim a As New IntFunc(AddressOf SquareString)  
 Dim b As New IntFunc( \_  
 Function(x As Integer) CInt(SquareString(CStr(x))))  
 End Sub  
End Module

El resultado de una expresión de creación de delegados es una instancia de delegado que hace referencia al método coincidente con la expresión de destino asociada (si la hay) de la expresión de puntero a método. Si el tipo de la expresión de destino es un tipo de valor, el tipo de valor se copia en el montón del sistema porque un delegado solo puede apuntar a un método de un objeto del montón. El método y el objeto al que hace referencia un delegado permanecen constantes durante la duración del delegado. Dicho de otra forma, no se pueden cambiar el destino ni el objeto de un delegado una vez haya sido creado.

### Expresiones de creación de objetos anónimos

Una expresión de creación de objetos con inicializadores de miembro puede omitir el nombre de tipo totalmente. En ese caso, se construye un tipo anónimo basado en los tipos y nombres de los miembros inicializados como una parte de la expresión. Por ejemplo:

Module Test  
 Sub Main()  
 Dim Customer = New With { .Name = "John Smith", .Age = 34 }  
  
 Console.WriteLine(Customer.Name)  
 End Sub  
End Module

El tipo creado por una expresión de creación de objetos anónimos es una clase que no tiene nombre, hereda directamente de Object y tiene un grupo de propiedades con el mismo nombre que los miembros asignados en la lista del inicializador de miembros. El tipo de cada propiedad se infiere utilizando las mismas reglas que en la inferencia de tipos de variable local. Los tipos anónimos generados también invalidan ToString y devuelven una representación de cadena de todos los miembros y sus valores. (El formato exacto de esta cadena escapa al ámbito de esta especificación).

De manera predeterminada, las propiedades generadas por el tipo anónimo son de lectura y escritura. Es posible marcar una propiedad de tipo anónimo como de solo lectura con el modificador Key. El modificador Key especifica que el campo se puede usar para identificar de forma única el valor que el tipo anónimo representa. Además de marcar la propiedad como de solo lectura, también hace que el tipo anónimo invalide Equals y GetHashCode y que implemente la interfaz System.IEquatable(Of T) (completando el tipo anónimo para T). Las miembros se definen de la manera siguiente:

Function Equals(obj As Object) As Boolean y Function Equals(val As T) As Boolean se implementan validando que las dos instancias son del mismo tipo y comparando después cada miembro Key usando Object.Equals. Si todos los miembros Key son iguales, Equals devuelve True; de lo contrario, devuelve False.

Function GetHashCode() As Integer se implementa de modo que si Equals es true para dos instancias del tipo anónimo, GetHashCode devolverá el mismo valor. El código hash comienza con un valor de inicialización y después, por cada miembro Key, en orden multiplica el código hash por 31 y agrega el valor hash del miembro Key (proporcionado por GetHashCode) si el miembro no es un tipo de referencia o un tipo de valor que admite valores null con el valor de Nothing.

Por ejemplo, el tipo creado en la instrucción:

Dim zipState = New With { Key .ZipCode = 98112, .State = "WA" }

crea una clase que tiene un aspecto similar a este (aunque la implementación exacta puede variar):

Friend NotInheritable Class $Anonymous1  
 Implements IEquatable(Of $Anonymous1)  
  
 Private ReadOnly \_zipCode As Integer  
 Private \_state As String  
  
 Public Sub New(zipCode As Integer, state As String)  
 \_zipCode = zipcode  
 \_state = state  
 End Sub  
  
 Public ReadOnly Property ZipCode As Integer  
 Get  
 Return \_zipCode  
 End Get  
 End Property  
  
 Public Property State As String  
 Get  
 Return \_state  
 End Get  
 Set (value As Integer)  
 \_state = value  
 End Set  
 End Property  
  
 Public Overrides Function Equals(obj As Object) As Boolean  
 Dim val As $Anonymous1 = TryCast(obj, $Anonymous1)  
 Return Equals(val)  
 End Function  
  
 Public Overloads Function Equals(val As $Anonymous1) As Boolean \_  
 Implements IEquatable(Of $Anonymous1).Equals  
  
 If val Is Nothing Then   
 Return False  
 End If  
  
 If Not Object.Equals(\_zipCode, val.\_zipCode) Then   
 Return False  
 End If  
  
 Return True  
 End Function  
  
 Public Overrides Function GetHashCode() As Integer  
 Dim hash As Integer = 0  
  
 hash = hash Xor \_zipCode.GetHashCode()  
  
 Return hash  
 End Function  
  
 Public Overrides Function ToString() As String  
 Return "{ Key .ZipCode = " & \_zipCode & ", .State = " & \_state & " }"  
 End Function  
End Class

Para simplificar la situación en la que un tipo anónimo se crea a partir de los campos de otro tipo, los nombres de campo se pueden inferir directamente de expresiones en los casos siguientes:

Una expresión de nombre simple x infiere el nombre x.

Una expresión de acceso a miembro x.y infiere el nombre y.

Una expresión de búsqueda de diccionario x!y infiere el nombre y.

Una expresión de índice o invocación sin argumentos x() infiere el nombre x.

Una expresión XML de acceso a miembro x.<y>, x...<y>, x.@y infiere el nombre y.

Una expresión XML de acceso a miembro que es el destino de una expresión de acceso a miembro x.<y>.z infiere el nombre z.

Una expresión XML de acceso a miembro que es el destino de una expresión de índice o invocación sin argumentos x.<y>.z() infiere el nombre z.

Una expresión XML de acceso a miembro que es el destino de una expresión de índice o invocación x.<y>(0) infiere el nombre y.

El inicializador se interpreta como una asignación de la expresión al nombre inferido. Por ejemplo, los siguientes inicializadores son equivalentes:

Class Address  
 Public Street As String  
 Public City As String  
 Public State As String  
 Public ZIP As String  
End Class  
  
Class C1  
 Sub Test(a As Address)  
 Dim cityState1 = New With { .City = a.City, .State = a.State }  
 Dim cityState2 = New With { a.City, a.State }  
 End Sub  
End Class

Si un nombre de miembro que se infiere entra en conflicto con algún miembro existente del tipo, como GetHashCode, se produce un error en tiempo de compilación. A diferencia de los inicializadores de miembros normales, las expresiones de creación de objetos anónimos no permiten que los inicializadores de miembros tengan referencias cruzadas ni hagan referencia a un miembro antes de que se haya inicializado. Por ejemplo:

Module Test  
 Sub Main()  
 ' Error: Circular references  
 Dim x = New With { .a = .b, .b = .a }  
  
 ' Error: Referring to .b before it has been assigned to  
 Dim y = New With { .a = .b, .b = 10 }  
  
 ' Error: Referring to .a before it has been assigned to  
 Dim z = New With { .a = .a }  
 End Sub  
End Module

Si dos expresiones de creación de objetos anónimos tienen lugar en el mismo método y resultan con la misma forma (el orden de las propiedades, los nombres y los tipos de propiedad coinciden) ambas harán referencia a la misma clase anónima. El ámbito de método de una variable de miembro compartido o de instancia con un inicializador es el constructor en el que se inicializa la variable.

Anotación

Es posible que un compilador elija unificar los tipos anónimos más, por ejemplo en el nivel de ensamblado, pero no se puede confiar en ello en este punto.

AnonymousObjectCreationExpression ::=  
 New ObjectMemberInitializer

## Expresiones de conversión

Una expresión de conversión fuerza un tipo determinado para una expresión. Las palabras clave de conversión específicas fuerzan las expresiones a los tipos primitivos. Tres palabras clave de conversión generales, CType, TryCast y DirectCast, fuerzan una expresión a un tipo.

DirectCast y TryCast tienen comportamientos especiales. Por esto, solo admiten conversiones nativas. Además, el tipo de destino de una expresión TryCast no puede ser un tipo de valor. Los operadores de conversión definidos por el usuario no se consideran cuando se usa DirectCast o TryCast.

Anotación

El grupo de conversiones que DirectCast y TryCast admiten son restringidas porque implementan conversiones “CLR nativo”. El propósito de DirectCast es ofrecer la funcionalidad de la instrucción “unbox”, mientras que el objetivo de TryCast es ofrecer la funcionalidad de la instrucción “isinst”. Dado que se asignan a instrucciones CLR, las conversiones compatibles no compatibles directamente con CLR anularían el fin perseguido.

DirectCast convierte las expresiones cuyo tipo es Object de distinto modo que CType. Cuando se convierte una expresión de tipo Object cuyo tipo en tiempo de ejecución es un tipo de valor primitivo, DirectCast produce una excepción System.InvalidCastException si el tipo especificado no es el mismo que el tipo en tiempo de ejecución de la expresión o una System.NullReferenceException si la expresión se evalúa como Nothing.

Anotación

Como se observa arriba, DirectCast se asigna directamente a la instrucción CLR “unbox” cuando el tipo de la expresión es Object. Por el contrario, CType se convierte en una llamada a un asistente en tiempo de ejecución para hacer la conversión de forma que se puedan admitir las conversiones entre tipos primitivos. En el caso en que una expresión Object se está convirtiendo en un tipo de valor primitivo y el tipo de la instancia actual coincide con el tipo de destino, DirectCast será significativamente más rápido que CType.

TryCast convierte expresiones pero no produce una excepción si la expresión no se puede convertir en el tipo de destino. En cambio, TryCast dará como resultado Nothing si la expresión no se puede convertir en tiempo de ejecución. Por ejemplo:

Interface ITest  
 Sub Test()  
End Interface  
  
Module Test  
 Sub Convert(o As Object)  
 Dim i As ITest = TryCast(o, ITest)  
  
 If i IsNot Nothing Then  
 i.Test()  
 End If  
 End Sub  
End Module

Anotación

Como se observa arriba, TryCast se asigna directamente a la instrucción CLR “isinst”. Combinando la comprobación de tipos y la conversión en una operación única, TryCast es más económico que hacer TypeOf…Is y después CType.

Si no existe una conversión del tipo de la expresión en el tipo especificado, se produce un error durante la compilación. De lo contrario, la expresión se clasifica como valor y el resultado es el valor producido por la conversión.

CastExpression ::=  
 DirectCast OpenParenthesis Expression Comma TypeName CloseParenthesis |  
 TryCast OpenParenthesis Expression Comma TypeName CloseParenthesis |  
 CType OpenParenthesis Expression Comma TypeName CloseParenthesis |  
 CastTarget OpenParenthesis Expression CloseParenthesis

CastTarget ::=  
 CBool | CByte | CChar | CDate | CDec | CDbl | CInt | CLng | CObj | CSByte | CShort |  
 CSng | CStr | CUInt | CULng | CUShort

## Expresiones de operador

Existen dos modalidades de operadores. Los operadores unarios tienen un operando y utilizan la notación de prefijos (por ejemplo, –x). Los operadores binarios tienen dos operandos y utilizan una notación de infijos (por ejemplo, x + y). A excepción de los operadores relacionales, que siempre dan un resultado Boolean, un operador definido para un tipo concreto da como resultado ese tipo. Los operandos a un operador deben clasificarse siempre como valor; el resultado de una expresión de operador se clasifica como valor.

OperatorExpression ::=  
 ArithmeticOperatorExpression |  
 RelationalOperatorExpression |  
 LikeOperatorExpression |  
 ConcatenationOperatorExpression |  
 ShortCircuitLogicalOperatorExpression |  
 LogicalOperatorExpression |  
 ShiftOperatorExpression |  
 AwaitOperatorExpression

### Prioridad y asociatividad de los operadores

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

|  |  |
| --- | --- |
| Categoría | Operadores |
| Principal | Todas las expresiones que no son de operador |
| Await | Await |
| Exponenciación | ^ |
| Negación unaria | +, - |
| Multiplicativo | \*, / |
| División de números enteros | \ |
| Módulo | Mod |
| Sumatorio | +, - |
| Concatenación | & |
| Desplazamiento | <<, >> |
| Relacionales | =, <>, <, >, <=, >=, Like, Is, IsNot |
| NOT lógico | Not |
| AND lógico | And, AndAlso |
| OR lógico | Or, OrElse |
| XOR lógico | Xor |

Cuando una expresión contiene dos operadores con igual grado de prioridad, la asociatividad de los operadores controla el orden en que se ejecutan las operaciones. Todos los operadores binarios son asociativos por la izquierda, lo que significa que las operaciones se realizan de izquierda a derecha. La precedencia y asociatividad pueden controlarse mediante expresiones entre paréntesis.

### Operandos de objeto

Además de los tipos normales que admite cada operador, todos los operadores admiten operandos de tipo Object. Los operadores aplicados a los operandos Object se controlan de forma similar a las llamadas a métodos hechas en valores Object: puede elegirse una llamada a métodos enlazados en tiempo de ejecución, en cuyo caso el tipo en tiempo en ejecución de los operandos, en lugar del tipo en tiempo de compilación, determina la validez y tipo de la operación. Si el entorno de compilación u Option Strict especifica semántica estricta, cualquier operador con operandos de tipo Object causará un error en tiempo de compilación, excepto los operadores TypeOf...Is, Is e IsNot.

Cuando la resolución de operador determina que una operación debe realizarse enlazada en tiempo de ejecución, el resultado de la operación es el resultado de aplicar el operador a los tipos de operandos si los tipos en tiempo de ejecución de los operandos son tipos que el operador admite. El valor Nothing se trata como el valor predeterminado del tipo del otro operador en una expresión de operador binario. En una expresión de operador unario, o si ambos operandos son Nothing en una expresión de operador binario, el tipo de la operación es Integer o el tipo de resultado único del operador, si el operador no da como resultado Integer. El resultado de la operación siempre se vuelve a convertir en Object. Si los tipos de operandos no tienen operador válido, se produce una excepción System.InvalidCastException. En tiempo de ejecución, las conversiones se hacen con independencia de si son explícitas o implícitas.

Si el resultado de una operación binaria numérica produciría una excepción de desbordamiento (con independencia de si la comprobación de desbordamiento de enteros está habilitada o deshabilitada), el tipo de resultado se promueve hasta el siguiente tipo numérico más amplio, si es posible. Por ejemplo, considere el siguiente código:

Module Test  
 Sub Main()  
 Dim o As Object = CObj(CByte(2)) \* CObj(CByte(255))  
  
 Console.WriteLine(o.GetType().ToString() & " = " & o)  
 End Sub  
End Module

Imprime el resultado siguiente:

System.Int16 = 512

Si no se dispone de tipo numérico más amplio, se produce una excepción System.OverflowException.

### Resolución de operadores

Dado un tipo de operador y un conjunto de operandos, la resolución de operador determina el operador que se usará para los operandos. Cuando se resuelven los operadores, se consideran primero los definidos por el usuario siguiendo estos pasos:

Primero se recopilan todos los operadores candidatos. Los operadores candidatos son todos los operadores definidos por el usuario del tipo de operador concreto en el tipo de origen y todos los operadores definidos por el usuario del tipo concreto en el tipo de destino. Si el tipo de origen y el de destino están relacionados, los operadores comunes solo se tienen en cuenta una vez.

Después, la resolución de sobrecarga se aplica a los operadores y operandos para seleccionar el operador más específico. En el caso de los operadores binarios, esto puede producir una llamada enlazada en tiempo de ejecución.

Cuando se recopilan los operadores candidatos para un tipo T?, se usan los operadores del tipo T en su lugar. Todos los operadores definidos por el usuario de T que implican tipos de valor que admiten valores null también se elevan. Un operador elevado usa la versión que admite valores null de los tipos de valor, a excepción de los tipos devueltos de IsTrue e IsFalse (que deben ser Boolean). Los operadores elevados se evalúan convirtiendo los operandos en su versión que no admiten valores null evaluado después el operador definido por el usuario y convirtiendo a continuación el tipo de resultado en su versión que admite valores null. Si alguno de los operandos es Nothing, el resultado de la expresión es un valor de Nothing con tipo como la versión que admite valores null del tipo de resultado. Por ejemplo:

Structure T  
 ...  
End Structure  
  
Structure S  
 Public Shared Operator +(ByVal op1 As S, ByVal op2 As T) As T  
 ...  
 End Operator  
End Structure  
  
Module Test  
 Sub Main()  
 Dim x As S?  
 Dim y, z As T?  
  
 ' Valid, as S + T = T is lifted to S? + T? = T?  
 z = x + y   
 End Sub  
End Module

Si el operador es binario y uno de los operandos es un tipo de referencia, el operador también se eleva pero cualquier enlace con el operador da error. Por ejemplo:

Structure S1  
 Public F1 As Integer  
  
 Public Shared Operator +(left As S1, right As String) As S1  
 ...  
 End Operator  
End Structure  
  
Module Test  
 Sub Main()  
 Dim a? As S1  
 Dim s As String  
  
 ' Error: '+' is not defined for S1? and String  
 a = a + s  
 End Sub  
End Module

Anotación

Esta regla existe porque se ha tenido en cuenta la posibilidad de agregar tipos de referencia de propagación de valores null en una futura versión, en cuyo caso el comportamiento en el caso de operadores binarios entre los dos tipos cambiaría.

Al igual que con las conversiones, siempre se prefieren los operadores definidos por el usuario a los operadores de elevación.

Cuando se resuelven operadores sobrecargados, puede haber diferencias entre las clases definidas en Visual Basic y las definidas en otros lenguajes:

En otros lenguajes, Not, And y Or pueden estar sobrecargados como operadores lógicos y operadores bit a bit. Al importar de un ensamblado externo, ambas formas se aceptan como una sobrecarga válida para estos operadores. Sin embargo, para un tipo que define operadores lógicos y bit a bit, solo se considerará la implementación bit a bit.

En otros lenguajes, >> y << pueden estar sobrecargados como operadores con signo y sin signo. Al importar de un ensamblado externo, ambas formas se aceptan como una sobrecarga válida. Sin embargo, para un tipo que define operadores con signo y sin signo, solo se considerará la implementación con signo.

Si ningún operador definido por el usuario es más específico para los operandos, se considerarán los operadores intrínsecos. Si no se define ningún operador intrínseco para los operandos y cualquiera de los operandos tiene el tipo Object, el operador se resolverá enlazado en tiempo de ejecución; de lo contrario, se produce un error en tiempo de compilación.

Anotación  
  
En versiones anteriores de Visual Basic, si había exactamente un operando de tipo Object, no existía ningún operador definido por el usuario aplicable y tampoco ningún operador intrínseco aplicable, era un error. A partir de Visual Basic 11, este problema se resuelve enlazado en tiempo de ejecución. Por ejemplo:

Module Module1  
 Sub Main()  
 Dim p As Object = Nothing  
 Dim U As New Uri("http://www.microsoft.com")  
 Dim j = U \* p ' is now resolved late-bound  
 End Sub  
End Module

Un tipo T que tiene un operador intrínseco también define ese mismo operador para T?. El resultado del operador en T? será el mismo que para T, excepto en que si alguno operando es Nothing, el resultado del operador será Nothing (es decir, el valor null se propaga). Con el fin de resolver el tipo de una operación, ? se quita de los operandos que lo tienen, se determina el tipo de la operación y se agrega ? al tipo de la operación si alguno de los operandos eran tipos de valor que admiten valores null. Por ejemplo:

Dim v1? As Integer = 10  
Dim v2 As Long = 20  
  
' Type of operation will be Long?  
Console.WriteLine(v1 + v2)

Cada operador enumera los tipos intrínsecos para los que está definido y el tipo de la operación efectuada dados los tipos de operandos. El resultado de tipo de una operación intrínseca sigue estas reglas generales:

Si todos los operandos son del mismo tipo y el operador está definido para el tipo, no se produce ninguna conversión y se usa el operador para ese tipo.

Cualquier operando cuyo tipo no esté definido para el operador se convierte según los pasos siguientes y el operador se resuelve con los nuevos tipos:

El operando se convierte en el siguiente tipo más amplio que esté definido para el operador y el operando, y en el que es convertible implícitamente.

Si no hay ningún tipo así, el operando se convierte en el siguiente tipo más restringido que esté definido para el operador y el operando y en el que es convertible implícitamente.

Si no hay un tipo así o no se puede realizar la conversión, se produce un error en tiempo de compilación.

De lo contrario, los operandos se convierten en el tipo más amplio de los tipos de operando y se usa el operador de ese tipo. Si el tipo de operando más restringido no se puede convertir implícitamente en el tipo de operador más amplio, se produce un error en tiempo de compilación.

A pesar de estas reglas generales, sin embargo, hay una serie de casos especiales descritos en las tablas de resultados de los operadores.

Nota Por razones de formato, en las tablas de resultados de los operadores se abrevian los nombres predefinidos a los dos primeros caracteres. Por tanto, “By” es Byte, “UI” es UInteger, “St” es String, etc. “Err” significa que no hay ninguna operación definida para los tipos de operandos dados.

## Operadores aritméticos

Los operadores \*, /, \, ^, Mod, + y – son los operadores aritméticos.

Las operaciones aritméticas 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 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, Visual Basic permite utilizar el 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.

ArithmeticOperatorExpression ::=  
 UnaryPlusExpression |  
 UnaryMinusExpression |  
 AdditionOperatorExpression |  
 SubtractionOperatorExpression |  
 MultiplicationOperatorExpression |  
 DivisionOperatorExpression |  
 ModuloOperatorExpression |  
 ExponentOperatorExpression

### Operador unario de signo más

El operador unario de signo menos se define para los tipos Byte, SByte, UShort, Short, UInteger Integer, ULong, Long, Single, Double y Decimal.

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Sh | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |

UnaryPlusExpression ::= + Expression

### Operador unario de signo menos

El operador unario de signo menos se define para los tipos siguientes:

SByte, Short, Integer y Long. El resultado se calcula restando el operando de cero. Si está habilitada la comprobación de desbordamiento de enteros y el valor del operando es el máximo negativo SByte, Short, Integer o Long, se produce una excepción System.OverflowException. De lo contrario, si el valor del operando es el máximo negativo SByte, Short, Integer o Long, el resultado es el mismo valor y no se indica el desbordamiento.

Single y Double. El resultado es el valor del operando con el signo invertido, incluidos los valores 0 e infinito. Si el operando es NaN, el resultado también es NaN.

Decimal. El resultado se calcula restando el operando de cero.

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Sh | SB | Sh | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |

UnaryMinusExpression ::= - Expression

### Operador de suma

El operador de suma calcula la suma de los dos operandos. Este operador se define para los tipos siguientes:

Byte, SByte, UShort, Short, UInteger, Integer, ULong y Long. Si está activada la comprobación de desbordamiento de enteros y la suma excede el intervalo del tipo de resultado, se produce una excepción System.OverflowException. De lo contario, no se informa de los desbordamientos y se descarta cualquier bit significativo de nivel superior del resultado.

Single y Double. La suma se calcula según las reglas de aritmética IEEE 754.

Decimal. 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.

String. Los dos operandos String se encadenan juntos.

Nota El tipo System.DateTime define los operadores de adición sobrecargados. Como System.DateTime es equivalente al tipo Date intrínseco, estos operadores también están disponibles en el tipo Date.

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | Sh | SB | Sh | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | UL | De | SI | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | De | SI | Do | Err | Err | Do | Ob |
| SI |  |  |  |  |  |  |  |  |  |  | SI | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | St | Err | St | Ob |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | St | St | Ob |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | St | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

AdditionOperatorExpression ::= Expression + [ LineTerminator ] Expression

### Operador de resta

El operador de sustracción resta el segundo operando del primero. Este operador se define para los tipos siguientes:

Byte, SByte, UShort, Short, UInteger, Integer, ULong y Long. Si está activada la comprobación de desbordamiento de enteros y la diferencia excede el intervalo del tipo de resultado, se produce una excepción System.OverflowException. De lo contario, no se informa de los desbordamientos y se descarta cualquier bit significativo de nivel superior del resultado.

Single y Double. La diferencia se calcula según las reglas de aritmética IEEE 754.

Decimal. 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.

Nota El tipo System.DateTime define los operadores de resta sobrecargados. Como System.DateTime es equivalente al tipo Date intrínseco, estos operadores también están disponibles en el tipo Date.

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | Sh | SB | Sh | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | UL | De | SI | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | De | SI | Do | Err | Err | Do | Ob |
| SI |  |  |  |  |  |  |  |  |  |  | SI | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Do | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

SubtractionOperatorExpression ::= Expression - [ LineTerminator ] Expression

### Operador de multiplicación

El operador de multiplicación calcula el producto de dos operandos. El operador de multiplicación se define para los tipos siguientes:

Byte, SByte, UShort, Short, UInteger, Integer, ULong y Long. Si está activada la comprobación de desbordamiento de enteros y el producto excede el intervalo del tipo de resultado, se produce una excepción System.OverflowException. De lo contario, no se informa de los desbordamientos y se descarta cualquier bit significativo de nivel superior del resultado.

Single y Double. El producto se calcula según las reglas de aritmética IEEE 754.

Decimal. 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.

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | Sh | SB | Sh | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | UL | De | SI | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | De | SI | Do | Err | Err | Do | Ob |
| SI |  |  |  |  |  |  |  |  |  |  | SI | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Do | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

MultiplicationOperatorExpression ::= Expression \* [ LineTerminator ] Expression

### Operadores de división

Los operadores de división calculan el cociente de dos operadores. Hay dos operadores de división: regular (punto flotante) y de enteros.

El operador de división regular se define para los tipos siguientes:

Single y Double. El cociente se calcula según las reglas de aritmética IEEE 754.

Decimal. 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, antes y después del redondeo, es la escala más cercana a la escala preferida que preservará un resultado igual al resultado exacto. La escala preferida es la escala del primer operando menos la escala del segundo.

De acuerdo con las reglas de resolución de operadores normales, la división regular solamente entre operandos de tipos como Byte, Short, Integer y Long haría que ambos operandos se convirtieran en el tipo Decimal. Sin embargo, al hacer la resolución de operador en el operador de división cuando ningún tipo es Decimal, Double se considera más restringido que Decimal. Se respeta esta convención porque la división Double es más eficaz que la división Decimal.

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | Do | Do | Do | Do | Do | Do | Do | Do | Do | De | SI | Do | Err | Err | Do | Ob |
| SB |  | Do | Do | Do | Do | Do | Do | Do | Do | De | SI | Do | Err | Err | Do | Ob |
| By |  |  | Do | Do | Do | Do | Do | Do | Do | De | SI | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Do | Do | Do | Do | Do | Do | De | SI | Do | Err | Err | Do | Ob |
| US |  |  |  |  | Do | Do | Do | Do | Do | De | SI | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | Do | Do | Do | Do | De | SI | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | Do | Do | Do | De | SI | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Do | Do | De | SI | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | Do | De | SI | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | De | SI | Do | Err | Err | Do | Ob |
| SI |  |  |  |  |  |  |  |  |  |  | SI | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Do | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

El operador de división de enteros se define para Byte, SByte, UShort, Short, UInteger, Integer, ULong y Long. Si el valor del operando derecho es cero, se produce una excepción System.DivideByZeroException. La división redondea el resultado hacia cero, y el valor absoluto del resultado es el entero mayor posible que sea menor 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 valor del operando izquierdo es el máximo negativo SByte, Short, Integer o Long y el operador derecho es –1, se produce una excepción; si la comprobación de desbordamiento de enteros está habilitada, se produce una excepción System.OverflowException. De lo contrario, no se informa del desbordamiento y el resultado es el valor del operando izquierdo.

Anotación

Como los dos operandos de los tipos sin signo siempre serán cero o positivo, el resultado siempre es cero o positivo. Como el resultado de la expresión siempre será menor o igual que el mayor de los dos operandos, no es posible que se produzca un desbordamiento. Como tal la comprobación de desbordamiento de enteros no se efectúa para dividir enteros con dos enteros sin signo. El resultado es el tipo que sea del operando izquierdo.

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | Sh | SB | Sh | Sh | In | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| In |  |  |  |  |  | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| Lo |  |  |  |  |  |  |  | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| UL |  |  |  |  |  |  |  |  | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| De |  |  |  |  |  |  |  |  |  | Lo | Lo | Lo | Err | Err | Lo | Ob |
| SI |  |  |  |  |  |  |  |  |  |  | Lo | Lo | Err | Err | Lo | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Lo | Err | Err | Lo | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Lo | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

DivisionOperatorExpression ::=  
 FPDivisionOperatorExpression |  
 IntegerDivisionOperatorExpression

FPDivisionOperatorExpression ::= Expression / [ LineTerminator ] Expression

IntegerDivisionOperatorExpression ::= Expression \ [ LineTerminator ] Expression

### Mod (Operador)

El operador Mod (módulo) calcula el resto de la división de dos operandos. El operador Mod se define para los tipos siguientes:

Byte, SByte, UShort, Short, UInteger, Integer, ULong y Long. El resultado de x Mod y es el valor producido por x – (x \ y) \* y. Si y es cero, se produce una excepción System.DivideByZeroException. El operador de módulo nunca causa un desbordamiento.

Single y Double. El resto se calcula según las reglas de aritmética IEEE 754.

Decimal. 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.

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | Sh | SB | Sh | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | UL | De | SI | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | De | SI | Do | Err | Err | Do | Ob |
| SI |  |  |  |  |  |  |  |  |  |  | SI | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Do | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

ModuloOperatorExpression ::= Expression Mod [ LineTerminator ] Expression

### Operador de exponenciación

El operador de exponenciación calcula el primer operando elevado a la potencia del segundo. El operador de exponenciación se define para el tipo Double. El valor se calcula según las reglas de aritmética IEEE 754.

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | Do | Do | Do | Do | Do | Do | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| SB |  | Do | Do | Do | Do | Do | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| By |  |  | Do | Do | Do | Do | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Do | Do | Do | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| US |  |  |  |  | Do | Do | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | Do | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | Do | Do | Do | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | Do | Do | Do | Err | Err | Do | Ob |
| SI |  |  |  |  |  |  |  |  |  |  | Do | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Do | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

ExponentOperatorExpression ::= Expression ^ [ LineTerminator ] Expression

## Operadores relacionales

Los operadores relacionales comparan dos valores entre sí. Los operadores de comparación son =, <>, <, >, <= y >=. Todos los operadores relacionales dan como resultado un valor Boolean.

Los operadores relacionales tienen el significado general siguiente:

El operador = comprueba si dos operandos son iguales.

El operador <> comprueba si dos operandos no son iguales.

El operador < comprueba si el primer operando es menor que el segundo.

El operador > comprueba si el primer operando es mayor que el segundo.

El operador <= comprueba si el primer operando es menor o igual que el segundo.

El operador >= comprueba si el primer operando es mayor o igual que el segundo.

Los operadores relaciones se definen para los tipos siguientes:

Boolean. Los operadores comparan los valores de veracidad de dos operandos. True se considera menos que False, que coincide con sus valores numéricos.

Byte, SByte, UShort, Short, UInteger, Integer, ULong y Long. Los operadores comparan los valores de veracidad de dos operandos enteros.

Single y Double. Los operadores comparan los operandos según las reglas del estándar IEEE 754.

Decimal. Los operadores comparan los valores de veracidad de dos operandos decimales.

Date. Los operadores devuelven el resultado de comparar los dos valores de fecha y hora.

Char. Los operadores devuelven el resultado de comparar los dos valores Unicode.

String. Los operadores devuelven el resultado de comparar los dos valores usando una comparación binaria o una comparación textual. La comparación usada vendrá por el entorno de compilación y la instrucción Option Compare. Una comparación binaria determina si el valor numérico Unicode de cada carácter de cada cadena es el mismo. Una comparación de texto lleva a cabo una comparación de texto Unicode basada en la referencia cultural que se usa en .NET. Cuando se lleva a cabo una comparación de cadenas, un valor null equivale al literal de cadena "".

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | Bo | SB | Sh | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Bo | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | In | Lo | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | De | SI | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Lo | De | De | SI | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | UL | De | SI | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | De | SI | Do | Err | Err | Do | Ob |
| SI |  |  |  |  |  |  |  |  |  |  | SI | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Da | Err | Da | Ob |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Ch | St | Ob |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | St | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

RelationalOperatorExpression ::=  
 Expression = [ LineTerminator ] Expression |  
 Expression < > [ LineTerminator ] Expression |  
 Expression < [ LineTerminator ] Expression |  
 Expression > [ LineTerminator ] Expression |  
 Expression < = [ LineTerminator ] Expression |  
 Expression > = [ LineTerminator ] Expression

## Like (Operador)

El operador Like se define para el tipo String y determina si una cadena coincide con un modelo determinado. El primer operando es la cadena y el segundo es el modelo. El modelo consiste en caracteres Unicode. Las siguientes secuencias de caracteres tienen significado especial:

El carácter ? coincide con cualquier carácter.

El carácter \* coincide con cero o más caracteres.

El carácter # coincide con cualquier dígito único (0-9).

Una lista de caracteres entre corchetes ([ab...]) coincide con cualquier carácter único de la lista.

Una lista de caracteres entre corchetes y precedidos por un signo de exclamación ([!ab...]) coincide con cualquier carácter único que no esté en la lista.

Dos caracteres en una lista de caracteres separados por un guion (-) especifican un intervalo de caracteres Unicode comenzando por el primero y terminando con el segundo. Si el segundo carácter no es posterior en el criterio de ordenación que el primer carácter, se produce una excepción en tiempo de ejecución. Un guión al principio o al final de una lista de caracteres se especifica a sí mismo.

Nota   Si desea buscar los caracteres especiales corchete de apertura ([), interrogación de cierre (?), signo de número (#) y asterisco (\*), escríbalos entre corchetes. No se puede usar el corchete de cierre (]) dentro de un grupo para identificar una autocoincidencia, pero sí fuera de un grupo, como carácter individual. La secuencia [] se considera un literal de cadena "".

Obsérvese también que la comparación de caracteres y la ordenación de la lista de caracteres son independientes del tipo de comparaciones que se esté usando. Si se usan comparaciones binarias, las comparaciones y ordenación de caracteres se basan en los valores Unicode numéricos. Si se usan comparaciones de texto, las comparaciones y ordenación de caracteres se basan en la referencia cultural empleada en .NET Framework.

En algunos idiomas, hay caracteres alfabéticos especiales que representan dos caracteres separados y viceversa. Por ejemplo, existen algunos idiomas que utilizan el carácter æ para representar los caracteres a y e cuando aparecen juntos, y los caracteres ˆ y O se pueden usar para representar el carácter Ô. Cuando se usan estas comparaciones de texto, el operador Like reconoce estas equivalencias culturales. En ese caso, una aparición de un único carácter especial, ya sea en modelo o en cadena, coincidirá con la secuencia de dos caracteres equivalente de la otra cadena. Del mismo modo, un carácter especial que aparezca en un modelo entre corchetes (aislado, en una lista o en un intervalo) coincidirá con la secuencia de dos caracteres equivalente de la cadena y viceversa.

En una expresión Like donde ambos operadores son Nothing o un operador tiene una conversión intrínseca en String y el otro operando es Nothing, Nothing se trata como si fuera un literal de cadena vacío "".

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | St | St | St | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| SB |  | St | St | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| By |  |  | St | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| Sh |  |  |  | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| US |  |  |  |  | St | St | St | St | St | St | St | St | St | St | St | Ob |
| In |  |  |  |  |  | St | St | St | St | St | St | St | St | St | St | Ob |
| UI |  |  |  |  |  |  | St | St | St | St | St | St | St | St | St | Ob |
| Lo |  |  |  |  |  |  |  | St | St | St | St | St | St | St | St | Ob |
| UL |  |  |  |  |  |  |  |  | St | St | St | St | St | St | St | Ob |
| De |  |  |  |  |  |  |  |  |  | St | St | St | St | St | St | Ob |
| SI |  |  |  |  |  |  |  |  |  |  | St | St | St | St | St | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | St | St | St | St | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | St | St | St | Ob |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | St | St | Ob |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | St | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

LikeOperatorExpression ::= Expression Like [ LineTerminator ] Expression

## Operador de concatenación

El operador de concatenación se define para todos los tipos intrínsecos, incluidas las versiones que admiten valores null de los tipos de valor intrínsecos. También se define para la concatenación entre los tipos mencionados anteriormente y System.DBNull, que se trata como una cadena Nothing. El operador de concatenación convierte todos sus operandos en String; en la expresión, todas las conversiones en String se consideran de ampliación (widening), con independencia de si se usa semántica estricta. Un valor System.DBNull se convierte en el literal Nothing con tipo String. Un tipo de valor que admite valores null cuyo valor es Nothing también se puede convertir en el literal Nothing con tipo String, en lugar de producir un error en tiempo de ejecución.

Una operación de concatenación da como resultado una cadena que es una concatenación de los dos operadores en orden de izquierda a derecha. Un valor Nothing se trata como si fuera el literal de cadena vacía "".

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | St | St | St | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| SB |  | St | St | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| By |  |  | St | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| Sh |  |  |  | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| US |  |  |  |  | St | St | St | St | St | St | St | St | St | St | St | Ob |
| In |  |  |  |  |  | St | St | St | St | St | St | St | St | St | St | Ob |
| UI |  |  |  |  |  |  | St | St | St | St | St | St | St | St | St | Ob |
| Lo |  |  |  |  |  |  |  | St | St | St | St | St | St | St | St | Ob |
| UL |  |  |  |  |  |  |  |  | St | St | St | St | St | St | St | Ob |
| De |  |  |  |  |  |  |  |  |  | St | St | St | St | St | St | Ob |
| SI |  |  |  |  |  |  |  |  |  |  | St | St | St | St | St | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | St | St | St | St | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | St | St | St | Ob |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | St | St | Ob |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | St | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

ConcatenationOperatorExpression ::= Expression & [ LineTerminator ] Expression

## Operadores lógicos

Los operadores And, Not, Or y Xor, que se denominan operadores lógicos, se evalúan del modo siguiente:

Para el tipo Boolean:

Una operación lógica And se efectúa en sus dos operandos.

Una operación lógica Not se efectúa en su operando.

Una operación lógica Or se efectúa en sus dos operandos.

Una operación exclusiva Or se efectúa en sus dos operandos.

Para Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long y todos los tipos enumerados, la operación especificada se efectúa en cada bit de la representación binaria de los dos operandos:

And: el bit de resultado es 1 si ambos bits son 1; de lo contrario, el bit de resultado es 0.

Not: el bit de resultado es 1 si el bit es 0; de lo contrario, el bit de resultado es 1.

Or: el bit de resultado es 1 si alguno es 1; de lo contrario, el bit de resultado es 0.

Xor: el bit de resultado es 1 si un bit es 1 pero no ambos; de lo contrario, el bit de resultado es 0 (esto es, 1 Xor 0 = 1, 1 Xor 1 = 0).

Cuando los operadores lógicos And y Or se elevan para el tipo Boolean?, se amplían para abarcar la lógica booleana de tres valores:

And se evalúa como true si ambos operandos son verdadero; false si alguno es falso; Nothing en otro caso.

Or se evalúa como true si algún operando es verdadero; false si ambos son falsos; Nothing en otro caso.

Por ejemplo:

Module Test  
 Sub Main()  
 Dim x?, y? As Boolean  
  
 x = Nothing  
 y = True   
  
 If x Or y Then  
 ' Will execute  
 End If  
 End Sub  
End Module

Anotación

Idealmente, los operadores lógicos And y Or se elevarían usando lógica de tres valores para cualquier tipo que se pueda usar en una expresión booleana (es decir, un tipo que implementa IsTrue e IsFalse), del mismo modo que AndAlso y OrElse cortocircuitan cualquier tipo que se pueda usar en una expresión booleana. Lamentablemente, la elevación de tres valores solo se aplica a Boolean?, de manera que los tipos definidos por el usuario que desean lógica de tres valores deben hacerlo definiendo manualmente los operadores And y Or para su versión que admite valores null.

Los desbordamientos no son posibles en estas operaciones. Los operadores de tipo enumerado hacen la operación bit a bit en el tipo subyacente del tipo enumerado, pero el valor devuelto es el tipo enumerado.

Tipo de operación Not:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | SB | By | Sh | US | In | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |

Tipo de operación And, Or, Xor:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | Bo | SB | Sh | Sh | In | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Bo | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| In |  |  |  |  |  | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| Lo |  |  |  |  |  |  |  | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| UL |  |  |  |  |  |  |  |  | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| De |  |  |  |  |  |  |  |  |  | Lo | Lo | Lo | Err | Err | Lo | Ob |
| SI |  |  |  |  |  |  |  |  |  |  | Lo | Lo | Err | Err | Lo | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Lo | Err | Err | Lo | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Lo | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

LogicalOperatorExpression ::=  
 Not Expression |  
 Expression And [ LineTerminator ] Expression |  
 Expression Or [ LineTerminator ] Expression |  
 Expression Xor [ LineTerminator ] Expression

### Cortocircuitar las operaciones lógicas

Los operadores AndAlso y OrElse son las versiones de cortocircuito de los operadores lógicos And y Or. Dado el comportamiento de cortocircuito, el segundo operando no se evalúa en tiempo de ejecución si el resultado del operador se conoce después de evaluar el primer operando.

Los operadores lógicos de “cortocircuito” se evalúan del modo siguiente:

Si el primer operando de la operación AndAlso se evalúa como False o devuelve True desde su operador IsFalse, la expresión devuelve su primer operando. De lo contrario, el segundo operador se evalúa y se efectúa una operación lógica And en los dos resultados.

Si el primer operando de la operación OrElse se evalúa como True o devuelve True desde su operador IsTrue, la expresión devuelve su primer operando. De lo contrario, el segundo operador se evalúa y se efectúa una operación lógica Or en sus dos resultados.

Los operadores AndAlso y OrElse se definen para el tipo Boolean o para cualquier tipo T que sobrecargue los operadores siguientes:

Public Shared Operator IsTrue(op As T) As Boolean  
Public Shared Operator IsFalse(op As T) As Boolean

además de sobrecargar el operador And u Or correspondiente:

Public Shared Operator And(op1 As T, op2 As T) As T  
Public Shared Operator Or(op1 As T, op2 As T) As T

Cuando se evalúan los operadores AndAlso u OrElse, el primer operando se evalúa una sola vez y el segundo no se evalúa o bien se evalúa exactamente una vez. Por ejemplo, considere el siguiente código:

Module Test  
 Function TrueValue() As Boolean  
 Console.Write(" True")  
 Return True  
 End Function  
  
 Function FalseValue() As Boolean  
 Console.Write(" False")  
 Return False  
 End Function  
  
 Sub Main()  
 Console.Write("And:")  
 If FalseValue() And TrueValue() Then  
 End If  
 Console.WriteLine()  
  
 Console.Write("Or:")  
 If TrueValue() Or FalseValue() Then  
 End If  
 Console.WriteLine()  
  
 Console.Write("AndAlso:")  
 If FalseValue() AndAlso TrueValue() Then  
 End If  
 Console.WriteLine()  
  
 Console.Write("OrElse:")  
 If TrueValue() OrElse FalseValue() Then  
 End If  
 Console.WriteLine()  
 End Sub  
End Module

Imprime el resultado siguiente:

And: False True  
Or: True False  
AndAlso: False  
OrElse: True

En la forma elevada de los operadores AndAlso y OrElse, si el primer operador es un valor null Boolean?, el segundo operando se evalúa pero el resultado es siempre un valor null Boolean?.

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| SB |  | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| By |  |  | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| Sh |  |  |  | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| US |  |  |  |  | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| In |  |  |  |  |  | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| UI |  |  |  |  |  |  | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| Lo |  |  |  |  |  |  |  | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| UL |  |  |  |  |  |  |  |  | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| De |  |  |  |  |  |  |  |  |  | Bo | Bo | Bo | Err | Err | Bo | Ob |
| SI |  |  |  |  |  |  |  |  |  |  | Bo | Bo | Err | Err | Bo | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Bo | Err | Err | Bo | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Bo | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

ShortCircuitLogicalOperatorExpression ::=  
 Expression AndAlso [ LineTerminator ] Expression |  
 Expression OrElse [ LineTerminator ] Expression

## Operadores de desplazamiento

Los operadores binarios << y >> realizan operaciones de desplazamiento de bits. Los operadores relaciones se definen para los tipos siguientes: Byte, SByte, UShort, Short, UInteger, Integer, ULong y Long. A diferencia de otros operadores binarios, el tipo de resultado de una operación de desplazamiento se determina como si el operador fuera unario y solo tuviera el operador izquierdo. El tipo del operando derecho debe poder convertirse implícitamente en Integer y no se usa para determinar el tipo de resultado de la operación.

El operador << desplaza los bits del primer operando a la izquierda el número de lugares especificado por la cantidad de desplazamiento. Los bits de orden superior no comprendidos en el intervalo del tipo de resultado se descartan y las posiciones de bits libres se rellenan con ceros.

El operador >> desplaza los bits del primer operando a la derecha el número de lugares especificado por la cantidad de desplazamiento. Los bits de orden inferior se descartan y las posiciones vacías de los bits de orden superior se establecen en cero si el operando izquierdo es positivo y en uno, si es negativo. Si el operando izquierdo es de tipo Byte, UShort, UInteger o ULong, los bits de orden superior se rellenan con ceros.

Los operadores de desplazamiento desplazan los bits de la representación subyacente del primer operando en la cantidad del segundo operando. Si el valor del segundo operando es mayor que el número de bits del primero o es negativo, la cantidad de desplazamiento se calcula como RightOperand And SizeMask donde SizeMask es:

|  |  |
| --- | --- |
| Tipo de operando izquierdo | SizeMask |
| Byte, SByte | 7 (&H7) |
| UShort, Short | 15 (&HF) |
| UInteger, Integer | 31 (&H1F) |
| ULong, Long | 63 (&H3F) |

Si la cantidad de desplazamiento es cero, el resultado de la operación es idéntico al valor del primer operando. Los desbordamientos no son posibles en estas operaciones.

Tipo de operación:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bo | SB | By | Sh | US | In | UI | Lo | UL | De | SI | Do | Da | Ch | St | Ob |
| Sh | SB | By | Sh | US | In | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |

ShiftOperatorExpression ::=  
 Expression < < [ LineTerminator ] Expression |  
 Expression > > [ LineTerminator ] Expression

## Expresiones booleanas

Una expresión booleana es la que se puede probar para ver si es verdadera o falsa. Un tipo T se puede usar en una expresión booleana si, por orden de preferencia:

T es Boolean o Boolean?

T tiene una conversión widening (de ampliación) en Boolean

T tiene una conversión widening (de ampliación) en Boolean?

T define dos pseudo-operadores, IsTrue e IsFalse.

T tiene una conversión narrowing (de restricción) en Boolean? que no implica un conversión de Boolean en Boolean?.

T tiene una conversión narrowing (de restricción) en Boolean.

Anotación

Es interesante observar que si Option Strict está desactivado, una expresión con una conversión narrowing en Boolean será aceptada sin errores en tiempo de compilación pero el lenguaje seguirá prefiriendo un operador IsTrue si existe. Esto se debe a que Option Strict solo cambia lo que es y no es aceptable por el lenguaje, y nunca cambia el significado de una expresión. Por tanto, IsTrue siempre se prefiere a una conversión narrowing, independientemente de Option Strict.

Por ejemplo, la clase siguiente no define una conversión widening en Boolean. Como resultado, su uso en la instrucción If provoca una llamada al operador IsTrue.

Class MyBool  
 Public Shared Widening Operator CType(b As Boolean) As MyBool  
 ...  
 End Operator  
  
 Public Shared Narrowing Operator CType(b As MyBool) As Boolean  
 ...  
 End Operator  
  
 Public Shared Operator IsTrue(b As MyBool) As Boolean  
 ...  
 End Operator  
  
 Public Shared Operator IsFalse(b As MyBool) As Boolean  
 ...  
 End Operator  
End Class  
  
Module Test  
 Sub Main()  
 Dim b As New MyBool  
  
 If b Then Console.WriteLine("True")  
 End Sub  
End Module

Si una expresión booleana tiene o se convierte en un tipo Boolean o Boolean?, es verdadera si el valor es True; de lo contrario es falsa.

De lo contrario, una expresión booleana llama al operador IsTrue y devuelve True si el operador devuelve True; si no, es falsa (pero nunca llama al operador IsFalse).

En el ejemplo siguiente, Integer tiene una conversión narrowing en Boolean, de manera que un valor null Integer? tiene una conversión narrowing en Boolean? (que da un valor null Boolean) y en Boolean (que produce una excepción). Se prefiere la conversión narrowing en Boolean? y, por tanto, el valor de “i” como expresión booleana es False.

Dim i As Integer? = Nothing  
If i Then Console.WriteLine()

BooleanExpression ::= Expression

## Expresiones lambda

Una expresión lambda define un método anónimo denominado método lambda. Los métodos lambda facilitan pasar métodos “en-línea” a otros métodos que toman tipos de delegado. El ejemplo:

Module Test  
 Delegate Function IntFunc(x As Integer) As Integer  
  
 Sub Apply(a() As Integer, func As IntFunc)  
 For index As Integer = 0 To a.Length - 1  
 a(index) = func(a(index))  
 Next index  
 End Sub  
  
 Sub Main()  
 Dim a() As Integer = { 1, 2, 3, 4 }  
  
 Apply(a, Function(x As Integer) x \* 2)  
  
 For Each value In a  
 Console.Write(value & " ")  
 Next value  
 End Sub  
End Module

imprimirá:

2 4 6 8

Una expresión lambda comienza con los modificadores opcionales Async o Iterator, seguidos de la palabra clave Function o Sub y una lista de parámetros. Los parámetros de una expresión lambda no se pueden declarar Optional o ParamArray y no pueden tener atributos. A diferencia de los métodos normales, la omisión de un tipo de parámetro para un método lambda no infiere automáticamente Object. En cambio, cuando se reclasifica un método lambda, los tipos de parámetros omitidos y los modificadores ByRef se infieren a partir del tipo de destino. En el ejemplo anterior, la expresión lambda se podía haber escrito como Function(x) x \* 2, y se habría inferido que el tipo de x es Integer cuando se usó el método lambda para crear una instancia del tipo de delegado IntFunc. A diferencia de la inferencia de variables locales, si un parámetro de un método lambda omite un tipo pero tiene una matriz o modificador de nombre que admite valores null, se produce un error en tiempo de ejecución.

Una **expresión lambda regular** es aquella que no tiene modificadores Async ni Iterator.

Una **expresión lambda de iterador** es aquella que tiene un modificador Iterator y ninguno Async. Debe ser una función. Cuando se reclasifica en un valor, solo puede reclasificarse en un valor de tipo delegado cuyo tipo devuelto es IEnumerator, IEnumerable, IEnumerator(Of T) o IEnumerable(Of T) para algunos T y que no tiene parámetros ByRef.

Una **expresión lambda asincrónica** es aquella que tiene el modificador Async y ninguno Iterator. Una expresión lambda sub asincrónica solo puede reclasificarse en un valor de tipo delegado sub sin parámetros ByRef. Una función lambda asincrónica solo puede reclasificarse en un valor de tipo delegado de función cuyo tipo devuelto es Task o Task(Of T) para algunos T y que no tiene parámetros ByRef.

Las expresiones lambda pueden ser de una o de varias líneas. Las expresiones lambda Function de línea única contienen una única expresión que representa el valor devuelto del método lambda. Las expresiones lambda Sub de línea única contienen una única instrucción sin el elemento StatementTerminator de cierre. Por ejemplo:

Module Test Sub Do(a() As Integer, action As Action(Of Integer))  
 For index As Integer = 0 To a.Length - 1  
 action(a(index))  
 Next index  
 End Sub  
  
 Sub Main()  
 Dim a() As Integer = { 1, 2, 3, 4 }  
  
 Do(a, Sub(x As Integer) Console.WriteLine(x))  
 End Sub  
End Module

Las construcciones lambda de línea única enlazan con menor presión que el resto de expresiones e instrucciones. Así, por ejemplo, “Function() x + 5” es equivalente a “Function() (x+5)” en lugar de a “(Function() x) + 5”. Para evitar la ambigüedad, una expresión lambda Sub de línea única no puede incluir una instrucción Dim ni una instrucción de declaración de etiqueta. Además, a no ser que se encierre entre paréntesis, una expresión lambda Sub de línea única no puede ir seguida inmediatamente de dos puntos “:”, de un operador de acceso a miembros “.”, de un operador de acceso a miembros de diccionario “!” ni de un paréntesis de apertura “(“. No puede incluir instrucciones de bloque (With, SyncLock, If…EndIf, While, For, Do, Using) ni OnError ni Resume.

Anotación

En la expresión lambda “Function(i) x=i”, el cuerpo se interpreta como una *expresión* (que comprueba si x e i son iguales). Pero en la expresión lambda “Sub(i) x=i”, el cuerpo se interpreta como una instrucción (que asigna i a x).

Una expresión lambda de varias líneas contiene un bloque de instrucciones y debe terminar con una instrucción End adecuada (es decir, End Function o End Sub). Como sucede con los métodos normales, la instrucción Function o Sub y las instrucciones End del método lambda de varias líneas deben estar en sus propias líneas. Por ejemplo:

' Error: Function statement must be on its own line!  
Dim x = Sub(x As Integer) : Console.WriteLine(x) : End Sub  
  
' OK  
Dim y = Sub(x As Integer)  
 Console.WriteLine(x)  
 End Sub

Las expresiones lambda Function de varias líneas pueden declarar un tipo devuelto pero no pueden ponerle atributos. Si una expresión lambda Function de múltiples líneas no declara un tipo devuelto pero el tipo devuelto se puede inferir del contexto en el que se usa la expresión lambda, entonces se usa el tipo devuelto. Si no es así, el tipo devuelto de la función se calcula de la siguiente forma.

1. En una expresión lambda regular, el tipo devuelto es el tipo dominante de la expresión en todas las instrucciones Return del bloque de instrucciones.
2. En una expresión lambda asincrónica, el tipo devuelto es Task(Of T), donde T es el tipo dominante de la expresión en todas las instrucciones Return del bloque de instrucciones.
3. En una expresión lambda de iterador, el tipo devuelto es IEnumerable(Of T), donde T es el tipo dominante de la expresión en todas las instrucciones Yield del bloque de instrucciones.

Por ejemplo:

Function f(min As Integer, max As Integer) As IEnumerable(Of Integer)  
 If min > max Then Throw New ArgumentException()  
 Dim x = Iterator Function()  
 For i = min To max  
 Yield i  
 Next  
 End Function

' infers x to be a delegate with return type IEnumerable(Of Integer)  
 Return x()  
End Function

En cualquier caso, si no hay instrucciones Return (respectivamente, Yield) o no hay tipo dominante entre ellas y se utiliza semántica estricta, se produce un error en tiempo de compilación; de lo contrario el tipo dominante es implícitamente Object.

Observe que el tipo devuelto se calcula a partir de todas las instrucciones Return, aunque sean inalcanzables. Por ejemplo:

' Return type is Double  
Dim x = Function()  
 Return 10  
 Return 10.50  
 End Function

No hay variable devuelta implícita, ya que no hay nombre para la variable.

Los bloques de instrucciones dentro de expresiones lambda de varias líneas tienen las restricciones siguientes:

Las instrucciones On Error y Resume no se permiten, pero sí las instrucciones Try.

Las variables locales estáticas no se pueden declarar en expresiones lambda de varias líneas.

No es posible bifurcar hacia un bloque de instrucciones o fuera de un bloque de instrucciones de una expresión lambda de varias líneas, aunque las reglas de bifurcación normales se aplican dentro del bloque. Por ejemplo:

Label1:  
Dim x = Sub()  
 ' Error: Cannot branch out  
 GoTo Label1  
  
 ' OK: Wholly within the lamba.  
 GoTo Label2:  
 Label2:  
 End Sub  
  
' Error: Cannot branch in  
GoTo Label2

Una expresión lambda es más o menos equivalente a un método anónimo declarado en el tipo contenedor. El ejemplo inicial es más o menos equivalente a:

Module Test  
 Delegate Function IntFunc(x As Integer) As Integer  
  
 Sub Apply(a() As Integer, func As IntFunc)  
 For index As Integer = 0 To a.Length - 1  
 a(index) = func(a(index))  
 Next index  
 End Sub  
  
 Function $Lambda1(x As Integer) As Integer  
 Return x \* 2  
 End Function  
  
 Sub Main()  
 Dim a() As Integer = { 1, 2, 3, 4 }  
  
 Apply(a, AddressOf $Lambda1)  
  
 For Each value In a  
 Console.Write(value & " ")  
 Next value  
 End Sub  
End Module

LambdaExpression ::=  
 SingleLineLambda |  
 MultiLineLambda

SingleLineLambda ::=  
 [ LambdaModifier+ ] Function [ OpenParenthesis [ ParametertList ] CloseParenthesis ] Expression |  
 [ LambdaModifier+ ] Sub [ OpenParenthesis [ ParametertList ] CloseParenthesis ] Statement

MultiLineLambda ::=  
 MultiLineFunctionLambda |  
 MultiLineSubLambda

MultiLineFunctionLambda ::=  
 [ LambdaModifier+ ] Function [ OpenParenthesis [ ParametertList ] CloseParenthesis ] [ As TypeName ] LineTerminator  
 Block  
 End Function

MultiLineSubLambda ::=  
 [ LambdaModifier+ ] Sub [ OpenParenthesis [ ParametertList ] CloseParenthesis ] LineTerminator  
 Block  
 End Sub

LambdaModifier ::=  
 Async |  
 Iterator

### Cierres

Las expresiones lambda tienen acceso a todas las variables del ámbito, incluidas las variables locales o los parámetros definidos en el método contenedor y las expresiones lambda. Cuando una expresión lambda hace referencia a un parámetro o variable local, la expresión lambda captura la variable a la que se hace referencia en un cierre. Un cierre es un objeto que reside en el montón y no en la pila, y cuando se captura una variable, todas las referencias a la variable se redireccionan a ese cierre. Esto permite a las expresiones lambda seguir haciendo referencia a las variables locales o los parámetros incluso después de que finalice el método contenedor. Por ejemplo:

Module Test  
 Delegate Function D() As Integer  
  
 Function M() As D  
 Dim x As Integer = 10  
 Return Function() x  
 End Function  
  
 Sub Main()  
 Dim y As D = M()  
  
 ' Prints 10  
 Console.WriteLine(y())  
 End Sub  
End Module

es más o menos equivalente a:

Module Test  
 Delegate Function D() As Integer  
  
 Class $Closure1  
 Public x As Integer  
  
 Function $Lambda1() As Integer  
 Return x  
 End Function  
 End Class  
  
 Function M() As D  
 Dim c As New $Closure1()  
 c.x = 10  
 Return AddressOf c.$Lambda1  
 End Function  
  
 Sub Main()  
 Dim y As D = M()  
  
 ' Prints 10  
 Console.WriteLine(y())  
 End Sub  
End Module

Un cierre captura una nueva copia de una variable local cada vez que entra en un bloque en el que está declarada la variable local, pero la nueva copia se inicializa con el valor de la copia anterior, si existe. Por ejemplo:

Module Test  
 Delegate Function D() As Integer  
  
 Function M() As D()  
 Dim a(9) As D  
  
 For i As Integer = 0 To 9  
 Dim x  
 a(i) = Function() x  
 x += 1  
 Next i  
  
 Return a  
 End Function  
  
 Sub Main()  
 Dim y() As D = M()  
  
 For i As Integer = 0 To 9  
 Console.Write(y(i)() & " ")  
 Next i  
 End Sub  
End Module

imprime

1 2 3 4 5 6 7 8 9 10

en lugar de

9 9 9 9 9 9 9 9 9 9

Como los cierres tienen que inicializarse cuando se entra en un bloque, no está permitido GoTo a un bloque con cierre desde el exterior del bloque, aunque se permite Resume a un bloque con cierre. Por ejemplo:

Module Test  
 Sub Main()  
 Dim a = 10  
  
 If a = 10 Then  
L1:  
 Dim x = Function() a  
  
 ' Valid, source is within block  
 GoTo L2  
L2:  
 End If  
  
 ' ERROR: target is inside block with closure  
 GoTo L1  
 End Sub  
End Module

Ya que no se pueden capturar en un cierre, los siguientes elementos no pueden aparecer dentro de una expresión lambda:

Parámetros de referencia.

Las expresiones de instancia (Me, MyClass, MyBase), si el tipo de Me no es una clase.

Los miembros de una expresión de creación de tipos anónimos, si la expresión lambda es parte de la expresión. Por ejemplo:

' Error: Lambda cannot refer to anonymous type field  
Dim x = New With { .a = 12, .b = Function() .a }

Las variables o los constructores de instancia ReadOnly o las variables compartidas ReadOnly en constructores compartidos donde las variables se usan en un contexto que no es de valor. Por ejemplo:

Class C1  
 ReadOnly F1 As Integer  
  
 Sub New()  
 ' Valid, doesn’t modify F1  
 Dim x = Function() F1  
  
 ' Error, tries to modify F1  
 Dim f = Function() ModifyValue(F1)  
 End Sub  
  
 Sub ModifyValue(ByRef x As Integer)  
 End Sub  
End Class

## Expresiones de consulta

Una expresión de consulta es aquella que aplica una serie de operadores de consulta a los elementos de una colección consultable. Por ejemplo, la expresión siguiente toma una colección de objetos Customer y devuelve los nombres de todos los clientes del estado de Washington:

Dim names = \_  
 From cust In Customers \_  
 Where cust.State = "WA" \_  
 Select cust.Name

Una expresión de consulta debe comenzar por un operador From o Aggregate y puede terminar con un operador de consulta. El resultado de una expresión de consulta se clasifica como un valor; el tipo de resultado de la expresión depende del tipo de resultado del último operador de consulta de la expresión.

QueryExpression ::=   
 FromOrAggregateQueryOperator |  
 QueryExpression QueryOperator

FromOrAggregateQueryOperator ::= FromQueryOperator | AggregateQueryOperator

*JoinOrGroupJoinQueryOperator ::= JoinQueryOperator* | GroupJoinQueryOperator

QueryOperator ::=  
 FromQueryOperator |  
 AggregateQueryOperator |  
 SelectQueryOperator |  
 DistinctQueryOperator |  
 WhereQueryOperator |  
 OrderByQueryOperator |  
 PartitionQueryOperator |  
 LetQueryOperator |  
 GroupByQueryOperator |  
 JoinOrGroupJoinQueryOperator

### Variables de intervalo

Algunos operadores de consulta introducen una modalidad de variable especial denominada variable de intervalo. Las variables de intervalo no son variables reales, más bien representan los valores individuales durante la evaluación de la consulta en las colecciones de entrada. Las variables de intervalo tienen el ámbito del operador de consulta introductor al final de la expresión de consulta, o de un operador de consulta como Select que las oculta. Por ejemplo, en la consulta siguiente

Dim waCusts = \_  
 From cust As Customer In Customers \_  
 Where cust.State = "WA"

El operador de consulta From presenta una variable de intervalo cust con tipo Customer que representa cada cliente de la colección Customers. El operador de consulta siguiente Where hace referencia a la variable de intervalo cust en la expresión de filtro para determinar si se filtra un cliente concreto fuera de la colección resultante.

Hay dos tipos de variables de intervalo: variables de intervalo de colección y variables de intervalo de expresión. Las variables de intervalo de colección toman sus valores de los elementos de la colección que se está consultando. La expresión de colección en una declaración de variable de intervalo de colección debe clasificarse como un valor cuyo tipo es consultable. Si se omite el tipo de la variable de intervalo de colección, se infiere que es el tipo de elemento de la colección u Object si la expresión de la colección no tiene un tipo de elemento (esto es, solo define un método Cast). Si la expresión de la colección no es consultable (es decir, el tipo de elemento de la colección no se puede inferir), se produce un error en tiempo de compilación.

Una variable de intervalo de expresión es una variable de intervalo cuyo valor se calcula mediante una expresión en lugar de una colección. En el ejemplo siguiente, el operador de consulta Select presenta una variable de intervalo de expresión denominada cityState calculada a partir de dos campos:

Dim cityStates = \_  
 From cust As Customer In Customers \_  
 Select cityState = cust.City & "," & cust.State \_  
 Where cityState.Length() < 10

No se requiere una variable de intervalo de expresión para hacer referencia a otra variable de intervalo, aunque dicha variable puede ser de valor dudoso. La expresión asignada a una variable de intervalo de expresión debe clasificarse como un valor y debe ser de un tipo que se pueda convertir implícitamente en el tipo de la variable de intervalo, si se proporciona.

Una variable de intervalo de expresión solo puede tener su tipo especificado en un operador Let. En otros operadores, o si no se especifica su tipo, se usa la inferencia de tipos de variable local para determinar el tipo de la variable de intervalo.

Una variable de intervalo debe cumplir las reglas para declarar variables locales relativas al ocultamiento. En consecuencia, una variable de intervalo no puede ocultar el nombre de un parámetro o variable local en el método contenedor u otra variable de intervalo (salvo que el operador de consulta oculte específicamente todas las variables de intervalo actualmente en ámbito).

CollectionRangeVariableDeclarationList ::=  
 CollectionRangeVariableDeclaration |  
 CollectionRangeVariableDeclarationList Comma CollectionRangeVariableDeclaration

CollectionRangeVariableDeclaration ::=   
 Identifier [ As TypeName ] In [ LineTerminator ] Expression

ExpressionRangeVariableDeclarationList ::=  
 ExpressionRangeVariableDeclaration |  
 ExpressionRangeVariableDeclarationList Comma ExpressionRangeVariableDeclaration

ExpressionRangeVariableDeclaration ::=   
 [ Identifier [ As TypeName ] Equals ] Expression

### Tipos consultables

Las expresiones de consulta se implementan traduciendo la expresión en llamadas a métodos bien conocidos de un tipo de colección. Estos métodos definen el tipo de elemento de la colección consultable, así como los tipos de resultado de los operadores de consulta ejecutados en la colección. Cada operador de consulta especifica el método o métodos al que se traduce por lo general el operador de consulta, aunque la traducción específica es dependiente de la implementación. Estos métodos se ofrecen en la especificación con un formato general que tiene este aspecto:

Function Select(selector As Func(Of T, R)) As CR

Lo siguiente se aplica a los métodos:

El método debe ser un miembro de instancia o de extensión del tipo de colección, y debe ser accesible.

El método puede ser genérico, siempre que sea posible inferir todos los argumentos de tipo.

El método puede estar sobrecargado, en cuyo caso se usa la resolución de sobrecarga para determinar el método exacto que se va a emplear.

Otro tipo de delegado se puede usar en lugar del tipo de delegado Func, siempre que tenga la misma signatura, incluido el tipo devuelto, que el tipo Func coincidente.

El tipo System.Linq.Expressions.Expression(Of D) se puede usar en lugar del tipo de delegado Func, siempre que D sea un tipo delegado que tenga la misma signatura, incluido el tipo devuelto, que el tipo Func coincidente.

El tipo T representa el tipo de elemento de la colección de entrada. Todos los métodos definidos en un tipo de colección deben tener el mismo tipo de elemento de entrada para el tipo de colección que será consultable.

El tipo S representa el tipo de elemento de la segunda colección de entrada en el caso de los operadores de consulta que llevan a cabo combinaciones.

El tipo K representa un tipo clave en el caso de los operadores de consulta que tienen un grupo de variables de intervalo que actúan como claves.

El tipo N representa un tipo que es usa como tipo numérico (aunque podría seguir siendo un tipo definido por el usuario en lugar de un tipo numérico intrínseco).

El tipo B representa un tipo que puede usarse en una expresión booleana.

El tipo R representa el tipo de elemento de la colección de resultado, si el operador de consulta produce una colección de resultado. R depende del número de variables de intervalo en el ámbito en la finalización del operador de consulta. Si solo hay una variable de intervalo en el ámbito, R es el tipo de la variable de intervalo. En el ejemplo

Dim custNames = \_  
 From c In Customers \_  
 Select c.Name

el resultado de la consulta será un tipo de colección con un tipo de elemento String. Si hay varias variables de intervalo en ámbito, R es un tipo anónimo que contiene todas las variables de intervalo en el ámbito como campos Key. En el siguiente ejemplo:

Dim custAndOrderNames = \_  
 From c In Customers, o In c.Orders \_  
 Select Name = c.Name, ProductName = o.ProductName

el resultado de la consulta será un tipo de colección con un tipo de elemento de un tipo anónimo con una propiedad de solo lectura denominada Name de tipo String y una propiedad de solo lectura denominada ProductName de tipo String.

Dentro de una expresión de consulta, los tipos anónimos generados para contener variables de intervalo son transparentes, lo que significa que las variables de intervalo siempre están disponibles sin calificación. Por ejemplo, en el ejemplo previo se podría tener acceso a las variables de intervalo c y o sin calificación en el operador de consulta Select, incluso aunque el tipo de elemento de la colección fuera un tipo anónimo.

El tipo CX representa un tipo de colección, no necesariamente el tipo de colección de entrada, cuyo tipo de elemento es algún tipo X.

Un tipo de colección consultable debe cumplir una de las siguientes condiciones, en orden de prioridad:

Debe definir un método Select.

Debe tener uno de los métodos siguientes

Function AsEnumerable() As CT  
Function AsQueryable() As CT

que se pueda invocar para obtener una colección consultable. Si se proporcionan ambos métodos, se prefiere AsQueryable a AsEnumerable.

Debe tener un método

Function Cast(Of T)() As CT

que se pueda invocar con el tipo de la variable de intervalo para producir una colección consultable.

Como el tipo de elemento de una colección se determina de forma independiente de una invocación de un método real, no es posible determinar la aplicabilidad de métodos concretos. Por consiguiente, cuando se determina el tipo de elemento de una colección si hay métodos de instancia que coinciden con métodos bien conocidos, cualquier método de extensión que coincida con métodos bien conocidos se pasa por alto.

La traducción de los operadores de consulta ocurre en el orden en que los operadores de consulta aparecen en la expresión. No es necesario que un objeto de colección implemente todos los métodos que necesitan los operadores de consulta, aunque cada objeto de colección debe al menos admitir el operador de consulta Select. Se produce un error en tiempo de compilación si un método necesario no está presente. Cuando se enlaza con nombres de métodos bien conocidos, los que no son métodos se pasan por alto con el fin de la herencia múltiple en el enlace de métodos de extensión y de interfaz, aunque siga vigente la semántica de ocultamiento. Por ejemplo:

Class Q1  
 Public Function [Select](selector As Func(Of Integer, Integer)) As Q1  
 End Function  
End Class  
  
Class Q2  
 Inherits Q1  
  
 Public [Select] As Integer  
End Class  
  
Module Test  
 Sub Main()  
 Dim qs As New Q2()  
  
 ' Error: Q2.Select still hides Q1.Select  
 Dim zs = From q In qs Select q  
 End Sub  
End Module

### Indizador de consultas predeterminado

Cada tipo de colección consultable cuyo tipo de elemento es T y no tenga una propiedad predeterminada se considera que tiene una propiedad predeterminada con la forma general siguiente:

Public ReadOnly Default Property Item(index As Integer) As T  
 Get  
 Return Me.ElementAtOrDefault(index)  
 End Get  
End Property

Solo se puede hacer referencia a la propiedad predeterminada usando la sintaxis de acceso de propiedad predeterminada; no se puede hacer referencia a la propiedad predeterminada por el nombre. Por ejemplo:

Dim customers As IEnumerable(Of Customer) = ...  
Dim customerThree = customers(2)  
  
' Error, no such property  
Dim customerFour = customers.Item(4)

Si el tipo de colección no tiene un miembro ElementAtOrDefault, se producirá un error en tiempo de compilación.

### Operador de consulta From

El operador de consulta From presenta una variable de intervalo de colección que representa los miembros individuales de la colección que se consultará. Por ejemplo, la expresión de consulta:

From c As Customer In Customers ...

puede verse como equivalente a

For Each c As Customer In Customers  
 ...  
Next c

Cuando un operador de consulta From declara múltiples variables de intervalo de colección o no es el primer operador de consulta From en la expresión de consulta, cada nueva variable de intervalo de colección tiene una combinación cruzada con el grupo existente de variables de intervalo. El resultado es que la consulta se evalúa en el producto cruzado de todos los elementos en las colecciones combinadas. Por ejemplo, la expresión:

From c In Customers \_  
From e In Employees \_  
...

puede verse como equivalente a:

For Each c In Customers  
 For Each e In Employees  
 ...  
 Next e  
Next c

y equivale exactamente a:

From c In Customers, e In Employees ...

Las variables de intervalo introducidas en los operadores de consulta previos se pueden usar más tarde en un operador de consulta From. Por ejemplo, en la expresión siguiente, el segundo operador de consulta From hace referencia al valor de la primera variable de intervalo:

From c As Customer In Customers \_  
From o As Order In c.Orders \_  
Select c.Name, o

Las variables de intervalo en un operador de consulta From o varios operadores de consulta From solo se admiten si el tipo de colección contiene uno o ambos de los métodos siguientes:

Function SelectMany(selector As Func(Of T, CR)) As CR  
Function SelectMany(selector As Func(Of T, CS), \_  
 resultsSelector As Func(Of T, S, R)) As CR

El código

Dim xs() As Integer = ...  
Dim ys() As Integer = ...  
Dim zs = From x In xs, y In ys ...

se traduce generalmente a

Dim xs() As Integer = ...  
Dim ys() As Integer = ...  
Dim zs = \_  
 xs.SelectMany( \_  
 Function(x As Integer) ys, \_  
 Function(x As Integer, y As Integer) New With {x, y})...

Nota   From no es una palabra reservada.

FromQueryOperator ::=  
 [ LineTerminator ] From [ LineTerminator ] CollectionRangeVariableDeclarationList

### Operador de consulta Join

El operador de consulta Join combina variables de intervalo existentes con una nueva variable de intervalo de colección, y da como resultado una única colección cuyos elementos se han combinado basándose en una expresión de igualdad. Por ejemplo:

Dim customersAndOrders = \_  
 From cust In Customers \_  
 Join ord In Orders On cust.ID Equals ord.CustomerID

La expresión de igualdad es más restringida que una expresión de igualdad normal:

Ambas expresiones deben clasificarse como valor.

Ambas expresiones deben hacer referencia al menos a una variable de intervalo.

Una de las expresiones debe hacer referencia a la variable de intervalo declarada en el operador de consulta combinada y la expresión no debe hacer referencia a ninguna otra variable.

Si los tipos de las dos expresiones no son del mismo tipo exacto, entonces

Si el operador de igualdad se define para los dos tipos, ambas expresiones son convertibles implícitamente en él y no es Object, entonces se convierten ambas en ese tipo.

De lo contrario, si hay un tipo dominante en el que ambas expresiones son convertibles implícitamente, se convierten ambas en ese tipo.

En caso contrario, se producirá un error en tiempo de compilación.

Las expresiones se comparan usando valores hash (esto es, llamando a GetHashCode()) en lugar de usar operadores de igualdad por cuestiones de eficacia. Un operador de consulta Join puede hacer múltiples combinaciones o condiciones de igualdad en el mismo operador. Solo se admite un operador de consulta Join si el tipo de colección contiene un método:

Function Join(inner As CS, \_  
 outerSelector As Func(Of T, K), \_  
 innerSelector As Func(Of S, K), \_  
 resultSelector As Func(Of T, S, R)) As CR

El código

Dim xs() As Integer = ...  
Dim ys() As Integer = ...  
Dim zs = From x In xs \_  
 Join y In ys On x Equals y \_  
 ...

se traduce generalmente a

Dim xs() As Integer = ...  
Dim ys() As Integer = ...  
Dim zs = \_  
 xs.Join( \_  
 ys, \_  
 Function(x As Integer) x, \_  
 Function(y As Integer) y, \_  
 Function(x As Integer, y As Integer) New With {x, y})...

Nota   Join, On y Equals no son palabras reservadas.

JoinQueryOperator ::=  
 [ LineTerminator ] Join [ LineTerminator ] CollectionRangeVariableDeclaration  
 [ JoinOrGroupJoinQueryOperator ] [ LineTerminator ] On [ LineTerminator ] JoinConditionList

JoinConditionList ::=  
 JoinCondition |  
 JoinConditionList And [ LineTerminator ] JoinCondition

JoinCondition ::= Expression Equals [ LineTerminator ] Expression

### Operador de consulta Let

El operador de consulta Let presenta una variable de intervalo de expresión. Esto permite calcular un valor intermedio una vez que se usará varias veces en operadores de consulta. Por ejemplo:

Dim taxedPrices = \_  
 From o In Orders \_  
 Let tax = o.Price \* 0.088 \_  
 Where tax > 3.50 \_  
 Select o.Price, tax, total = o.Price + tax

puede verse como equivalente a:

For Each o In Orders  
 Dim tax = o.Price \* 0.088  
 ...  
Next o

Solo se admite un operador de consulta Let si el tipo de colección contiene un método:

Function Select(selector As Func(Of T, R)) As CR

El código

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Let y = x \* 10 \_  
 ...

se traduce generalmente a

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.Select(Function(x As Integer) New With {x, .y = x \* 10})...

LetQueryOperator ::=  
 [ LineTerminator ] Let [ LineTerminator ] ExpressionRangeVariableDeclarationList

### Operador de consulta Select

El operador de consulta Select es como Let porque presenta variables de intervalo de expresión; sin embargo, un operador de consulta Select oculta las variables de intervalo existentes disponibles, en lugar de agregarse a ellas. Asimismo, el tipo de una variable de intervalo de expresión que presenta un operador de consulta Select se infiere siempre aplicando las reglas de inferencia de tipos de variable local; no se puede especificar un tipo explícito y, si no se puede inferir ningún tipo, se produce un error en tiempo de compilación. Por ejemplo, en la consulta:

Dim smiths = \_  
 From cust In Customers \_  
 Select name = cust.name \_  
 Where name.EndsWith("Smith")

el operador de consulta Where solo tiene acceso a la variable de intervalo name introducida por el operador Select; si el operador Where hubiera intentado hacer referencia a cust, se habría producido un error en tiempo de compilación.

En lugar de especificar de forma explícita los nombres de las variables de intervalo, un operador de consulta Select puede inferir los nombres de las variables de intervalo usando las mismas reglas que las expresiones de creación de objetos de tipo anónimos. Por ejemplo:

Dim custAndOrderNames = \_  
 From cust In Customers, ord In cust.Orders \_  
 Select cust.name, ord.ProductName \_  
 Where name.EndsWith("Smith")

Si el nombre de la variable de intervalo no se proporciona y no se puede inferir ningún nombre, se produce un error en tiempo de compilación. Si el operador de consulta Select contiene únicamente una expresión, no se produce ningún error si no se puede inferir un nombre para la variable de intervalo, pero la variable de intervalo se queda sin nombre. Por ejemplo:

Dim custAndOrderNames = \_  
 From cust In Customers, ord In cust.Orders \_  
 Select cust.Name & " bought " & ord.ProductName \_  
 Take 10

Si se produce ambigüedad en un operador de consulta Select entre asignar un nombre a una variable de intervalo y a una expresión de igualdad, se prefiere la asignación de nombre. Por ejemplo:

Dim badCustNames = \_  
 From c In Customers \_  
 Let name = "John Smith" \_  
 Select name = c.Name ' Creates a range variable named "name"

Dim goodCustNames = \_  
 From c In Customers \_  
 Let name = "John Smith" \_  
 Select match = (name = c.Name)

Cada expresión del operador de consulta Select debe clasificarse como un valor. Solo se admite un operador de consulta Select si el tipo de colección contiene un método:

Function Select(selector As Func(Of T, R)) As CR

El código

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Select x, y = x \* 10 \_  
 ...

se traduce generalmente a

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.Select(Function(x As Integer) New With {x, .y = x \* 10})...

SelectQueryOperator ::=  
 [ LineTerminator ] Select [ LineTerminator ] ExpressionRangeVariableDeclarationList

### Operador de consulta Distinct

El operador de consulta Distinct restringe los valores de una colección solo a aquellos con valores distintos, según se determina comparando la igualdad del tipo de elemento. Por ejemplo, la consulta:

Dim distinctCustomerPrice = \_  
 From cust In Customers, ord In cust.Orders \_  
 Select cust.Name, ord.Price \_  
 Distinct

solo devolverá una fila por cada emparejamiento distinto de nombre de cliente y precio de pedido, aunque el cliente tenga varios pedidos con el mismo precio. Solo se admite un operador de consulta Distinct si el tipo de colección contiene un método:

Function Distinct() As CT

El código

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Distinct \_  
 ...

se traduce generalmente a

Dim xs() As Integer = ...  
Dim zs = xs.Distinct()...

Nota   Distinct no es una palabra reservada.

DistinctQueryOperator ::= [ LineTerminator ] Distinct [ LineTerminator ]

### Operador de consulta Where

El operador de consulta Where restringe los valores de una colección a los que satisfacen cierta condición dada. Un operador de consulta Where toma una expresión booleana que se evalúa para cada grupo de valores de variable de intervalo; si el valor de la expresión es verdadero, los valores aparecen en la colección de resultado, en caso contrario, se pasan por alto. Por ejemplo, la expresión de consulta:

From cust In Customers, ord In Orders \_  
Where cust.ID = ord.CustomerID \_  
...

puede verse como equivalente al bucle anidado

For Each cust In Customers  
 For Each ord In Orders  
 If cust.ID = ord.CustomerID Then  
 ...  
 End If  
 Next ord  
Next cust

Solo se admite un operador de consulta Where si el tipo de colección contiene un método:

Function Where(predicate As Func(Of T, B)) As CT

El código

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Where x < 10 \_  
 ...

se traduce generalmente a

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.Where(Function(x As Integer) x < 10)...

Nota   Where no es una palabra reservada.

WhereQueryOperator ::= [ LineTerminator ] Where [ LineTerminator ] BooleanExpression

### Operadores de consulta de partición

El operador de consulta Take da como resultado los primeros elementos n de una colección. Cuando se usa con el modificador While, el operador Take da como resultado los primeros elementos n de una colección que satisfacen una expresión booleana. El operador de consulta Skip pasa por alto los primeros elementos n de una colección y devuelve lo que queda de la colección. Cuando se usa con el modificador While, el operador Skip pasa por alto los primeros elementos n de una colección que satisfacen una expresión booleana y devuelve lo que queda de la colección. Las expresiones de un operador de consulta Take o Skip deben clasificarse como un valor.

Solo se admite un operador de consulta Take si el tipo de colección contiene un método:

Function Take(count As N) As CT

Solo se admite un operador de consulta Skip si el tipo de colección contiene un método:

Function Skip(count As N) As CT

Solo se admite un operador de consulta Take While si el tipo de colección contiene un método:

Function TakeWhile(predicate As Func(Of T, B)) As CT

Solo se admite un operador de consulta Skip While si el tipo de colección contiene un método:

Function SkipWhile(predicate As Func(Of T, B)) As CT

El código

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Skip 10 \_  
 Take 5 \_  
 Skip While x < 10 \_  
 Take While x > 5 \_  
 ...

se traduce generalmente a

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.Skip(10). \_  
 Take(5). \_  
 SkipWhile(Function(x) x < 10). \_  
 TakeWhile(Function(x) x > 5)...

Nota   Take y Skip no son palabras reservadas.

PartitionQueryOperator ::=   
 [ LineTerminator ] Take [ LineTerminator ] Expression |  
 [ LineTerminator ] Take While [ LineTerminator ] BooleanExpression |  
 [ LineTerminator ] Skip [ LineTerminator ] Expression |  
 [ LineTerminator ] Skip While [ LineTerminator ] BooleanExpression

### Operador de consulta Order By

El operador de consulta Order By ordena los valores que aparecen en las variables de intervalo. Un operador de consulta Order By toma expresiones que especifican los valores clave que se deberían usar para ordenar las variables de iteración. Por ejemplo, la consulta siguiente devuelve productos ordenados por precio:

Dim productsByPrice = \_  
 From p In Products \_  
 Order By p.Price \_  
 Select p.Name

Un pedido se puede marcar como Ascending, en cuyo caso los valores menores van antes que los mayores, o Descending, en cuyo caso los valores mayores van antes que los menores. El criterio de ordenación predeterminado si no se especifica ninguno es Ascending. Por ejemplo, la consulta siguiente devuelve productos ordenados por precio, con el producto más caro en primer lugar:

Dim productsByPriceDesc = \_  
 From p In Products \_  
 Order By p.Price Descending \_  
 Select p.Name

El operador de consulta Order By puede especificar varias expresiones de ordenamiento, en cuyo caso la colección se ordena de un modo anidado. Por ejemplo, la consulta siguiente ordena los clientes por estado, luego por ciudad dentro del estado y después por el código postal de cada ciudad:

Dim customersByLocation = \_  
 From c In Customers \_  
 Order By c.State, c.City, c.ZIP \_  
 Select c.Name, c.State, c.City, c.ZIP

Las expresiones de un operador de consulta Order By deben clasificarse como un valor. Solo se admite un operador de consulta Order By si el tipo de colección contiene uno o ambos de los métodos siguientes:

Function OrderBy(keySelector As Func(Of T, K)) As CT  
Function OrderByDescending(keySelector As Func(Of T, K)) As CT

El tipo de valor devuelto CT debe ser una colección ordenada. Una colección ordenada es un tipo de colección que contiene uno o ambos de los métodos siguientes:

Function ThenBy(keySelector As Func(Of T, K)) As CT  
Function ThenByDescending(keySelector As Func(Of T, K)) As CT

El código

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Order By x Ascending, x Mod 2 Descending \_  
 ...

se traduce generalmente a

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.OrderBy(Function(x) x).ThenByDescending(Function(x) x Mod 2)...

Anotación

Como los operadores de consulta simplemente asignan sintaxis a métodos que implementan una operación de consulta concreta, la conservación del orden no lo dicta el lenguaje, sino que lo determina la implementación del propio operador. Por tanto es muy similar a los operadores definidos por el usuario en que la implementación para sobrecargar el operador de adición para un tipo numérico definido por el usuario no puede efectuar nada parecido a una adición. Por supuesto que para mantener la previsibilidad, no se recomienda implementar algo que no responda a las expectativas de los usuarios.

Nota   Order y By no son palabras reservadas.

OrderByQueryOperator ::= [ LineTerminator ] Order By [ LineTerminator ] OrderExpressionList

OrderExpressionList ::=  
 OrderExpression |  
 OrderExpressionList Comma OrderExpression

OrderExpression ::=  
 Expression [ Ordering ]

Ordering ::= Ascending | Descending

### Operador de consulta Group By

El operador de consulta Group By agrupa las variables de intervalo del ámbito basándose en una o más expresiones, y después produce nuevas variables de intervalo basándose en estas agrupaciones. Por ejemplo, la consulta siguiente agrupa todos los clientes por State y luego calcula la cuenta y el promedio de edad de cada grupo:

Dim averageAges = \_  
 From cust In Customers \_  
 Group By cust.State \_  
 Into Count(), Average(cust.Age)

El operador de consulta Group By tiene tres cláusulas: la cláusula opcional Group, la cláusula By y la cláusula Into. La cláusula Group tiene la misma sintaxis y efecto que un operador de consulta Select, salvo que solo afecta a las variables de intervalo de la cláusula Into y no de la cláusula By. Por ejemplo:

Dim averageAges = \_  
 From cust In Customers \_  
 Group cust.Age By cust.State \_  
 Into Count(), Average(Age)

La cláusula By declara variables de intervalo de expresión que se usan como valores clave en la operación de agrupamiento. La cláusula Into permite la declaración de variables de intervalo de expresión que calculan agregaciones sobre cada uno de los grupos formados por la cláusula By. Dentro de la cláusula Into, la variable de intervalo de expresión solo se puede asignar a una expresión que sea una invocación al método de una función de agregado. Una función de agregado es una función en el tipo de colección del grupo (que no tiene por qué ser el mismo tipo de colección que la colección original) que se parece a uno de estos métodos:

Function <name>() As <type>  
Function <name>(selector As Func(Of T, R)) As R

Si una función de agregado toma un argumento de delegado, la expresión de invocación puede tener una expresión de argumento que debe clasificarse como valor. La expresión de argumento puede usar las variables de intervalo que haya en el ámbito; dentro de la llamada a una función de agregado, esas variables de expresión representan los valores del grupo que se está formando, no todos los valores de la colección. Por ejemplo, en el ejemplo original de esta sección, la función Average calcula el promedio de las edades de los clientes por estado, en lugar de para todos los clientes juntos.

Todos los tipos de colección se considera que tiene la función de agregado Group definida, que no toma parámetros y simplemente devuelve el grupo. Otras funciones agregadas estándar que un tipo de colección puede proporcionar son:

Count y LongCount, que devuelven la cuenta de los elementos del grupo o la cuenta de los elementos del grupo que satisface una expresión booleana. Count y LongCount solo se admiten si el tipo de colección contiene uno de los métodos:

Function Count() As N  
Function Count(selector As Func(Of T, B)) As N  
Function LongCount() As N  
Function LongCount(selector As Func(Of T, B)) As N

Sum, que devuelve la suma de una expresión en todos los elementos del grupo. Sum solo se admite si el tipo de colección contiene uno de los métodos:

Function Sum() As N  
Function Sum(selector As Func(Of T, N)) As N

Min, que devuelve el valor mínimo de una expresión en todos los elementos del grupo. Min solo se admite si el tipo de colección contiene uno de los métodos:

Function Min() As N  
Function Min(selector As Func(Of T, N)) As N

Max, que devuelve el valor máximo de una expresión en todos los elementos del grupo. Max solo se admite si el tipo de colección contiene uno de los métodos:

Function Max() As N  
Function Max(selector As Func(Of T, N)) As N

Average, que devuelve el promedio de una expresión en todos los elementos del grupo. Average solo se admite si el tipo de colección contiene uno de los métodos:

Function Average() As N  
Function Average(selector As Func(Of T, N)) As N

Any, que determina si un grupo contiene miembros o si una expresión booleana es verdadera para alguno de los elementos del grupo. Any devuelve un valor que se puede usar en una expresión booleana y solo se admite si el tipo de colección contiene uno de los métodos:

Function Any() As B  
Function Any(predicate As Func(Of T, B)) As B

All, que determina si un grupo una expresión booleana es verdadera para todos los elementos del grupo. All devuelve un valor que se puede usar en una expresión booleana y solo se admite si el tipo de colección contiene un método:

Function All(predicate As Func(Of T, B)) As B

Después de un operador de consulta Group By, las variables de intervalo que había previamente en el ámbito se ocultan y las introducidas por las cláusulas By e Into están disponibles. Solo se admite un operador de consulta Group By si el tipo de colección contiene el método:

Function GroupBy(keySelector As Func(Of T, K), \_  
 resultSelector As Func(Of K, CT, R)) As CR

Las declaraciones de variables de intervalo de la cláusula Group solo se admiten si el tipo de colección contiene el método:

Function GroupBy(keySelector As Func(Of T, K), \_  
 elementSelector As Func(Of T, S), \_  
 resultSelector As Func(Of K, CS, R)) As CR

El código

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Group y = x \* 10, z = x / 10 By evenOdd = x Mod 2 \_  
 Into Sum(y), Average(z) \_  
 ...

se traduce generalmente a

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.GroupBy( \_  
 Function(x As Integer) x Mod 2, \_  
 Function(x As Integer) New With {.y = x \* 10, .z = x / 10}, \_  
 Function(evenOdd, group) New With { \_  
 evenOdd, \_  
 .Sum = group.Sum(Function(e) e.y), \_  
 .Average = group.Average(Function(e) e.z)})...

Nota   Group, By e Into no son palabras reservadas.

GroupByQueryOperator ::=  
 [ LineTerminator ] Group [ [ LineTerminator ] ExpressionRangeVariableDeclarationList ]  
 [ LineTerminator ] By [ LineTerminator ] ExpressionRangeVariableDeclarationList   
 [ LineTerminator ] Into [ LineTerminator ] ExpressionRangeVariableDeclarationList

### Operador de consulta Aggregate

El operador de consulta Aggregate lleva a cabo una función similar a la del operador Group By, con la excepción de que permite agregar a grupos que ya se han formado. Como el grupo ya está formado, la cláusula Into de un operador de consulta Aggregate no oculta las variables de intervalo del ámbito (en este sentido, Aggregate se parece más a Let y Group By se parece más a Select). Por ejemplo, la consulta siguiente agrega el total de todos los pedidos hechos por clientes en Washington:

Dim orderTotals = \_  
 From cust In Customers \_  
 Where cust.State = "WA" \_  
 Aggregate order In cust.Orders \_  
 Into Sum(order.Total)

El resultado de la consulta es una colección cuyo tipo de elemento es un tipo anónimo con una propiedad denominada cust de tipo Customer y una propiedad denominada Sum de tipo Integer.

A diferencia de Group By, los operadores de consulta adicionales se pueden colocar entre las cláusulas Aggregate e Into. Entre una cláusula Aggregate y el final de la cláusula Into, pueden usarse todas las variables de intervalo en el ámbito, incluidas las que declara la cláusula Aggregate. Por ejemplo, la consulta siguiente agrega la suma total de todos los pedidos hechos por clientes en Washington antes de 2006:

Dim orderTotals = \_  
 From cust In Customers \_  
 Where cust.State = "WA" \_  
 Aggregate order In cust.Orders \_  
 Where order.Date <= #01/01/2006# \_  
 Into Sum = Sum(order.Total)

El operador Aggregate también se puede usar para iniciar una expresión de consulta. En este caso, el resultado de la expresión de consulta será el único valor que calcule la cláusula Into. Por ejemplo, la consulta siguiente calcula la suma de todos los totales de los pedidos anteriores al 1 de enero de 2006:

Dim ordersTotal = \_  
 Aggregate order In Orders \_  
 Where order.Date <= #01/01/2006# \_  
 Into Sum(order.Total)

El resultado es un único valor Integer. Un operador de consulta Aggregate siempre está disponible (aunque la función de agregado también debe estar disponible para que la expresión sea válida). El código

Dim xs() As Integer = ...  
Dim zs = \_  
 Aggregate x In xs \_  
 Where x < 5 \_  
 Into Sum()

se traduce generalmente a

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.Where(Function(x) x < 5).Sum()

Nota   Aggregate e Into no son palabras reservadas.

AggregateQueryOperator ::=  
 [ LineTerminator ] Aggregate [ LineTerminator ] CollectionRangeVariableDeclaration   
 [ QueryOperator+ ]   
 [ LineTerminator ] Into [ LineTerminator ] ExpressionRangeVariableDeclarationList

### Operador de consulta Group Join

El operador de consulta Group Join combina las funciones de los operadores de consulta Join y Group By en un operador único. Group Join combina dos colecciones basadas en claves coincidentes extraídas de los elementos mediante la agrupación de todos los elementos del lado derecho de la combinación que coincide con un elemento concreto del lado izquierdo de la combinación. De esta forma, el operador produce un grupo de resultados jerárquicos. Por ejemplo, la consulta siguiente obtiene elementos que contienen un único nombre de cliente, un grupo de todos sus pedidos y la cantidad total de todos esos pedidos:

Dim custsWithOrders = \_  
 From cust In Customers \_  
 Group Join order In Orders On cust.ID Equals order.CustomerID \_   
 Into Orders = Group, OrdersTotal = Sum(order.Total) \_  
 Select cust.Name, Orders, OrdersTotal

El resultado de la consulta es una colección cuyo tipo de elemento es un tipo anónimo con tres propiedades: Name con tipo String, Orders cuyo tipo es una colección con el tipo de elemento Order y OrdersTotal, cuyo tipo es Integer. Solo se admite un operador de consulta Group Join si el tipo de colección contiene el método:

Function GroupJoin(inner As CS, \_  
 outerSelector As Func(Of T, K), \_  
 innerSelector As Func(Of S, K), \_  
 resultSelector As Func(Of T, CS, R)) As CR

El código

Dim xs() As Integer = ...  
Dim ys() As Integer = ...  
Dim zs = From x In xs \_  
 Group Join y in ys On x Equals y \_  
 Into g = Group \_  
 ...

se traduce generalmente a

Dim xs() As Integer = ...  
Dim ys() As Integer = ...  
Dim zs = \_  
 xs.GroupJoin( \_  
 ys, \_  
 Function(x As Integer) x, \_  
 Function(y As Integer) y, \_  
 Function(x, group) New With {x, .g = group})...

Nota   Group, Join e Into no son palabras reservadas.

GroupJoinQueryOperator ::=  
 [ LineTerminator ] Group Join [ LineTerminator ] CollectionRangeVariableDeclaration  
 [ JoinOrGroupJoinQueryOperator ] [ LineTerminator ] On [ LineTerminator ] JoinConditionList  
 [ LineTerminator ] Into [ LineTerminator ] ExpressionRangeVariableDeclarationList

## Expresiones condicionales

Una expresión condicional If comprueba una expresión y devuelve un valor. A diferencia de la función en tiempo de ejecución IIF, sin embargo, una expresión condicional solo evalúa sus operandos si es necesario. De este modo por ejemplo, la expresión If(c Is Nothing, c.Name, "Unknown") no producirá ninguna excepción si el valor de c es Nothing. La expresión condicional tiene dos formas: una que toma dos operandos y otra que toma tres operandos.

Si se proporcionan tres operandos, las tres expresiones deben clasificarse como valores y el primer operando debe ser una expresión booleana. Si el resultado de la expresión es verdadero, la segunda expresión será el resultado del operador, de lo contrario la tercera expresión será el resultado del operador. El tipo resultante de la expresión es el tipo dominante entre los tipos de la segunda y tercera expresión. Si no hay un tipo dominante, se produce un error en tiempo de compilación.

Si se proporcionan dos operandos, ambos deben clasificarse como valores y el primero debe ser un tipo de referencia o tipo de valor que admite valores null. Después se evalúa la expresión If(x, y) como si fuera la expresión If(x IsNot Nothing, x, y), con dos excepciones. Primera, la primera expresión solo se evalúa una vez y segunda, si el tipo del segundo operando es un tipo de valor que no admite valores null y el tipo del primero sí, se quita ? del tipo del primer operando cuando se determine el tipo dominante para el tipo del resultado de la expresión. Por ejemplo:

Module Test  
 Sub Main()  
 Dim x?, y As Integer  
 Dim a?, b As Long  
  
 a = If(x, a) ' Result type: Long?  
 y = If(x, 0) ' Result type: Integer  
 End Sub  
End Module

En ambas formas de expresión, si un operando es Nothing, su tipo no se usa para determinar el tipo dominante. En el caso de la expresión If(<expression>, Nothing, Nothing), el tipo dominante se considera Object.

ConditionalExpression ::=   
 If OpenParenthesis BooleanExpression Comma Expression Comma Expression CloseParenthesis |  
 If OpenParenthesis Expression Comma Expression CloseParenthesis

## Expresiones literales XML

Una expresión literal XML representa un valor de XML (eXtensible Markup Language) 1.0. El resultado de una expresión literal XML es un valor cuyo tipo es uno de los tipos del espacio de nombres System.Xml.Linq. Si los tipos del espacio de nombres no están disponibles, la expresión literal XML dará error en tiempo de compilación. Los valores se generan a través de las llamadas al constructor traducidas a partir de la expresión literal XML. Por ejemplo, el código:

Dim book As System.Xml.Linq.XElement = \_  
 <book title="My book"></book>

es más o menos equivalente a:

Dim book As System.Xml.Linq.XElement = \_  
 New System.Xml.Linq.XElement( \_  
 "book", \_  
 New System.Xml.Linq.XAttribute("title", "My book"))

Una expresión literal XML puede adoptar la forma de un documento XML, un elemento XML, una instrucción de procesamiento XML, un comentario XML o una sección CDATA.

Anotación

Esta especificación contiene lo suficiente de una descripción de XML para describir el comportamiento del lenguaje Visual Basic. Se puede obtener más información en [http://www.w3.org/TR/REC-xml/](http://vstfdevdiv:8080/web/wi.aspx).

XMLLiteralExpression ::=  
 XMLDocument |  
 XMLElement | XMLProcessingInstruction |  
 XMLComment |  
 XMLCDATASection

### Reglas léxicas

Las expresiones literales XML se interpretan como reglas léxicas de XML en lugar de como reglas léxicas del código normal de Visual Basic. Los dos conjuntos de reglas difieren, en general, en lo siguiente:

El espacio en blanco tiene significado en XML. Como resultado, la gramática de las expresiones literales XML enuncia explícitamente los lugares donde se permiten los espacios en blanco. Los espacios en blanco no se conservan, excepto cuando aparecen en el contexto de datos de caracteres dentro de un elemento. Por ejemplo:

' The following element preserves no whitespace  
Dim e1 = \_  
 <customer>  
 <name>Bob</>  
 </>  
  
' The following element preserves all of the whitespace  
Dim e2 = \_  
 <customer>  
 Bob  
 </>

El espacio en blanco de final de línea en XML se normaliza según la especificación XML.

XML distingue entre mayúsculas y minúsculas. Las palabras clave deben coincidir exactamente en mayúsculas y minúsculas, o se producirá un error en tiempo de compilación.

Nota   Los terminadores de línea no se consideran espacios en blanco en XML. Como resultado, no se precisan caracteres de continuación de línea en expresiones literales de XML.

XML no acepta caracteres de ancho completo. Si se utilizan, se producirá un error en tiempo de compilación.

XMLCharacter ::=  
 < Unicode tab character (0x0009) > |  
 < Unicode linefeed character (0x000A) > |  
 < Unicode carriage return character (0x000D) > |  
 < Unicode characters 0x0020 – 0xD7FF > |  
 < Unicode characters 0xE000 – 0xFFFD > |  
 < Unicode characters 0x10000 – 0x10FFFF >

XMLString ::= XMLCharacter+

XMLWhitespace ::= XMLWhitespaceCharacter+

XMLWhitespaceCharacter ::=  
 < Unicode carriage return character (0x000D) > |  
 < Unicode linefeed character (0x000A) > |  
 < Unicode space character (0x0020) > |  
 < Unicode tab character (0x0009) >

XMLNameCharacter ::= XMLLetter | XMLDigit | . | - | \_ | : | XMLCombiningCharacter | XMLExtender

XMLNameStartCharacter ::= XMLLetter | \_ | :

XMLName ::= XMLNameStartCharacter [ XMLNameCharacter+ ]

XMLLetter ::=   
 < Unicode character as defined in the Letter production of the XML 1.0 specification >

XMLDigit ::=  
 < Unicode character as defined in the Digit production of the XML 1.0 specification >

XMLCombiningCharacter ::=  
 < Unicode character as defined in the CombiningChar production of the XML 1.0 specification >

XMLExtender ::=  
 < Unicode character as defined in the Extender production of the XML 1.0 specification >

### Expresiones incrustadas

Las expresiones literales XML pueden incluir expresiones incrustadas. Una expresión incrustada es una expresión de Visual Basic que se evalúa y se usa para completar uno o más valores en la ubicación de la expresión incrustada. Por ejemplo, el siguiente código sitúa la cadena John Smith como el valor del elemento XML:

Dim name as String = "John Smith"  
Dim element As System.Xml.Linq.XElement = <customer><%= name %></customer>

Las expresiones pueden estar incrustadas en una serie de contextos. Por ejemplo, el siguiente ejemplo de código produce un elemento denominado customer:

Dim name As String = "customer"  
Dim element As System.Xml.Linq.XElement = <<%= name %>>John Smith</>

Cada contexto en el que se puede usar una expresión incrustada especifica los tipos que se aceptarán. Dentro del contexto de la parte de expresión de una expresión incrustada, se siguen aplicando las reglas léxicas normales para el código de Visual Basic, así que, por ejemplo, se deben usar las continuaciones de línea:

' Visual Basic expression uses line continuation, XML does not  
Dim element As System.Xml.Linq.XElement = \_  
 <<%= name & \_  
 name %>>John   
 Smith</>

XMLEmbeddedExpression ::=  
 < % = [ LineTerminator ] Expression [ LineTerminator ] % >

### Documentos XML

Un documento XML da como resultado un valor de tipo System.Xml.Linq.XDocument. A diferencia de la especificación XML 1.0, los documentos XML en expresiones literales XML se requieren para especificar el prólogo del documento; las expresiones literales XML sin prólogo de documento se interpretan como su entidad individual. Por ejemplo:

Dim doc As System.Xml.Linq.XDocument = \_  
 <?xml version="1.0"?>  
 <?instruction?>  
 <customer>Bob</>  
  
Dim pi As System.Xml.Linq.XProcessingInstruction = \_  
 <?instruction?>

Un documento XML puede contener expresiones incrustadas cuyo tipo puede ser cualquiera; en tiempo de ejecución, sin embargo, el objeto debe satisfacer los requisitos del constructor XDocument o se producirá un error en tiempo de ejecución.

A diferencia del XML normal, las expresiones de documento XML no admiten DTD (Declaraciones de tipos de documento). Además, el atributo de codificación, si se proporciona, se pasará por alto, ya que la codificación de expresión literal XML siempre es la misma que la codificación del archivo de origen.

Anotación

Aunque el atributo de codificación se ignore, sigue siendo válido para mantener la capacidad de incluir documentos de Xml 1.0 válidos en el código fuente.

XMLDocument ::=  
 XMLDocumentPrologue [ XMLMisc+ ] XMLDocumentBody [ XMLMisc+ ]

XMLDocumentPrologue ::=  
 < ? xml XMLVersion [ XMLEncoding ] [ XMLStandalone ] [ XMLWhitespace ] ? >

XMLVersion ::=  
 XMLWhitespace version [ XMLWhitespace ] = [ XMLWhitespace ] XMLVersionNumberValue

XMLVersionNumberValue ::=   
 SingleQuoteCharacter 1 . 0 SingleQuoteCharacter |  
 DoubleQuoteCharacter 1 . 0 DoubleQuoteCharacter

XMLEncoding ::=  
 XMLWhitespace encoding [ XMLWhitespace ] = [ XMLWhitespace ] XMLEncodingNameValue

XMLEncodingNameValue ::=   
 SingleQuoteCharacter XMLEncodingName SingleQuoteCharacter |  
 DoubleQuoteCharacter XMLEncodingName DoubleQuoteCharacter

XMLEncodingName ::= XMLLatinAlphaCharacter [ XMLEncodingNameCharacter+ ]

XMLEncodingNameCharacter ::=  
 XMLUnderscoreCharacter |  
 XMLLatinAlphaCharacter |  
 XMLNumericCharacter |  
 XMLPeriodCharacter |  
 XMLDashCharacter

XMLLatinAlphaCharacter ::=  
 < Unicode Latin alphabetic character (0x0041-0x005a, 0x0061-0x007a) >

XMLNumericCharacter ::= < Unicode digit character (0x0030-0x0039) >

XMLHexNumericCharacter ::=  
 XMLNumericCharacter |  
 < Unicode Latin hex alphabetic character (0x0041-0x0046, 0x0061-0x0066) >

XMLPeriodCharacter ::= < Unicode period character (0x002e) >

XMLUnderscoreCharacter ::= < Unicode underscore character (0x005f) >

XMLDashCharacter ::= < Unicode dash character (0x002d) >

XMLStandalone ::=  
 XMLWhitespace standalone [ XMLWhitespace ] = [ XMLWhitespace ] XMLYesNoValue

XMLYesNoValue ::=   
 SingleQuoteCharacter XMLYesNo SingleQuoteCharacter |  
 DoubleQuoteCharacter XMLYesNo DoubleQuoteCharacter

XMLYesNo ::= yes | no

XMLMisc ::=  
 XMLComment |  
 XMLProcessingInstruction |  
 XMLWhitespace

XMLDocumentBody ::= XMLElement | XMLEmbeddedExpression

### Elementos XML

Un elemento XML da como resultado un valor de tipo System.Xml.Linq.XElement. A diferencia del XML normal, los elementos XML pueden omitir el nombre en la etiqueta de cierre y el elemento más recientemente anidado se cerrará. Por ejemplo:

Dim name = <name>Bob</>

Las declaraciones de atributo en un elemento XML dan como resultado valores de tipo System.Xml.Linq.XAttribute. Los valores de atributo se normalizan según la especificación XML. Cuando el valor de un atributo es Nothing, no se creará, de forma que no tendrá que comprobarse si la expresión de valor de atributo es Nothing. Por ejemplo:

Dim expr = Nothing  
  
' Throws null argument exception  
Dim direct = New System.Xml.Linq.XElement( \_  
 "Name", \_  
 New System.Xml.Linq.XAttribute("Length", expr))  
  
' Doesn't throw exception, the result is <Name/>  
Dim literal = <Name Length=<%= expr %>/>

Los elementos y atributos XML pueden contener expresiones anidadas en los lugares siguientes:

El nombre del elemento, en cuyo caso la expresión incrustada debe ser un valor de un tipo implícitamente convertible en System.Xml.Linq.XName. Por ejemplo:

Dim name = <<%= "name" %>>Bob</>

El nombre de un atributo del elemento, en cuyo caso la expresión incrustada debe ser un valor de un tipo implícitamente convertible en System.Xml.Linq.XName. Por ejemplo:

Dim name = <name <%= "length" %>="3">Bob</>

El valor de un atributo del elemento, en cuyo caso la expresión incrustada puede ser un valor de cualquier tipo. Por ejemplo:

Dim name = <name length=<%= 3 %>>Bob</>

Un atributo del elemento, en cuyo caso la expresión incrustada puede ser un valor de cualquier tipo. Por ejemplo:

Dim name = <name <%= new XAttribute("length", 3) %>>Bob</>

El contenido del elemento, en cuyo caso la expresión incrustada puede ser un valor de cualquier tipo. Por ejemplo:

Dim name = <name><%= "Bob" %></>

Si el tipo de la expresión incrustada es Object(), se pasará la matriz como paramarray al constructor XElement.

XMLElement ::=  
 XMLEmptyElement |  
 XMLElementStart XMLContent XMLElementEnd

XMLEmptyElement ::=  
 < XMLQualifiedNameOrExpression [ XMLAttribute+ ] [ XMLWhitepace ] / >

XMLElementStart ::=  
 < XMLQualifiedNameOrExpression [ XMLAttribute+ ] [ XMLWhitespace ] >

XMLElementEnd ::=  
 < / > |  
 < / XMLQualifiedName [ XMLWhitespace ] >

XMLContent ::=  
 [ XMLCharacterData ] [ XMLNestedContent [ XMLCharacterData ] ]+

XMLCharacterData ::=  
 < Any XMLCharacterDataString that does not contain the string "]]>" >

XMLCharacterDataString ::=  
 < Any Unicode character except < or & >+

XMLNestedContent ::=  
 XMLElement |  
 XMLReference |  
 XMLCDATASection |  
 XMLProcessingInstruction |  
 XMLComment |  
 XMLEmbeddedExpression

XMLAttribute ::=  
 XMLWhitespace XMLAttributeName [ XMLWhitespace ] = [ XMLWhitespace ] XMLAttributeValue |  
 XMLWhitespace XMLEmbeddedExpression

XMLAttributeName ::=  
 XMLQualifiedNameOrExpression |  
 XMLNamespaceAttributeName

XMLAttributeValue ::=  
 DoubleQuoteCharacter [ XMLAttributeDoubleQuoteValueCharacter+ ] DoubleQuoteCharacter |  
 SingleQuoteCharacter [ XMLAttributeSingleQuoteValueCharacter+ ] SingleQuoteCharacter |  
 XMLEmbeddedExpression

XMLAttributeDoubleQuoteValueCharacter ::=   
 < Any XMLCharacter except <, &, or DoubleQuoteCharacter > |  
 XMLReference

XMLAttributeSingleQuoteValueCharacter ::=  
 < Any XMLCharacter except <, &, or SingleQuoteCharacter > |  
 XMLReference

XMLReference ::= XMLEntityReference | XMLCharacterReference

XMLEntityReference ::=  
 & XMLEntityName ;

XMLEntityName ::= lt | gt | amp | apos | quot

XMLCharacterReference ::=  
 & # XMLNumericCharacter+ ; |  
 & # x XMLHexNumericCharacter+ ;

### Espacios de nombres XML

Los elementos XML pueden contener declaraciones de espacios de nombres XML, como se define en la especificación de espacios de nombres XML 1.0. Es preciso cumplir las restricciones de definición de los espacios de nombres xml y xmlns, y producirán errores en tiempo de compilación. Las declaraciones de espacios de nombres XML no pueden tener una expresión incrustada para su valor; el valor proporcionado debe ser un literal de cadena no vacía. Por ejemplo:

' Declares a valid namespace  
Dim customer = <db:customer xmlns:db="http://example.org/database">Bob</>  
  
' Error: xmlns cannot be re-defined  
Dim bad1 = <elem xmlns:xmlns="http://example.org/namespace"/>  
  
' Error: cannot have an embedded expression  
Dim bad2 = <elem xmlns:db=<%= "http://example.org/database" %>>Bob</>

Anotación

Esta especificación contiene lo suficiente de una descripción del espacio de nombres XML para describir el comportamiento del lenguaje Visual Basic. Se puede obtener más información sobres espacios de nombres XML en [http://www.w3.org/TR/REC-xml-names/](http://vstfdevdiv:8080/web/wi.aspx).

Los nombres de atributos y elementos XML deben estar calificados con nombres del espacio de nombres. Los espacios de nombres se enlazan como en XML normal, con la excepción de que cualquier importación de espacio de nombres declarada en el nivel de archivos se considera declarada en un contexto que encierra la declaración, que está encerrada en alguna importación de espacio de nombres declarada por el entorno de compilación. Si no se encuentra un nombre de espacio de nombres, se produce un error en tiempo de compilación. Por ejemplo:

Imports System.Xml.Linq  
Imports <xmlns:db="http://example.org/database">  
  
Module Test  
 Sub Main()  
 ' Binds to the imported namespace above.  
 Dim c1 = <db:customer>Bob</>  
  
 ' Binds to the namespace declaration in the element  
 Dim c2 = \_  
 <db:customer xmlns:db="http://example.org/database-other">Mary</>  
  
 ' Binds to the inner namespace declaration  
 Dim c3 = \_  
 <database xmlns:db="http://example.org/database-one">  
 <db:customer xmlns:db="http://example.org/database-two">Joe</>  
 </>  
  
 ' Error: namespace db2 cannot be found  
 Dim c4 = \_  
 <db2:customer>Jim</>  
 End Sub  
End Module

Los espacios de nombres XML declarados en un elemento no se aplican a los literales XML dentro de expresiones incrustadas. Por ejemplo:

' Error: Namespace prefix 'db' is not declared  
Dim customer = \_  
 <db:customer xmlns:db="http://example.org/database">  
 <%= <db:customer>Bob</> %>  
 </>

Anotación

Esto se debe a que la expresión incrustada puede ser cualquier cosa, incluida una llamada a función. Si la llamada a la función contenía una expresión de literal XML, no está claro si los programadores esperarán que el espacio de nombres XML se aplique o ignore.

XMLNamespaceAttributeName ::=  
 XMLPrefixedNamespaceAttributeName |  
 XMLDefaultNamespaceAttributeName

XMLPrefixedNamespaceAttributeName ::=  
 xmlns : XMLNamespaceName

XMLDefaultNamespaceAttributeName ::=  
 xmlns

XMLNamespaceName ::= XMLNamespaceNameStartCharacter [ XMLNamespaceNameCharacter+ ]

XMLNamespaceNameStartCharacter ::=  
 < Any XMLNameCharacter except : >

XMLNamespaceNameCharacter ::= XMLLetter | \_

XMLQualifiedNameOrExpression ::= XMLQualifiedName | XMLEmbeddedExpression

XMLQualifiedName ::=  
 XMLPrefixedName |  
 XMLUnprefixedName

XMLPrefixedName ::= XMLNamespaceName : XMLNamespaceName

XMLUnprefixedName ::= XMLNamespaceName

### Instrucciones de procesamiento XML

Una instrucción de procesamiento de XML da como resultado un valor de tipo System.Xml.Linq.XProcessingInstruction. Las instrucciones de procesamiento XML no pueden contener expresiones incrustadas, porque son sintaxis válida dentro de la instrucción de procesamiento.

XMLProcessingInstruction ::=  
 < ? XMLProcessingTarget [ XMLWhitespace [ XMLProcessingValue ] ] ? >

XMLProcessingTarget ::=  
 < Any XMLName except a casing permutation of the string "xml" >

XMLProcessingValue ::=  
 < Any XMLString that does not contain the string "?>" >

### Comentarios XML

Un comentario XML da como resultado un valor de tipo System.Xml.Linq.XComment. Los comentarios XML no pueden contener expresiones incrustadas, porque son sintaxis válida dentro del comentario.

XMLComment ::=  
 < ! - - [ XMLCommentCharacter+ ] - - >

XMLCommentCharacter ::=  
 < Any XMLCharacter except dash (0x002D) > |  
 - < Any XMLCharacter except dash (0x002D) >

### Secciones CDATA

Una sección CDATA da como resultado un valor de tipo System.Xml.Linq.XCData. Las secciones CDATA no pueden contener expresiones incrustadas, porque son sintaxis válida dentro de la sección.

XMLCDATASection ::=  
 < ! [ CDATA [ [ XMLCDATASectionString ] ] ] >

XMLCDATASectionString ::=  
 < Any XMLString that does not contain the string "]]>" >

## Expresiones de acceso a miembros XML

Una expresión de acceso a miembros XML tiene acceso a los miembros de un valor XML. Hay tres tipos de expresión de acceso a miembros XML:

Acceso de elemento, donde un nombre XML va después de un punto. Por ejemplo:

Dim customer = \_  
 <customer>  
 <name>Bob</>  
 </>  
Dim customerName = customer.<name>.Value

El acceso de elemento se asigna a una función:

Function Elements(name As System.Xml.Linq.XName) As \_  
 System.Collections.Generic.IEnumerable(Of \_  
 System.Xml.Linq.XNode)

Así, el ejemplo anterior es equivalente a:

Dim customerName = customer.Elements("name").Value

Acceso de atributo, donde un identificador de Visual Basic va después de un punto y signo de arroba, o un nombre XML va después de un punto y un signo de arroba. Por ejemplo:

Dim customer = <customer age="30"/>  
Dim customerAge = customer.@age

El acceso de atributo se asigna a una función:

Function AttributeValue(name As System.Xml.Linq.XName) as String

Así, el ejemplo anterior es equivalente a:

Dim customerAge = customer.AttributeValue("age")

Anotación

El método de extensión AttributeValue (así como la propiedad de extensión relacionada Value) no está definido en ningún ensamblado. Si se necesitan métodos de extensión, se definen automáticamente en el ensamblado que se está produciendo.

Acceso de descendientes, donde un nombre XML va después de tres puntos. Por ejemplo:

Dim company = \_  
 <company>  
 <customers>  
 <customer>Bob</>  
 <customer>Mary</>  
 <customer>Joe</>  
 </>  
 </>  
Dim customers = company...<customer>

El acceso de descendientes se asigna a una función:

Function Descendents(name As System.Xml.Linq.XName) As \_  
 System.Collections.Generic.IEnumerable(Of \_  
 System.Xml.Linq.XElement)

Así, el ejemplo anterior es equivalente a:

Dim customers = company.Descendants("customer")

La expresión base de una expresión de acceso a miembros XML debe ser un valor y debe ser del tipo:

Si es un elemento o acceso de descendientes, System.Xml.Linq.XContainer o un tipo derivado o System.Collections.Generic.IEnumerable(Of T) o un tipo derivado, donde T es System.Xml.Linq.XContainer o un tipo derivado.

Si es un acceso de atributo, System.Xml.Linq.XElement o un tipo derivado o System.Collections.Generic.IEnumerable(Of T) o un tipo derivado, donde T es System.Xml.Linq.XElement o un tipo derivado.

Los nombres en las expresiones de acceso a miembros XML no pueden estar vacíos. Pueden estar calificadas con el espacio de nombres, usando cualquier espacio de nombres definido por importaciones. Por ejemplo:

Imports <xmlns:db="http://example.org/database">  
  
Module Test  
 Sub Main()  
 Dim customer = \_  
 <db:customer>  
 <db:name>Bob</>  
 </>  
 Dim name = customer.<db:name>  
 End Sub  
End Module

No se permiten espacios en blanco después de puntos en una expresión de acceso a miembros XML o entre corchetes angulares y el nombre. Por ejemplo:

Dim customer = \_  
 <customer age="30">  
 <name>Bob</>  
 </>  
' All the following are error cases  
Dim age = customer.@ age  
Dim name = customer.< name >  
Dim names = customer...< name >

Si los tipos del espacio de nombres System.Xml.Linq no están disponibles, la expresión de acceso a miembros XML dará error en tiempo de compilación.

XMLMemberAccessExpression ::=  
 Expression . [ LineTerminator ] < XMLQualifiedName > |  
 Expression . [ LineTerminator ] @ [ LineTerminator ] < XMLQualifiedName > |  
 Expression . [ LineTerminator ] @ [ LineTerminator ] IdentifierOrKeyword |  
 Expression . . . [ LineTerminator ] < XMLQualifiedName >

## Operador Await

El operador await está relacionado con los métodos asincrónicos, que se describen en la sección 10.1.3.

Await es una palabra reservada si el método o la expresión lambda envolvente inmediato donde aparece tiene un modificador Async, y si Await aparece después de dicho modificador Async; de lo contrario, no es reservada. Tampoco es reservada en las directivas de preprocesador. El operador await solamente se permite en el cuerpo de un método o de expresiones lambda donde es una palabra reservada. Dentro del método o la expresión lambda envolvente inmediato, una expresión await no puede aparecer en el cuerpo de un bloque Catch o Finally, ni tampoco en el cuerpo de una instrucción SyncLock, ni dentro de una expresión de consulta.

El operador await toma una única expresión que debe clasificarse como valor y cuyo tipo *admita await* o sea Object. Si su tipo es Object, todo el procesamiento se aplaza hasta el tiempo de ejecución. Se dice que un tipo C admite await si se cumplen las siguientes condiciones:

1. C contiene un método de instancia o de extensión accesible denominado GetAwaiter que no tiene argumentos y que devuelve algún tipo E;
2. E contiene una propiedad de extensión o de instancia legible denominada IsCompleted que no toma ningún argumento y tiene el tipo Boolean;
3. E contiene un método de instancia o de extensión accesible denominado GetResult que no toma ningún argumento;
4. E implementa System.Runtime.CompilerServices.INotifyCompletion o ICriticalNotifyCompletion.

Si GetResult era Sub, la expresión await se clasifica como vacía. De lo contrario, la expresión await se clasifica como un valor y su tipo es el tipo devuelto del método GetResult.

A continuación se ofrece un ejemplo de una clase que admite await:

Class MyTask(Of T)  
 Function GetAwaiter() As MyTaskAwaiter(Of T)  
 Return New MyTaskAwaiter With {.m\_Task = Me}  
 End Function

...  
End Class

Structure MyTaskAwaiter(Of T)  
 Implements INotifyCompletion  
  
 Friend m\_Task As MyTask(Of T)

ReadOnly Property IsCompleted As Boolean  
 Get  
 Return m\_Task.IsCompleted  
 End Get  
 End Property

Sub OnCompleted(r As Action) Implements INotifyCompletion.OnCompleted  
 ' r is the "resumptionDelegate"  
 Dim sc = SynchronizationContext.Current  
 If sc Is Nothing Then  
 m\_Task.ContinueWith(Sub() r())  
 Else  
 m\_Task.ContinueWith(Sub() sc.Post(Sub() r(), Nothing))  
 End If  
 End Sub

Function GetResult() As T  
 If m\_Task.IsCanceled Then Throw New TaskCanceledException(m\_Task)  
 If m\_Task.IsFaulted Then Throw m\_Task.Exception.InnerException  
 Return m\_Task.Result  
 End Function  
End Structure

Anotación

Se recomienda a los autores de la biblioteca que sigan el patrón de invocar al delegado de continuación en el mismo SynchronizationContext en el que se invocó OnCompleted. Asimismo, el delegado de reanudación no debería ejecutarse sincrónicamente en el método OnCompleted, ya que esto puede provocar un desbordamiento de pila; en su lugar, el delegado debería ponerse en cola para su ejecución posterior.

Cuando el flujo de control llega a un operador Await, el comportamiento es el siguiente.

1. Se invoca el método GetAwaiter del operando await. El resultado de esta invocación se denomina *awaiter*.
2. Se recupera la propiedad IsCompleted del elemento awaiter. Si el resultado es true:
   1. Se invoca el método GetResult del elemento awaiter. Si GetResult era una función, el valor de la expresión await es el valor devuelto de esta función.
3. Si la propiedad IsCompleted no es true:
   1. Se invoca ICriticalNotifyCompletion.UnsafeOnCompleted en el elemento awaiter (si el tipo E de este implementa ICriticalNotifyCompletion) o INotifyCompletion.OnCompleted (en los demás casos). En ambos casos se pasa un *delegado de reanudación* asociado a la instancia actual del método asincrónico.
   2. El punto de control de la instancia actual del método asincrónico se suspende y el flujo de control se reanuda en el *llamador actual* (definido en la sección 10.1.3).
   3. Si se invoca al delegado de reanudación más adelante,
      1. en primer lugar este restaura System.Threading.Thread.CurrentThread.ExecutionContext al estado que tenía en el momento en que se llamó a OnCompleted
      2. y, a continuación, reanuda el flujo de control en el punto de control de la instancia del método asincrónico (vea la sección 10.1.3).
      3. donde llama al método GetResult del elemento awaiter, como en el apartado 2.1 anterior.

Si el operando await tiene el tipo Object, este comportamiento se aplaza hasta el tiempo de ejecución:

* Para realizar el paso 1 se llama a GetAwaiter() sin argumentos; así, puede enlazarse en tiempo de ejecución a métodos de instancia que toman parámetros opcionales.
* Para realizar el paso 2 se recupera la propiedad IsCompleted() sin argumentos y se prueba a realizar una conversión intrínseca en Boolean.
* Para realizar el paso 3.1 se prueba a realizar TryCast(awaiter, ICriticalNotifyCompletion) y, si esto da error, se prueba con DirectCast(awaiter, INotifyCompletion).

El delegado de reanudación pasado en la sección 3.1 solamente se puede invocar una vez. Si se invoca más de una vez, el comportamiento queda sin definir.

AwaitOperatorExpression ::= Await Expression

# Comentarios de documentación

Los comentarios de documentación son comentarios con formato especial en el código fuente que se pueden analizar para generar documentación sobre el código al que se adjuntan. El formato básico para estos comentarios es XML. Cuando se compila código con comentarios de documentación, el compilador puede emitir opcionalmente un archivo XML que represente la suma total de los comentarios de documentación del código fuente. El archivo XML se puede usar con otras herramientas para generar documentación impresa o en línea.

En este capítulo se describen los comentarios de documentación y las etiquetas XML recomendadas para usar con los comentarios.

## Formato de comentario de documentación

Los comentarios de documento son comentarios especiales que comienzan con tres comillas simples '''. Deben preceder inmediatamente al tipo (como una clase, un delegado o una interfaz) o al miembro de tipo (como un campo, un evento, una propiedad o un método) que comentan. Un comentario de documento en una declaración de método parcial se reemplazará por el comentario de documento en el método que proporciona su cuerpo, si hay alguno. Todos los comentarios de documento adyacentes se juntan para producir un único comentario de documento. Si hay un espacio en blanco seguido por los caracteres ''', el espacio en blanco no se incluye en la concatenación. Por ejemplo:

''' <remarks>  
''' Class <c>Point</c> models a point in a two-dimensional plane.  
''' </remarks>  
Public Class Point   
 ''' <remarks>  
 ''' Method <c>Draw</c> renders the point.  
 ''' </remarks>  
 Sub Draw()  
 End Sub  
End Class

Los comentarios de documentación deben estar correctamente formados en XML según [http://www.w3.org/TR/REC-xml](http://vstfdevdiv:8080/web/wi.aspx). Si el código XML no está bien formado, se generará un error de advertencia y el archivo de documentación incluirá un comentario en el que se mencionará el error encontrado.

Aunque los desarrolladores pueden crear su propio juego de etiquetas, existe un juego recomendado en la sección siguiente. Algunas de las etiquetas recomendadas tienen significados especiales:

La etiqueta <param> se utiliza para describir parámetros. El parámetro especificado para una etiqueta <param> debe existir y todos los parámetros del miembro de tipo deben describirse en el comentario de documentación. Si alguna de las condiciones no es verdadera, el compilador emite una advertencia.

El atributo cref se puede asociar a cualquier etiqueta para proporcionar una referencia a un elemento de código. El elemento de código debe existir; en tiempo de compilación, el compilador reemplaza el nombre por la cadena de identificador que representa al miembro. Si el elemento de código no existe, el compilador emite una advertencia. Cuando se busca un nombre descrito en un atributo cref, el compilador respeta las instrucciones Imports que aparecen dentro del archivo de origen contenedor.

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.

Observe que el archivo de documentación no proporciona información completa sobre un tipo y los miembros, solo lo que contiene los comentarios de documento. Para obtener más 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.

## 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:

<c> Establece una fuente de estilo código para un texto

<code> Establece un o más líneas de código fuente o resultado de programa en una fuente de estilo código

<example> Indica un ejemplo

<exception> Identifica las excepciones que pueden iniciar un método

<include> Incluye un documento XML externo

<list> Crea una lista o una tabla

<para> Permite agregar una estructura al texto

<param> Describe un parámetro para un método o constructor

<paramref> Identifica una palabra como nombre de parámetro

<permission> Documenta la accesibilidad de seguridad de un miembro

<remarks> Describe un tipo

<returns> Describe el valor devuelto de un método

<see> Especifica un vínculo

<seealso> Genera una entrada de tipo Vea también

<summary> Describe un miembro de un tipo

<typeparam> Describe un parámetro de tipo

<value> Describe una propiedad

### <c>

Esta etiqueta especifica que un fragmento de texto dentro de una descripción debería usar una fuente como la utilizada en un bloque de código. (Para líneas de código real, use <code>).

Sintaxis:

<c>text to be set like code</c>

Ejemplo:

''' <remarks>  
 ''' Class <c>Point</c> models a point in a two-dimensional plane.  
 ''' </remarks>  
 Public Class Point   
 End Class

### <code>

Esta etiqueta especifica que una o más líneas de código fuente o resultado de programa debería usar una fuente de ancho fijo. (Para fragmentos de código pequeños, use <c>).

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>  
 ''' Dim p As Point = New Point(3,5)  
 ''' p.Translate(-1,3)  
 ''' </code>  
 ''' results in <c>p</c>'s having the value (2,8).  
 ''' </example>  
 ''' </summary>  
 Public Sub Translate(x As Integer, y As Integer)  
 Me.x += x  
 Me.y += y  
 End Sub

### <example>

Esta etiqueta permite que un código de ejemplo dentro de un comentario muestre cómo se puede usar un elemento. Generalmente, esto también supone la utilización de la etiqueta <code>.

Sintaxis:

<example>description</example>

Ejemplo:

Vea <code> para obtener un ejemplo.

### <exception>

Esta etiqueta proporciona un medio para documentar las excepciones que puede producir un método.

Sintaxis:

<exception cref="member">description</exception>

Ejemplo:

Public Module DataBaseOperations  
 ''' <exception cref="MasterFileFormatCorruptException"></exception>  
 ''' <exception cref="MasterFileLockedOpenException"></exception>  
 Public Sub ReadRecord(flag As Integer)  
 If Flag = 1 Then  
 Throw New MasterFileFormatCorruptException()  
 ElseIf Flag = 2 Then  
 Throw New MasterFileLockedOpenException()  
 End If  
 ' ...  
 End Sub  
 End Module

### <include>

Esta etiqueta se usa para incluir información de un documento externo correctamente formado en XML. Se aplica una expresión XPath al documento XML para especificar qué 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">

Ejemplo:

Si el código fuente contiene una declaración como la siguiente:

''' <include file="docs.xml" path="extra/class[@name="IntList"]/\*" />

y el archivo externo docs.xml tuviera el siguiente contenido

<?xml version="1.0"?>  
 <extra>  
 <class name="IntList">  
 <summary>  
 Contains a list of integers.  
 </summary>  
 </class>  
 <class name="StringList">  
 <summary>  
 Contains a list of strings.  
 </summary>  
 </class>  
 </extra>

entonces la documentación resultante sería la misma que si el código fuente contuviera:

''' <summary>  
 ''' Contains a list of integers.  
 ''' </summary>

### <list>

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>

Ejemplo:

Public Class TestClass  
 ''' <remarks>  
 ''' Here is an example of a bulleted list:  
 ''' <list type="bullet">  
 ''' <item>  
 ''' <description>Item 1.</description>  
 ''' </item>  
 ''' <item>  
 ''' <description>Item 2.</description>  
 ''' </item>  
 ''' </list>  
 ''' </remarks>  
 Public Shared Sub Main()  
 End Sub  
 End Class

### <para>

Este etiqueta debe utilizarse dentro de otras, como <remarks> o <returns> y permite agregar la estructura al texto.

Sintaxis:

<para>content</para>

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 Shared Sub Main()  
 End Sub

### <param>

Esta etiqueta describe un parámetro para un método, constructor o propiedad indizada.

Sintaxis:

<param name="name">description</param>

Ejemplo:

''' <summary>  
 ''' This method changes the point's location to the given  
 ''' coordinates.  
 ''' </summary>  
 ''' <param name="x"><c>x</c> is the new x-coordinate.</param>  
 ''' <param name="y"><c>y</c> is the new y-coordinate.</param>  
 Public Sub Move(x As Integer, y As Integer)  
 Me.x = x  
 Me.y = y  
 End Sub

### <paramref>

Esta etiqueta indica 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"/>

Ejemplo:

''' <summary>  
 ''' This constructor initializes the new Point to  
 ''' (<paramref name="x"/>,<paramref name="y"/>).  
 ''' </summary>  
 ''' <param name="x"><c>x</c> is the new Point's x-coordinate.</param>  
 ''' <param name="y"><c>y</c> is the new Point's y-coordinate.</param>  
 Public Sub New(x As Integer, y As Integer)  
 Me.x = x  
 Me.y = y  
 End Sub

### <permission>

Esta etiqueta documenta la accesibilidad de seguridad de un miembro

Sintaxis:

<permission cref="member">description</permission>

Ejemplo:

''' <permission cref="System.Security.PermissionSet">Everyone can  
 ''' access this method.</permission>  
 Public Shared Sub Test()  
 End Sub

### <remarks>

Esta etiqueta especifica información general sobre un tipo. (Utilice <summary> para describir los miembros de un tipo).

Sintaxis:

<remarks>description</remarks>

Ejemplo:

''' <remarks>  
 ''' Class <c>Point</c> models a point in a two-dimensional plane.  
 ''' </remarks>  
 Public Class Point   
 End Class

### <returns>

Esta etiqueta describe el valor devuelto de un método.

Sintaxis:

<returns>description</returns>

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, training, or embedded whitespace.  
 ''' </returns>  
 Public Overrides Function ToString() As String  
 Return "(" & x & "," & y & ")"  
 End Sub

### <see>

Esta etiqueta permite especificar un vínculo dentro del texto. (Utilice <seealso> para indicar el texto que debe aparecer en una sección Vea también).

Sintaxis:

<see cref="member"/>

Ejemplo:

''' <summary>  
 ''' This method changes the point's location to the given  
 ''' coordinates.  
 ''' </summary>  
 ''' <see cref="Translate"/>  
 Public Sub Move(x As Integer, y As Integer)  
 Me.x = x  
 Me.y = y  
 End Sub  
  
 ''' <summary>  
 ''' This method changes the point's location by the given x- and  
 ''' y-offsets.  
 ''' </summary>  
 ''' <see cref="Move"/>  
 Public Sub Translate(x As Integer, y As Integer)  
 Me.x += x  
 Me.y += y  
 End Sub

### <seealso>

Esta etiqueta generar una entrada para la sección Vea también. (Utilice <see> para especificar un vínculo desde dentro del texto).

Sintaxis:

<seealso cref="member"/>

Ejemplo:

''' <summary>  
 ''' This method determines whether two Points have the same location.  
 ''' </summary>  
 ''' <seealso cref="operator=="/>  
 ''' <seealso cref="operator!="/>  
 Public Overrides Function Equals(o As Object) As Boolean  
 ' ...  
 End Function

### <summary>

Esta etiqueta describe un miembro de tipo. Utilice <remarks> para describir el propio tipo).

Sintaxis:

<summary>description</summary>

Ejemplo:

''' <summary>  
 ''' This constructor initializes the new Point to (0,0).  
 ''' </summary>  
 Public Sub New()  
 Me.New(0,0)  
 End Sub

### <typeparam>

Esta etiqueta describe un tipo de parámetro.

Sintaxis:

<typeparam name="name">description</typeparam>

Ejemplo:

''' <typeparam name="T">  
 ''' The base item type. Must implement IComparable.  
 ''' </typeparam>  
 Public Class ItemManager(Of T As IComparable)  
 End Class

### <value>

Esta etiqueta describe una propiedad.

Sintaxis:

<value>property description</value>

Ejemplo:

''' <value>  
 ''' Property <c>X</c> represents the point's x-coordinate.  
 ''' </value>  
 Public Property X() As Integer  
 Get  
 Return \_x  
 End Get  
 Set (Value As Integer)  
 \_x = Value  
 End Set  
 End Property

## Cadenas id.

Cuando se genera el archivo de documentación, el compilador genera una cadena de identificador para cada elemento del código fuente etiquetado con un comentario de documentación que lo identifica de manera única. Esta cadena pueden usarla otras herramientas externas para identificar qué elemento de un ensamblado compilado corresponde al comentario del documento.

Las cadenas de identificador se generan del modo siguiente:

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 miembros siguientes, seguido del carácter correspondiente entre paréntesis: eventos (E), campos (F), métodos incluidos los constructores y los operadores (M), espacios de nombres (N), propiedades (P) y tipos (T). Un signo de exclamación (!) indica un error durante la generación de la cadena de id. y el resto de la cadena ofrece información sobre el error.

La segunda parte de la cadena es el nombre completo del elemento, empezando en el espacio de nombres global. El nombre del elemento, los tipos contenedores y el espacio de nombres se separan mediante puntos. Si el propio nombre del elemento contiene puntos, se reemplazan por el signo #. (Se supone que ningún elemento tiene este carácter en su nombre). El nombre de un tipo con parámetros de tipo termina con una comilla simple invertida (`) seguida de un número que representa la cantidad de los parámetros de tipo en el tipo. Es importante recordar que, como los tipos anidados tienen acceso a los parámetros de tipo de los tipos que los contienen, los tipos anidados contienen implícitamente los parámetros de tipo de los tipos contenedores y esos tipos se cuentan en el total de parámetros en este caso.

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 una firma CLI, como sigue: los argumentos se representan por su nombre completo. Por ejemplo, Integer se convierte en System.Int32, String se convierte en System.String, Object se convierte en System.Object, etc. Los argumentos que contienen el modificador ByRef llevan un signo '@' tras su nombre de tipo. Los argumentos que tienen el modificador ByVal, Optional o ParamArray no tienen ninguna notación especial. Los argumentos que son matrices se representan como [lowerbound:size, …, lowerbound:size], donde el número de comas es rank – 1, y los límites y tamaños inferiores de cada dimensión, si se conocen, se representan en formato decimal. Si no se especifica un límite ni un tamaño inferior, se omite. Si se omiten el límite y el tamaño inferior de una dimensión concreta, el signo ':' también se omite. Las matrices de matrices se representan con un "[]" por nivel.

### Ejemplos de cadena de identificador

En los siguientes ejemplos se muestra un fragmento de código de VB, junto con la cadena de identificador que se produce a partir de cada elemento de código fuente susceptible de tener un comentario de documentación:

Los tipos se representan con su nombre completo.

Enum Color  
 Red  
 Blue  
 Green  
End Enum  
  
Namespace Acme  
 Interface IProcess  
 End Interface  
  
 Structure ValueType  
 ...  
 End Structure  
  
 Class Widget  
 Public Class NestedClass  
 End Class  
  
 Public Interface IMenuItem  
 End Interface  
  
 Public Delegate Sub Del(i As Integer)  
  
 Public Enum Direction  
 North  
 South  
 East  
 West  
 End Enum  
 End Class  
End Namespace  
  
"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"

Los campos se representan con su nombre completo.

Namespace Acme  
 Structure ValueType  
 Private total As Integer  
 End Structure  
  
 Class Widget  
 Public Class NestedClass  
 Private value As Integer  
 End Class  
  
 Private message As String  
 Private Shared defaultColor As Color  
 Private Const PI As Double = 3.14159  
 Protected ReadOnly monthlyAverage As Double  
 Private array1() As Long  
 Private array2(,) As Widget  
 End Class  
End Namespace  
  
"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"

Constructores.

Namespace Acme  
 Class Widget  
 Shared Sub New()  
 End Sub  
  
 Public Sub New()  
 End Sub  
  
 Public Sub New(s As String)  
 End Sub  
 End Class  
End Namespace  
  
"M:Acme.Widget.#cctor"  
"M:Acme.Widget.#ctor"  
"M:Acme.Widget.#ctor(System.String)"

Métodos.

Namespace Acme  
 Structure ValueType  
 Public Sub M(i As Integer)  
 End Sub  
 End Structure  
  
 Class Widget  
 Public Class NestedClass  
 Public Sub M(i As Integer)  
 End Sub  
 End Class  
  
 Public Shared Sub M0()  
 End Sub  
  
 Public Sub M1(c As Char, ByRef f As Float, \_  
 ByRef v As ValueType)  
 End Sub  
  
 Public Sub M2(x1() As Short, x2(,) As Integer, \_  
 x3()() As Long)  
 End Sub  
  
 Public Sub M3(x3()() As Long, x4()(,,) As Widget)  
 End Sub  
  
 Public Sub M4(Optional i As Integer = 1)  
  
 Public Sub M5(ParamArray args() As Object)  
 End Sub  
 End Class  
End Namespace  
  
"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.Int32)”  
“M:Acme.Widget.M5(System.Object[])"

Propiedades.

Namespace Acme  
 Class Widget  
 Public Property Width() As Integer  
 Get  
 End Get  
 Set (Value As Integer)  
 End Set  
 End Property  
  
 Public Default Property Item(i As Integer) As Integer  
 Get  
 End Get  
 Set (Value As Integer)  
 End Set  
 End Property  
  
 Public Default Property Item(s As String, \_  
 i As Integer) As Integer  
 Get  
 End Get  
 Set (Value As Integer)  
 End Set  
 End Property  
 End Class  
End Namespace  
  
"P:Acme.Widget.Width"  
"P:Acme.Widget.Item(System.Int32)"  
"P:Acme.Widget.Item(System.String,System.Int32)"

Eventos

Namespace Acme  
 Class Widget  
 Public Event AnEvent As EventHandler  
 Public Event AnotherEvent()  
 End Class  
End Namespace  
  
"E:Acme.Widget.AnEvent"  
"E:Acme.Widget.AnotherEvent"

Operadores.

Namespace Acme  
 Class Widget  
 Public Shared Operator +(x As Widget) As Widget  
 End Operator  
  
 Public Shared Operator +(x1 As Widget, x2 As Widget) As Widget  
 End Operator  
 End Class  
End Namespace  
  
"M:Acme.Widget.op\_UnaryPlus(Acme.Widget)"  
"M:Acme.Widget.op\_Addition(Acme.Widget,Acme.Widget)"

Los operadores de conversión tienen un carácter final '~' seguido del tipo de valor devuelto.

Namespace Acme  
 Class Widget  
 Public Shared Narrowing Operator CType(x As Widget) As \_  
 Integer  
 End Operator  
  
 Public Shared Widening Operator CType(x As Widget) As Long  
 End Operator  
 End Class  
End Namespace  
  
"M:Acme.Widget.op\_Explicit(Acme.Widget)~System.Int32"  
"M:Acme.Widget.op\_Implicit(Acme.Widget)~System.Int64"

## Ejemplo de comentario de documentación

En el ejemplo siguiente se muestra el código fuente de una clase Point:

Namespace Graphics  
 ''' <remarks>  
 ''' Class <c>Point</c> models a point in a two-dimensional  
 ''' plane.  
 ''' </remarks>  
 Public Class Point  
 ''' <summary>  
 ''' Instance variable <c>x</c> represents the point's x-coordinate.  
 ''' </summary>  
 Private \_x As Integer  
  
 ''' <summary>  
 ''' Instance variable <c>y</c> represents the point's y-coordinate.  
 ''' </summary>  
 Private \_y As Integer  
  
 ''' <value>  
 ''' Property <c>X</c> represents the point's x-coordinate.  
 ''' </value>  
 Public Property X() As Integer  
 Get  
 Return \_x  
 End Get  
 Set(Value As Integer)  
 \_x = Value  
 End Set  
 End Property  
  
 ''' <value>  
 ''' Property <c>Y</c> represents the point's y-coordinate.  
 ''' </value>  
 Public Property Y() As Integer  
 Get  
 Return \_y  
 End Get  
 Set(Value As Integer)  
 \_y = Value  
 End Set  
 End Property  
  
 ''' <summary>  
 ''' This constructor initializes the new Point to (0,0).  
 ''' </summary>  
 Public Sub New()  
 Me.New(0, 0)  
 End Sub  
  
 ''' <summary>  
 ''' This constructor initializes the new Point to  
 ''' (<paramref name="x"/>,<paramref name="y"/>).  
 ''' </summary>  
 ''' <param name="x"><c>x</c> is the new Point's  
 ''' x-coordinate.</param>  
 ''' <param name="y"><c>y</c> is the new Point's  
 ''' y-coordinate.</param>  
 Public Sub New(x As Integer, y As Integer)  
 Me.X = x  
 Me.Y = y  
 End Sub  
  
 ''' <summary>  
 ''' This method changes the point's location to the given  
 ''' coordinates.  
 ''' </summary>  
 ''' <param name="x"><c>x</c> is the new x-coordinate.</param>  
 ''' <param name="y"><c>y</c> is the new y-coordinate.</param>  
 ''' <see cref="Translate"/>  
 Public Sub Move(x As Integer, y As Integer)  
 Me.X = x  
 Me.Y = y  
 End Sub  
  
 ''' <summary>  
 ''' This method changes the point's location by the given x- and  
 ''' y-offsets.  
 ''' <example>  
 ''' For example:  
 ''' <code>  
 ''' Dim p As Point = New Point(3, 5)  
 ''' p.Translate(-1, 3)  
 ''' </code>  
 ''' results in <c>p</c>'s having the value (2,8).  
 ''' </example>  
 ''' </summary>  
 ''' <param name="x"><c>x</c> is the relative x-offset.</param>  
 ''' <param name="y"><c>y</c> is the relative y-offset.</param>  
 ''' <see cref="Move"/>  
 Public Sub Translate(x As Integer, y As Integer)  
 Me.X += x  
 Me.Y += y  
 End Sub  
  
 ''' <summary>  
 ''' This method determines whether two Points have the same  
 ''' location.  
 ''' </summary>  
 ''' <param name="o"><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 op\_Equality"/>  
 ''' <seealso cref="Operator op\_Inequality"/>  
 Public Overrides Function Equals(o As Object) As Boolean  
 If o Is Nothing Then  
 Return False  
 End If  
 If o Is Me Then  
 Return True  
 End If  
 If Me.GetType() Is o.GetType() Then  
 Dim p As Point = CType(o, Point)  
 Return (X = p.X) AndAlso (Y = p.Y)  
 End If  
 Return False  
 End Function  
  
 ''' <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 Overrides Function ToString() As String  
 Return "(" & X & "," & Y & ")"  
 End Function  
  
 ''' <summary>  
 ''' This operator determines whether two Points have the  
 ''' same location.  
 ''' </summary>  
 ''' <param name="p1"><c>p1</c> is the first Point to be compared.  
 ''' </param>  
 ''' <param name="p2"><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="op\_Inequality"/>  
 Public Shared Operator =(p1 As Point, p2 As Point) As Boolean  
 If p1 Is Nothing OrElse p2 Is Nothing Then  
 Return False  
 End If  
 If p1.GetType() Is p2.GetType() Then  
 Return (p1.X = p2.X) AndAlso (p1.Y = p2.Y)  
 End If  
 Return False  
 End Operator  
  
 ''' <summary>  
 ''' This operator determines whether two Points have the  
 ''' same location.  
 ''' </summary>  
 ''' <param name="p1"><c>p1</c> is the first Point to be comapred.  
 ''' </param>  
 ''' <param name="p2"><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="op\_Equality"/>  
 Public Shared Operator <>(p1 As Point, p2 As Point) As Boolean  
 Return Not p1 = p2  
 End Operator  
  
 ''' <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 Shared Sub Main()  
 ' class test code goes here  
 End Sub  
 End Class  
End Namespace

Este es el resultado cuando se proporciona el mismo el código fuente para la clase Point, mostrado anteriormente:

<?xml version="1.0"?>  
<doc>  
 <assembly>  
 <name>Point</name>  
 </assembly>  
 <members>  
 <member name="T:Graphics.Point">  
 <remarks>Class <c>Point</c> models a point in a  
 two-dimensional plane. </remarks>  
 </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="x"/>,<paramref name="y"/>).</summary>  
 <param><c>x</c> is the new Point's x-coordinate.</param>  
 <param><c>y</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>x</c> is the new x-coordinate.</param>  
 <param><c>y</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>x</c> is the relative x-offset.</param>  
 <param><c>y</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>

# Resumen de la gramática

En esta sección se resume la gramática del lenguaje Visual Basic. Para obtener información sobre cómo leer la gramática, vea Notación gramatical.

## Gramática léxica

Start ::= [ LogicalLine+ ]

LogicalLine ::= [ LogicalLineElement+ ] [ Comment ] LineTerminator

LogicalLineElement ::= WhiteSpace | LineContinuation | Token

Token ::= Identifier | Keyword | Literal | Separator | Operator

### Caracteres y líneas

Character ::= < any Unicode character except a LineTerminator >

LineTerminator ::=  
 < Unicode carriage return character (0x000D) > |  
 < Unicode linefeed character (0x000A) > |  
 < Unicode carriage return character > < Unicode linefeed character > |  
 < Unicode line separator character (0x2028) > |  
 < Unicode paragraph separator character (0x2029) >

LineContinuation ::= WhiteSpace \_ [ WhiteSpace+ ] LineTerminator

Comma ::= , [ LineTerminator ]

OpenParenthesis ::= ( [ LineTerminator ]

CloseParenthesis ::= [ LineTerminator ] )

OpenCurlyBrace ::= { [ LineTerminator ]

CloseCurlyBrace ::= [ LineTerminator ] }

Equals ::= = [ LineTerminator ]

ColonEquals ::= : = [ LineTerminator ]

WhiteSpace ::=  
 < Unicode blank characters (class Zs) > |  
 < Unicode tab character (0x0009) >

Comment ::= CommentMarker [ Character+ ]

CommentMarker ::= SingleQuoteCharacter | REM

SingleQuoteCharacter ::=  
 ' |  
 < Unicode left single-quote character (0x2018) > |  
 < Unicode right single-quote character (0x2019) >

### Identificadores

Identifier ::=  
 NonEscapedIdentifier [ TypeCharacter ] |  
 Keyword TypeCharacter |  
 EscapedIdentifier

NonEscapedIdentifier ::= < IdentifierName but not Keyword >

EscapedIdentifier ::= [ IdentifierName ]

IdentifierName ::= IdentifierStart [ IdentifierCharacter+ ]

IdentifierStart ::=  
 AlphaCharacter |  
 UnderscoreCharacter IdentifierCharacter

IdentifierCharacter ::=  
 UnderscoreCharacter |  
 AlphaCharacter |  
 NumericCharacter |  
 CombiningCharacter |  
 FormattingCharacter

AlphaCharacter ::=  
 < Unicode alphabetic character (classes Lu, Ll, Lt, Lm, Lo, Nl) >

NumericCharacter ::= < Unicode decimal digit character (class Nd) >

CombiningCharacter ::= < Unicode combining character (classes Mn, Mc) >

FormattingCharacter ::= < Unicode formatting character (class Cf) >

UnderscoreCharacter ::= < Unicode connection character (class Pc) >

IdentifierOrKeyword ::= Identifier | Keyword

TypeCharacter ::=  
 IntegerTypeCharacter |  
 LongTypeCharacter |  
 DecimalTypeCharacter |  
 SingleTypeCharacter |  
 DoubleTypeCharacter |  
 StringTypeCharacter

IntegerTypeCharacter ::= %

LongTypeCharacter ::= &

DecimalTypeCharacter ::= @

SingleTypeCharacter ::= !

DoubleTypeCharacter ::= #

StringTypeCharacter ::= $

### Palabras clave

Keyword ::= < member of keyword table in 2.3 >

### Literales

Literal ::=  
 BooleanLiteral |  
 IntegerLiteral |  
 FloatingPointLiteral |  
 StringLiteral |  
 CharacterLiteral |  
 DateLiteral |  
 Nothing

BooleanLiteral ::= True | False

IntegerLiteral ::= IntegralLiteralValue [ IntegralTypeCharacter ]

IntegralLiteralValue ::= IntLiteral | HexLiteral | OctalLiteral

IntegralTypeCharacter ::=  
 ShortCharacter |  
 UnsignedShortCharacter |  
 IntegerCharacter |  
 UnsignedIntegerCharacter |  
 LongCharacter |  
 UnsignedLongCharacter |  
 IntegerTypeCharacter |  
 LongTypeCharacter

ShortCharacter ::= S

UnsignedShortCharacter ::= US

IntegerCharacter ::= I

UnsignedIntegerCharacter ::= UI

LongCharacter ::= L

UnsignedLongCharacter ::= UL

IntLiteral ::= Digit+

HexLiteral ::= & H HexDigit+

OctalLiteral ::= & O OctalDigit+

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

HexDigit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F

OctalDigit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

FloatingPointLiteral ::=  
 FloatingPointLiteralValue [ FloatingPointTypeCharacter ] |  
 IntLiteral FloatingPointTypeCharacter

FloatingPointTypeCharacter ::=  
 SingleCharacter |  
 DoubleCharacter |  
 DecimalCharacter |  
 SingleTypeCharacter |  
 DoubleTypeCharacter |  
 DecimalTypeCharacter

SingleCharacter ::= F

DoubleCharacter ::= R

DecimalCharacter ::= D

FloatingPointLiteralValue ::=  
 IntLiteral . IntLiteral [ Exponent ] |  
 . IntLiteral [ Exponent ] |  
 IntLiteral Exponent

Exponent ::= E [ Sign ] IntLiteral

Sign ::= + | -

StringLiteral ::=  
 DoubleQuoteCharacter [ StringCharacter+ ] DoubleQuoteCharacter

DoubleQuoteCharacter ::=  
 " |  
 < Unicode left double-quote character (0x201C) > |  
 < Unicode right double-quote character (0x201D) >

StringCharacter ::=  
 < Character except for DoubleQuoteCharacter > |  
 DoubleQuoteCharacter DoubleQuoteCharacter

CharacterLiteral ::= DoubleQuoteCharacter StringCharacter DoubleQuoteCharacter C

DateLiteral ::= # [ Whitespace+ ] DateOrTime [ Whitespace+ ] #

DateOrTime ::=  
 DateValue Whitespace+ TimeValue |  
 DateValue |  
 TimeValue

DateValue ::=  
 MonthValue / DayValue / YearValue |  
 MonthValue - DayValue - YearValue

TimeValue ::=  
 HourValue : MinuteValue [ : SecondValue ] [ WhiteSpace+ ] [ AMPM ] |  
 HourValue [ WhiteSpace+ ] AMPM

MonthValue ::= IntLiteral

DayValue ::= IntLiteral

YearValue ::= IntLiteral

HourValue ::= IntLiteral

MinuteValue ::= IntLiteral

SecondValue ::= IntLiteral

AMPM ::= AM | PM

ElseIf ::= ElseIf | Else If

Nothing ::= Nothing

Separator ::= ( | ) | { | } | ! | # | , | . | : | ?

Operator ::=  
 & | \* | + | - | / | \ | ^ | < | = | >

## Directivas de preprocesamiento

### Compilación condicional

Start ::= [ CCStatement+ ]

CCStatement ::=  
 CCConstantDeclaration |  
 CCIfGroup |  
 LogicalLine

CCExpression ::=  
 LiteralExpression |  
 CCParenthesizedExpression |  
 CCSimpleNameExpression |  
 CCCastExpression |  
 CCOperatorExpression |  
 CCConditionalExpression

CCParenthesizedExpression ::= ( CCExpression )

CCSimpleNameExpression ::= Identifier

CCCastExpression ::=   
 DirectCast ( CCExpression , TypeName ) |  
 TryCast ( CCExpression , TypeName ) |  
 CType ( CCExpression , TypeName ) |  
 CastTarget ( CCExpression )

CCOperatorExpression ::=  
 CCUnaryOperator CCExpression |  
 CCExpression CCBinaryOperator CCExpression

CCUnaryOperator ::= + | - | Not

CCBinaryOperator ::= + | - | \* | / | \ | Mod | ^ | = | < > | < | > |  
 < = | > = | & | And | Or | Xor | AndAlso | OrElse | < < | > >

CCConditionalExpression ::=   
 If ( CCExpression , CCExpression , CCExpression ) |  
 If ( CCExpression , CCExpression )

CCConstantDeclaration ::= # Const Identifier = CCExpression LineTerminator

CCIfGroup ::=  
 # If CCExpression [ Then ] LineTerminator  
 [ CCStatement+ ]  
 [ CCElseIfGroup+ ]  
 [ CCElseGroup ]  
 # End If LineTerminator

CCElseIfGroup ::=  
 # ElseIf CCExpression [ Then ] LineTerminator  
 [ CCStatement+ ]

CCElseGroup ::=  
 # Else LineTerminator  
 [ CCStatement+ ]

### Directivas de código fuente externo

Start ::= [ ExternalSourceStatement+ ]

ExternalSourceStatement ::= ExternalSourceGroup | LogicalLine

ExternalSourceGroup ::=  
 # ExternalSource ( StringLiteral , IntLiteral ) LineTerminator  
 [ LogicalLine+ ]  
 # End ExternalSource LineTerminator

### Directivas de región

Start ::= [ RegionStatement+ ]

RegionStatement ::= RegionGroup | LogicalLine

RegionGroup ::=  
 # Region StringLiteral LineTerminator  
 [ RegionStatement+ ]  
 # End Region LineTerminator

### Directivas de suma de comprobación externa

Start ::= [ ExternalChecksumStatement+ ]

ExternalChecksumStatement ::=  
 # ExternalChecksum ( StringLiteral , StringLiteral , StringLiteral ) LineTerminator

## Gramática sintáctica

AccessModifier ::= Public | Protected | Friend | Private | Protected Friend

TypeParameterList ::=  
 OpenParenthesis Of TypeParameters CloseParenthesis

TypeParameters ::=  
 TypeParameter |  
 TypeParameters Comma TypeParameter

TypeParameter ::=  
 [ VarianceModifier ] Identifier [ TypeParameterConstraints ]

VarianceModifier ::=  
 In | Out

TypeParameterConstraints ::=  
 As Constraint |  
 As OpenCurlyBrace ConstraintList CloseCurlyBrace

ConstraintList ::=  
 ConstraintList *Comma* Constraint |  
 Constraint

Constraint ::= TypeName | New | Structure | Class

### Atributos

Attributes ::=  
 AttributeBlock |  
 Attributes AttributeBlock

AttributeBlock ::= [ LineTerminator ] < AttributeList [ LineTerminator ] > [ LineTerminator ]

AttributeList ::=  
 Attribute |  
 AttributeList *Comma* Attribute

Attribute ::=  
 [ AttributeModifier : ] SimpleTypeName [ OpenParenthesis [ AttributeArguments ] CloseParenthesis ]

AttributeModifier ::= Assembly | Module

AttributeArguments ::=  
 AttributePositionalArgumentList |  
 AttributePositionalArgumentList Comma VariablePropertyInitializerList |  
 VariablePropertyInitializerList

AttributePositionalArgumentList ::=  
 AttributeArgumentExpression |  
 AttributePositionalArgumentList Comma AttributeArgumentExpression

VariablePropertyInitializerList ::=  
 VariablePropertyInitializer |  
 VariablePropertyInitializerList Comma VariablePropertyInitializer

VariablePropertyInitializer ::=  
 IdentifierOrKeyword ColonEquals AttributeArgumentExpression

AttributeArgumentExpression ::=  
 ConstantExpression |  
 GetTypeExpression |  
 ArrayExpression

### Archivos de código fuente y espacios de nombres

Start ::=  
 [ OptionStatement+ ]  
 [ ImportsStatement+ ]  
 [ AttributesStatement+ ]  
 [ NamespaceMemberDeclaration+ ]

StatementTerminator ::= LineTerminator | :

AttributesStatement ::= Attributes StatementTerminator

OptionStatement ::=  
 OptionExplicitStatement |  
 OptionStrictStatement |  
 OptionCompareStatement |  
 OptionInferStatement

OptionExplicitStatement ::= Option Explicit [ OnOff ] StatementTerminator

OnOff ::= On | Off

OptionStrictStatement ::= Option Strict [ OnOff ] StatementTerminator

OptionCompareStatement ::= Option Compare CompareOption StatementTerminator

CompareOption ::= Binary | Text

OptionInferStatement ::= Option Infer [ OnOff ] StatementTerminator

ImportsStatement ::= Imports ImportsClauses StatementTerminator

ImportsClauses ::=  
 ImportsClause |  
 ImportsClauses Comma ImportsClause

ImportsClause ::=  
 AliasImportsClause |  
 MembersImportsClause |  
 XMLNamespaceImportsClause

AliasImportsClause ::=   
 Identifier Equals TypeName

MembersImportsClause ::=  
 TypeName

XMLNamespaceImportsClause ::=  
 < XMLNamespaceAttributeName [ XMLWhitespace ] *Equals* [ XMLWhitespace ] XMLNamespaceValue >

XMLNamespaceValue ::=  
 DoubleQuoteCharacter [ XMLAttributeDoubleQuoteValueCharacter+ ] DoubleQuoteCharacter |  
 SingleQuoteCharacter [ XMLAttributeSingleQuoteValueCharacter+ ] SingleQuoteCharacter

NamespaceDeclaration ::=  
 Namespace NamespaceName StatementTerminator  
 [ NamespaceMemberDeclaration+ ]  
 End Namespace StatementTerminator

NamespaceName ::=   
 RelativeNamespaceName |  
 Global |  
 Global . RelativeNamespaceName

*RelativeNamespaceName ::=*  
 Identifier |  
 *Relative*NamespaceName Period IdentifierOrKeyword

NamespaceMemberDeclaration ::=  
 NamespaceDeclaration |  
 TypeDeclaration

TypeDeclaration ::=  
 ModuleDeclaration |  
 NonModuleDeclaration

NonModuleDeclaration ::=  
 EnumDeclaration |  
 StructureDeclaration |  
 InterfaceDeclaration |  
 ClassDeclaration |  
 DelegateDeclaration

### Tipos

TypeName ::=  
 ArrayTypeName |  
 NonArrayTypeName

NonArrayTypeName ::=  
 SimpleTypeName | NullableTypeName

SimpleTypeName ::=  
 QualifiedTypeName |  
 BuiltInTypeName

QualifiedTypeName ::=  
 Identifier [ TypeArguments ] |  
 Global Period IdentifierOrKeyword [ TypeArguments ] |  
 QualifiedTypeName Period IdentifierOrKeyword [ TypeArguments ]

TypeArguments ::=  
 OpenParenthesis Of TypeArgumentList CloseParenthesis

TypeArgumentList ::=  
 TypeName |  
 TypeArgumentList Comma TypeName

BuiltInTypeName ::= Object | PrimitiveTypeName

TypeModifier ::= AccessModifier | Shadows

IdentifierModifiers ::= [ *NullableNameModifier* ] [ ArrayNameModifier ]

NullableTypeName ::= NonArrayTypeName ?

NullableNameModifier ::= ?

TypeImplementsClause ::= Implements *Type*Implements StatementTerminator

TypeImplements ::=  
 NonArrayTypeName |  
 *Type*Implements Comma NonArrayTypeName

PrimitiveTypeName ::= NumericTypeName | Boolean | Date | Char | String

NumericTypeName ::= IntegralTypeName | FloatingPointTypeName | Decimal

IntegralTypeName ::= Byte | SByte | UShort | Short | UInteger | Integer | ULong | Long

FloatingPointTypeName ::= Single | Double

EnumDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Enum Identifier [ As NonArrayTypeName ] StatementTerminator  
 EnumMemberDeclaration+  
 End Enum StatementTerminator

EnumMemberDeclaration ::= [ Attributes ] Identifier [ Equals ConstantExpression ] StatementTerminator

ClassDeclaration ::=  
 [ Attributes ] [ ClassModifier+ ] Class Identifier [ TypeParameterList ] StatementTerminator  
 [ ClassBase ]  
 [ TypeImplementsClause+ ]  
 [ ClassMemberDeclaration+ ]  
 End Class StatementTerminator

ClassModifier ::= TypeModifier | MustInherit | NotInheritable | Partial

ClassBase ::= Inherits NonArrayTypeName StatementTerminator

ClassMemberDeclaration ::=  
 NonModuleDeclaration |  
 EventMemberDeclaration |  
 VariableMemberDeclaration |  
 ConstantMemberDeclaration |  
 MethodMemberDeclaration |  
 PropertyMemberDeclaration |  
 ConstructorMemberDeclaration |  
 OperatorDeclaration

StructureDeclaration ::=  
 [ Attributes ] [ StructureModifier+ ] Structure Identifier [ TypeParameterList ]  
 StatementTerminator  
 [ TypeImplementsClause+ ]  
 [ StructMemberDeclaration+ ]  
 End Structure StatementTerminator

StructureModifier ::= TypeModifier | Partial

StructMemberDeclaration ::=  
 NonModuleDeclaration |  
 VariableMemberDeclaration |  
 ConstantMemberDeclaration |  
 EventMemberDeclaration |  
 MethodMemberDeclaration |  
 PropertyMemberDeclaration |  
 ConstructorMemberDeclaration |  
 OperatorDeclaration

ModuleDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Module Identifier StatementTerminator  
 [ ModuleMemberDeclaration+ ]  
 End Module StatementTerminator

ModuleMemberDeclaration ::=  
 NonModuleDeclaration |  
 VariableMemberDeclaration |  
 ConstantMemberDeclaration |  
 EventMemberDeclaration |  
 MethodMemberDeclaration |  
 PropertyMemberDeclaration |  
 ConstructorMemberDeclaration

InterfaceDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Interface Identifier [ TypeParameterList ] StatementTerminator  
 [ InterfaceBase+ ]  
 [ InterfaceMemberDeclaration+ ]  
 End Interface StatementTerminator

InterfaceBase ::= Inherits InterfaceBases StatementTerminator

InterfaceBases ::=  
 NonArrayTypeName |  
 InterfaceBases Comma NonArrayTypeName

InterfaceMemberDeclaration ::=  
 NonModuleDeclaration |  
 InterfaceEventMemberDeclaration |  
 InterfaceMethodMemberDeclaration |  
 InterfacePropertyMemberDeclaration

ArrayTypeName ::= NonArrayTypeName ArrayTypeModifiers

ArrayTypeModifiers ::= ArrayTypeModifier+

ArrayTypeModifier ::= OpenParenthesis [ RankList ] CloseParenthesis

RankList ::=  
 Comma |  
 RankList Comma

ArrayNameModifier ::=  
 ArrayTypeModifiers |  
 ArraySizeInitializationModifier

DelegateDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Delegate MethodSignature StatementTerminator

MethodSignature ::= SubSignature | FunctionSignature

### Miembros de tipo

ImplementsClause ::= [ Implements ImplementsList ]

ImplementsList ::=  
 InterfaceMemberSpecifier |  
 ImplementsList Comma InterfaceMemberSpecifier

InterfaceMemberSpecifier ::= NonArrayTypeName Period IdentifierOrKeyword

MethodMemberDeclaration ::= MethodDeclaration | ExternalMethodDeclaration

InterfaceMethodMemberDeclaration ::= InterfaceMethodDeclaration

MethodDeclaration ::=  
 SubDeclaration |  
 MustOverrideSubDeclaration |  
 FunctionDeclaration |  
 MustOverrideFunctionDeclaration

InterfaceMethodDeclaration ::=  
 InterfaceSubDeclaration |  
 InterfaceFunctionDeclaration

SubSignature ::= Sub Identifier [ TypeParameterList ]  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ]

FunctionSignature ::= Function Identifier [ TypeParameterList ]  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ] [ As [ Attributes ] TypeName ]

SubDeclaration ::=  
 [ Attributes ] [ ProcedureModifier+ ] SubSignature [ HandlesOrImplements ] LineTerminator  
 Block  
 End Sub StatementTerminator

MustOverrideSubDeclaration ::=  
 [ Attributes ] MustOverrideProcedureModifier+ SubSignature [ HandlesOrImplements ]  
 StatementTerminator

InterfaceSubDeclaration ::=  
 [ Attributes ] [ InterfaceProcedureModifier+ ] SubSignature StatementTerminator

FunctionDeclaration ::=  
 [ Attributes ] [ ProcedureModifier+ ] FunctionSignature [ HandlesOrImplements ]  
 LineTerminator  
 Block  
 End Function StatementTerminator

MustOverrideFunctionDeclaration ::=  
 [ Attributes ] MustOverrideProcedureModifier+ FunctionSignature  
 [ HandlesOrImplements ] StatementTerminator

InterfaceFunctionDeclaration ::=  
 [ Attributes ] [ InterfaceProcedureModifier+ ] FunctionSignature StatementTerminator

ProcedureModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 Overridable |  
 NotOverridable |  
 Overrides |  
 Overloads |  
 Partial |  
 Iterator |  
 Async

MustOverrideProcedureModifier ::= ProcedureModifier | MustOverride

InterfaceProcedureModifier ::= Shadows | Overloads

HandlesOrImplements ::= HandlesClause | ImplementsClause

ExternalMethodDeclaration ::=  
 ExternalSubDeclaration |  
 ExternalFunctionDeclaration

ExternalSubDeclaration ::=  
 [ Attributes ] [ ExternalMethodModifier+ ] Declare [ CharsetModifier ] Sub Identifier  
 LibraryClause [ AliasClause ] [ OpenParenthesis [ ParameterList ] CloseParenthesis ] StatementTerminator

ExternalFunctionDeclaration ::=  
 [ Attributes ] [ ExternalMethodModifier+ ] Declare [ CharsetModifier ] Function Identifier  
 LibraryClause [ AliasClause ] [ OpenParenthesis [ ParameterList ] CloseParenthesis ] [ As [ Attributes ] TypeName ]  
 StatementTerminator

ExternalMethodModifier ::= AccessModifier | Shadows | Overloads

CharsetModifier ::= Ansi | Unicode | Auto

LibraryClause ::= Lib StringLiteral

AliasClause ::= Alias StringLiteral

ParameterList ::=  
 Parameter |  
 ParameterList Comma Parameter

Parameter ::=  
 [ Attributes ] [ ParameterModifier+ ] ParameterIdentifier [ As TypeName ]  
 [ *Equals* ConstantExpression ]

ParameterModifier ::= ByVal | ByRef | Optional | ParamArray

ParameterIdentifier ::= Identifier IdentifierModifiers

HandlesClause ::= [ Handles EventHandlesList ]

EventHandlesList ::=  
 EventMemberSpecifier |  
 EventHandlesList Comma EventMemberSpecifier

EventMemberSpecifier ::=  
 Identifier Period IdentifierOrKeyword |  
 MyBase Period IdentifierOrKeyword |  
 Me Period IdentifierOrKeyword

ConstructorMemberDeclaration ::=  
 [ Attributes ] [ ConstructorModifier+ ] Sub New  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ] LineTerminator  
 [ Block ]  
 End Sub StatementTerminator

ConstructorModifier ::= AccessModifier | Shared

EventMemberDeclaration ::=  
 RegularEventMemberDeclaration |  
 CustomEventMemberDeclaration

RegularEventMemberDeclaration ::=  
 [ Attributes ] [ EventModifiers+ ] Event Identifier ParametersOrType [ ImplementsClause ]  
 StatementTerminator

InterfaceEventMemberDeclaration ::=  
 [ Attributes ] [ InterfaceEventModifiers+ ] Event Identifier ParametersOrType StatementTerminator

ParametersOrType ::=  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ] |  
 As NonArrayTypeName

EventModifiers ::= AccessModifier | Shadows | Shared

InterfaceEventModifiers ::= Shadows

CustomEventMemberDeclaration ::=  
 [ Attributes ] [ EventModifiers+ ] Custom Event Identifier As TypeName [ ImplementsClause ]  
 StatementTerminator  
 EventAccessorDeclaration+  
 End Event StatementTerminator

EventAccessorDeclaration ::=  
 AddHandlerDeclaration |  
 RemoveHandlerDeclaration |  
 RaiseEventDeclaration

AddHandlerDeclaration ::=  
 [ Attributes ] AddHandler OpenParenthesis ParameterList CloseParenthesis LineTerminator  
 [ Block ]  
 End AddHandler StatementTerminator

RemoveHandlerDeclaration ::=  
 [ Attributes ] RemoveHandler OpenParenthesis ParameterList CloseParenthesis LineTerminator  
 [ Block ]  
 End RemoveHandler StatementTerminator

RaiseEventDeclaration ::=  
 [ Attributes ] RaiseEvent OpenParenthesis ParameterList CloseParenthesis LineTerminator  
 [ Block ]  
 End RaiseEvent StatementTerminator

ConstantMemberDeclaration ::=  
 [ Attributes ] [ ConstantModifier+ ] Const ConstantDeclarators StatementTerminator

ConstantModifier ::= AccessModifier | Shadows

ConstantDeclarators ::=  
 ConstantDeclarator |  
 ConstantDeclarators Comma ConstantDeclarator

ConstantDeclarator ::= Identifier [ As TypeName ] Equals ConstantExpression StatementTerminator

VariableMemberDeclaration ::=  
 [ Attributes ] VariableModifier+ VariableDeclarators StatementTerminator

VariableModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 ReadOnly |  
 WithEvents |  
 Dim

VariableDeclarators ::=  
 VariableDeclarator |  
 VariableDeclarators Comma VariableDeclarator

VariableDeclarator ::=  
 VariableIdentifiers As ObjectCreationExpression |  
 VariableIdentifiers [ As TypeName ] [ Equals Expression ]

VariableIdentifiers ::=  
 VariableIdentifier |  
 VariableIdentifiers Comma VariableIdentifier

VariableIdentifier ::= Identifier IdentifierModifiers

ArraySizeInitializationModifier ::=  
 OpenParenthesis BoundList CloseParenthesis [ ArrayTypeModifiers ]

BoundList::=  
 Bound |  
 BoundList Comma Bound

Bound ::=  
 Expression |  
 0 To Expression

PropertyMemberDeclaration ::=  
 RegularPropertyMemberDeclaration |  
 MustOverridePropertyMemberDeclaration |  
 AutoPropertyMemberDeclaration

PropertySignature ::=  
 Property Identifier [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 [ As [ Attributes ] TypeName ]

RegularPropertyMemberDeclaration ::=  
 [ Attributes ] [ PropertyModifier+ ] PropertySignature [ ImplementsClause ] LineTerminator  
 PropertyAccessorDeclaration+  
 End Property StatementTerminator

MustOverridePropertyMemberDeclaration ::=  
 [ Attributes ] MustOverridePropertyModifier+ PropertySignature [ ImplementsClause ]  
 StatementTerminator

AutoPropertyMemberDeclaration ::=  
 [ Attributes ] [ AutoPropertyModifier+ ] Property Identifier  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 [ As [ Attributes ] TypeName ] [ Equals Expression ] [ ImplementsClause ] LineTerminator |  
 [ Attributes ] [ AutoPropertyModifier+ ] Property Identifier  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 As [ Attributes ] New [ NonArrayTypeName  
 [ OpenParenthesis [ ArgumentList ] CloseParenthesis ] ] [ ObjectCreationExpressionInitializer ]  
 [ ImplementsClause ] LineTerminator

InterfacePropertyMemberDeclaration ::=  
 [ Attributes ] [ InterfacePropertyModifier+ ] PropertySignature StatementTerminator

AutoPropertyModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 Overridable |  
 NotOverridable |  
 Overrides |  
 Overloads

PropertyModifier ::=  
 AutoPropertyModifier |  
 Default |  
 ReadOnly |  
 WriteOnly |  
 Iterator

MustOverridePropertyModifier ::= PropertyModifier | MustOverride

InterfacePropertyModifier ::=  
 Shadows |  
 Overloads |  
 Default |  
 ReadOnly |  
 WriteOnly

PropertyAccessorDeclaration ::= PropertyGetDeclaration | PropertySetDeclaration

PropertyGetDeclaration ::=  
 [ Attributes ] [ AccessModifier ] Get LineTerminator  
 [ Block ]  
 End Get StatementTerminator

PropertySetDeclaration ::=  
 [ Attributes ] [ AccessModifier ] Set [ OpenParenthesis [ ParameterList ] CloseParenthesis ] LineTerminator  
 [ Block ]  
 End Set StatementTerminator

OperatorDeclaration ::=  
 [ Attributes ] [ OperatorModifier+ ] Operator OverloadableOperator OpenParenthesis ParameterList CloseParenthesis  
 [ As [ Attributes ] TypeName ] LineTerminator  
 [ Block ]  
 End Operator StatementTerminator

OperatorModifier ::= Public | Shared | Overloads | Shadows | Widening | Narrowing

OverloadableOperator ::=  
 + | - | \* | / | \ | & | Like | Mod | And | Or | Xor | ^ | < < | > > |  
 = | < > | > | < | > = | < = | Not | IsTrue | IsFalse | CType

### Instrucciones

Statement ::=  
 LabelDeclarationStatement |  
 LocalDeclarationStatement |  
 WithStatement |  
 SyncLockStatement |  
 EventStatement |  
 AssignmentStatement |  
 InvocationStatement |  
 ConditionalStatement |  
 LoopStatement |  
 ErrorHandlingStatement |  
 BranchStatement |  
 ArrayHandlingStatement |  
 UsingStatement |  
 AwaitStatement |  
 YieldStatement

Block ::= [ Statements+ ]

LabelDeclarationStatement ::= LabelName :

LabelName ::= Identifier | IntLiteral

Statements ::=  
 [ Statement ] |  
 Statements : [ Statement ]

LocalDeclarationStatement ::= LocalModifier VariableDeclarators StatementTerminator

LocalModifier ::= Static | Dim | Const

WithStatement ::=  
 With Expression StatementTerminator  
 [ Block ]  
 End With StatementTerminator

SyncLockStatement ::=  
 SyncLock Expression StatementTerminator  
 [ Block ]  
 End SyncLock StatementTerminator

EventStatement ::=  
 RaiseEventStatement |  
 AddHandlerStatement |  
 RemoveHandlerStatement

RaiseEventStatement ::= RaiseEvent IdentifierOrKeyword [ OpenParenthesis [ ArgumentList ] CloseParenthesis ]  
 StatementTerminator

AddHandlerStatement ::= AddHandler Expression Comma Expression StatementTerminator

RemoveHandlerStatement ::= RemoveHandler Expression Comma Expression StatementTerminator

AssignmentStatement ::=  
 RegularAssignmentStatement |  
 CompoundAssignmentStatement |  
 MidAssignmentStatement

RegularAssignmentStatement ::= Expression Equals Expression StatementTerminator

CompoundAssignmentStatement ::= Expression CompoundBinaryOperator [ LineTerminator ]  
 Expression StatementTerminator

CompoundBinaryOperator ::= ^ = | \* = | / = | \ = | + = | - = | & = | < < = | > > =

MidAssignmentStatement ::=  
 Mid [ $ ] OpenParenthesis Expression Comma Expression [ Comma Expression ] CloseParenthesis  
 Equals Expression StatementTerminator

InvocationStatement ::= [ Call ] InvocationExpression StatementTerminator

ConditionalStatement ::= IfStatement | SelectStatement

IfStatement ::= BlockIfStatement | LineIfThenStatement

BlockIfStatement ::=  
 If BooleanExpression [ Then ] StatementTerminator  
 [ Block ]  
 [ ElseIfStatement+ ]  
 [ ElseStatement ]  
 End If StatementTerminator

ElseIfStatement ::=  
 ElseIf BooleanExpression [ Then ] StatementTerminator  
 [ Block ]

ElseStatement ::=  
 Else StatementTerminator  
 [ Block ]

LineIfThenStatement ::=  
 If BooleanExpression Then Statements [ Else Statements ] StatementTerminator

SelectStatement ::=  
 Select [ Case ] Expression StatementTerminator  
 [ CaseStatement+ ]  
 [ CaseElseStatement ]  
 End Select StatementTerminator

CaseStatement ::=  
 Case CaseClauses StatementTerminator  
 [ Block ]

CaseClauses ::=  
 CaseClause |  
 CaseClauses Comma CaseClause

CaseClause ::=  
 [ Is [ LineTerminator ] ] ComparisonOperator [ LineTerminator ] Expression |  
 Expression [ To Expression ]

ComparisonOperator ::= = | < > | < | > | > = | < =

CaseElseStatement ::=  
 Case Else StatementTerminator  
 [ Block ]

LoopStatement ::=  
 WhileStatement |  
 DoLoopStatement |  
 ForStatement |  
 ForEachStatement

WhileStatement ::=  
 While BooleanExpression StatementTerminator  
 [ Block ]  
 End While StatementTerminator

DoLoopStatement ::= DoTopLoopStatement | DoBottomLoopStatement

DoTopLoopStatement ::=  
 Do [ WhileOrUntil BooleanExpression ] StatementTerminator  
 [ Block ]  
 Loop StatementTerminator

DoBottomLoopStatement ::=  
 Do StatementTerminator  
 [ Block ]  
 Loop WhileOrUntil BooleanExpression StatementTerminator

WhileOrUntil ::= While | Until

ForStatement ::=  
 For LoopControlVariable Equals Expression To Expression [ Step Expression ] StatementTerminator  
 [ Block ]  
 [ Next [ NextExpressionList ] StatementTerminator ]

LoopControlVariable ::=  
 Identifier [ IdentifierModifiers As TypeName ] |  
 Expression

NextExpressionList ::=  
 Expression |  
 NextExpressionList Comma Expression

ForEachStatement ::=  
 For Each LoopControlVariable In [ LineTerminator ] Expression StatementTerminator  
 [ Block ]  
 [ Next [ NextExpressionList ] StatementTerminator ]

ErrorHandlingStatement ::=  
 StructuredErrorStatement |  
 UnstructuredErrorStatement

StructuredErrorStatement ::=  
 ThrowStatement |  
 TryStatement

TryStatement ::=  
 Try StatementTerminator  
 [ Block ]  
 [ CatchStatement+ ]  
 [ FinallyStatement ]  
 End Try StatementTerminator

FinallyStatement ::=  
 Finally StatementTerminator  
 [ Block ]

CatchStatement ::=  
 Catch [ Identifier [ As NonArrayTypeName ] ] [ When BooleanExpression ] StatementTerminator  
 [ Block ]

ThrowStatement ::= Throw [ Expression ] StatementTerminator

UnstructuredErrorStatement ::=  
 ErrorStatement |  
 OnErrorStatement |  
 ResumeStatement

ErrorStatement ::= Error Expression StatementTerminator

OnErrorStatement ::= On Error ErrorClause StatementTerminator

ErrorClause ::=  
 GoTo - 1 |  
 GoTo 0 |  
 GoToStatement |  
 Resume Next

ResumeStatement ::= Resume [ ResumeClause ] StatementTerminator

ResumeClause ::= Next | LabelName

BranchStatement ::=  
 GoToStatement |  
 ExitStatement |  
 ContinueStatement |  
 StopStatement |  
 EndStatement |  
 ReturnStatement

GoToStatement ::= GoTo LabelName StatementTerminator

ExitStatement ::= Exit ExitKind StatementTerminator

ExitKind ::= Do | For | While | Select | Sub | Function | Property | Try

ContinueStatement ::= Continue ContinueKind StatementTerminator

ContinueKind ::= Do | For | While

StopStatement ::= Stop StatementTerminator

EndStatement ::= End StatementTerminator

ReturnStatement ::= Return [ Expression ] StatementTerminator

ArrayHandlingStatement ::=  
 RedimStatement |  
 EraseStatement

RedimStatement ::= ReDim [ Preserve ] RedimClauses StatementTerminator

RedimClauses ::=  
 RedimClause |  
 RedimClauses Comma RedimClause

RedimClause ::= Expression ArraySizeInitializationModifier

EraseStatement ::= Erase EraseExpressions StatementTerminator

EraseExpressions ::=  
 Expression |  
 EraseExpressions Comma Expression

UsingStatement ::=  
 Using UsingResources StatementTerminator  
 [ Block ]  
 End Using StatementTerminator

UsingResources ::= VariableDeclarators | Expression

AwaitStatement ::= AwaitOperatorExpression StatementTerminator

YieldStatement ::= Yield Expressions StatementTerminator

### Expresiones

Expression ::=  
 SimpleExpression |  
 TypeExpression |  
 MemberAccessExpression |  
 DictionaryAccessExpression |  
 InvocationExpression |  
 IndexExpression |  
 NewExpression |  
 CastExpression |  
 OperatorExpression |  
 ConditionalExpression |  
 LambdaExpression |  
 QueryExpression |  
 XMLLiteralExpression |  
 XMLMemberAccessExpression

ConstantExpression ::= Expression

SimpleExpression ::=  
 LiteralExpression |  
 ParenthesizedExpression |  
 InstanceExpression |  
 SimpleNameExpression |  
 AddressOfExpression

LiteralExpression ::= Literal

ParenthesizedExpression ::= OpenParenthesis Expression CloseParenthesis

InstanceExpression ::= Me

SimpleNameExpression ::= Identifier [ OpenParenthesis Of TypeArgumentList CloseParenthesis ]

AddressOfExpression ::= AddressOf Expression

TypeExpression ::=  
 GetTypeExpression |  
 TypeOfIsExpression |  
 IsExpression |  
 GetXmlNamespaceExpression

GetTypeExpression ::= GetType OpenParenthesis GetTypeTypeName CloseParenthesis

GetTypeTypeName ::=  
 TypeName |  
 *Qualified*OpenTypeName

QualifiedOpenTypeName ::=  
 Identifier [ TypeArityList ] |  
 Global Period IdentifierOrKeyword [ TypeArityList ] |  
 QualifiedOpenTypeName Period IdentifierOrKeyword [ TypeArityList ]

*TypeArityList* ::= ( Of [ *CommaList* ] )

CommaList ::=  
 Comma |  
 CommaList Comma

TypeOfIsExpression ::= TypeOf Expression Is [ LineTerminator ] TypeName

IsExpression ::=  
 Expression Is [ LineTerminator ] Expression |  
 Expression IsNot [ LineTerminator ] Expression

GetXmlNamespaceExpression ::= GetXmlNamespace OpenParenthesis [ XMLNamespaceName ]  
 CloseParenthesis

MemberAccessExpression ::=  
 [ MemberAccessBase ] *Period* IdentifierOrKeyword  
 [ OpenParenthesis Of TypeArgumentList CloseParenthesis ]

MemberAccessBase ::=  
 Expression |  
 BuiltInTypeName |  
 Global |  
 MyClass |  
 MyBase

DictionaryAccessExpression ::= [ Expression ] ! IdentifierOrKeyword

InvocationExpression ::= Expression [ OpenParenthesis [ ArgumentList ] CloseParenthesis ]

ArgumentList ::= PositionalArgumentList |  
 PositionalArgumentList Comma NamedArgumentList |  
 NamedArgumentList

PositionalArgumentList ::=  
 [ Expression ] |  
 PositionalArgumentList Comma [ Expression ]

NamedArgumentList ::=  
 IdentifierOrKeyword ColonEquals Expression |  
 NamedArgumentList Comma IdentifierOrKeyword ColonEquals Expression

IndexExpression ::= Expression OpenParenthesis [ ArgumentList ] CloseParenthesis

NewExpression ::=  
 ObjectCreationExpression |  
 ArrayExpression |  
 AnonymousObjectCreationExpression

AnonymousObjectCreationExpression ::=  
 New ObjectMemberInitializer

ObjectCreationExpression ::=  
 New NonArrayTypeName [ OpenParenthesis [ ArgumentList ] CloseParenthesis ]  
 [ ObjectCreationExpressionInitializer ]

ObjectCreationExpressionInitializer ::= ObjectMemberInitializer | ObjectCollectionInitializer

ObjectMemberInitializer ::=  
 With OpenCurlyBrace FieldInitializerList CloseCurlyBrace

FieldInitializerList ::=  
 FieldInitializer |  
 FieldInitializerList , FieldInitializer

FieldInitializer ::= [ [ Key ] . IdentifierOrKeyword = ] Expression

ObjectCollectionInitializer ::= From CollectionInitializer

CollectionInitializer ::= OpenCurlyBrace [ CollectionElementList ] CloseCurlyBrace

CollectionElementList ::=  
 CollectionElement |  
 CollectionElementList Comma CollectionElement

CollectionElement ::=  
 *Expression* |  
 CollectionInitializer

ArrayExpression ::=  
 *ArrayCreationExpression* |  
 *ArrayLiteralExpression*

ArrayCreationExpression ::=  
 New NonArrayTypeName ArrayNameModifier CollectionInitializerArrayLiteralExpression ::= CollectionInitializer

CastExpression ::=  
 DirectCast OpenParenthesis Expression Comma TypeName CloseParenthesis |  
 TryCast OpenParenthesis Expression Comma TypeName CloseParenthesis |  
 CType OpenParenthesis Expression Comma TypeName CloseParenthesis |  
 CastTarget OpenParenthesis Expression CloseParenthesis

CastTarget ::=  
 CBool | CByte | CChar | CDate | CDec | CDbl | CInt | CLng | CObj | CSByte | CShort |  
 CSng | CStr | CUInt | CULng | CUShort

OperatorExpression ::=  
 ArithmeticOperatorExpression |  
 RelationalOperatorExpression |  
 LikeOperatorExpression |  
 ConcatenationOperatorExpression |  
 ShortCircuitLogicalOperatorExpression |  
 LogicalOperatorExpression |  
 ShiftOperatorExpression |  
 AwaitOperatorExpression

ArithmeticOperatorExpression ::=  
 UnaryPlusExpression |  
 UnaryMinusExpression |  
 AdditionOperatorExpression |  
 SubtractionOperatorExpression |  
 MultiplicationOperatorExpression |  
 DivisionOperatorExpression |  
 ModuloOperatorExpression |  
 ExponentOperatorExpression

UnaryPlusExpression ::= + Expression

UnaryMinusExpression ::= - Expression

AdditionOperatorExpression ::= Expression + [ LineTerminator ] Expression

SubtractionOperatorExpression ::= Expression - [ LineTerminator ] Expression

MultiplicationOperatorExpression ::= Expression \* [ LineTerminator ] Expression

DivisionOperatorExpression ::=  
 FPDivisionOperatorExpression |  
 IntegerDivisionOperatorExpression

FPDivisionOperatorExpression ::= Expression / [ LineTerminator ] Expression

IntegerDivisionOperatorExpression ::= Expression \ [ LineTerminator ] Expression

ModuloOperatorExpression ::= Expression Mod [ LineTerminator ] Expression

ExponentOperatorExpression ::= Expression ^ [ LineTerminator ] Expression

RelationalOperatorExpression ::=  
 Expression = [ LineTerminator ] Expression |  
 Expression < > [ LineTerminator ] Expression |  
 Expression < [ LineTerminator ] Expression |  
 Expression > [ LineTerminator ] Expression |  
 Expression < = [ LineTerminator ] Expression |  
 Expression > = [ LineTerminator ] Expression

LikeOperatorExpression ::= Expression Like [ LineTerminator ] Expression

ConcatenationOperatorExpression ::= Expression & [ LineTerminator ] Expression

LogicalOperatorExpression ::=  
 Not Expression |  
 Expression And [ LineTerminator ] Expression |  
 Expression Or [ LineTerminator ] Expression |  
 Expression Xor [ LineTerminator ] Expression

ShortCircuitLogicalOperatorExpression ::=  
 Expression AndAlso [ LineTerminator ] Expression |  
 Expression OrElse [ LineTerminator ] Expression

ShiftOperatorExpression ::=  
 Expression < < [ LineTerminator ] Expression |  
 Expression > > [ LineTerminator ] Expression

BooleanExpression ::= Expression

LambdaExpression ::=  
 SingleLineLambda |  
 MultiLineLambda

SingleLineLambda ::=  
 [ LambdaModifier+ ] Function [ OpenParenthesis [ ParametertList ] CloseParenthesis ] Expression |  
 [ LambdaModifier+ ] Sub [ OpenParenthesis [ ParametertList ] CloseParenthesis ] Statement

MultiLineLambda ::=  
 MultiLineFunctionLambda |  
 MultiLineSubLambda

MultiLineFunctionLambda ::=  
 [ LambdaModifier+ ] Function [ OpenParenthesis [ ParametertList ] CloseParenthesis ] [ As TypeName ] LineTerminator  
 Block  
 End Function

MultiLineSubLambda ::=  
 [ LambdaModifier+ ] Sub [ OpenParenthesis [ ParametertList ] CloseParenthesis ] LineTerminator  
 Block  
 End Sub

LambdaModifier ::=  
 Async |  
 Iterator

QueryExpression ::=   
 FromOrAggregateQueryOperator |  
 QueryExpression QueryOperator

FromOrAggregateQueryOperator ::= FromQueryOperator | AggregateQueryOperator

JoinOrGroupJoinQueryOperator := JoinQueryOperator | GroupJoinQueryOperator

QueryOperator ::=  
 FromQueryOperator |  
 AggregateQueryOperator |  
 SelectQueryOperator |  
 DistinctQueryOperator |  
 WhereQueryOperator |  
 OrderByQueryOperator |  
 PartitionQueryOperator |  
 LetQueryOperator |  
 GroupByQueryOperator | *JoinOr*GroupJoinQueryOperator

CollectionRangeVariableDeclarationList ::=  
 CollectionRangeVariableDeclaration |  
 CollectionRangeVariableDeclarationList Comma CollectionRangeVariableDeclaration

CollectionRangeVariableDeclaration ::=   
 Identifier [ As TypeName ] In [ LineTerminator ] Expression

ExpressionRangeVariableDeclarationList ::=  
 ExpressionRangeVariableDeclaration |  
 ExpressionRangeVariableDeclarationList Comma ExpressionRangeVariableDeclaration

ExpressionRangeVariableDeclaration ::=   
 Identifier [ As TypeName ] Equals Expression

FromQueryOperator ::=  
 [ LineTerminator ] From [ LineTerminator ] CollectionRangeVariableDeclarationList

JoinQueryOperator ::=  
 [ LineTerminator ] Join [ LineTerminator ] CollectionRangeVariableDeclaration  
 [ JoinOrGroupJoinQueryOperator ] [ LineTerminator ] On [ LineTerminator ] JoinConditionList

JoinConditionList ::=  
 JoinCondition |  
 JoinConditionList And [ LineTerminator ] JoinCondition

JoinCondition ::= Expression Equals [ LineTerminator ] Expression

LetQueryOperator ::=  
 [ LineTerminator ] Let [ LineTerminator ] ExpressionRangeVariableDeclarationList

SelectQueryOperator ::=  
 [ LineTerminator ] Select [ LineTerminator ] ExpressionRangeVariableDeclarationList

DistinctQueryOperator ::=  
 [ LineTerminator ] Distinct [ LineTerminator ]

WhereQueryOperator ::=   
 [ LineTerminator ] Where [ LineTerminator ] BooleanExpression

PartitionQueryOperator ::=   
 [ LineTerminator ] Take [ LineTerminator ] Expression |  
 [ LineTerminator ] Take While [ LineTerminator ] BooleanExpression |  
 [ LineTerminator ] Skip [ LineTerminator ] Expression |  
 [ LineTerminator ] Skip While [ LineTerminator ] BooleanExpression

OrderByQueryOperator ::=  
 [ LineTerminator ] Order By [ LineTerminator ] OrderExpressionList

OrderExpressionList ::=  
 OrderExpression |  
 OrderExpressionList Comma OrderExpression

OrderExpression ::=  
 Expression [ Ordering ]

Ordering ::= Ascending | Descending

GroupByQueryOperator ::=  
 [ LineTerminator ] Group [ [ LineTerminator ] ExpressionRangeVariableDeclarationList ]  
 [ LineTerminator ] By [ LineTerminator ] ExpressionRangeVariableDeclarationList  
 [ LineTerminator ] Into [ LineTerminator ] ExpressionRangeVariableDeclarationList

AggregateQueryOperator ::=  
 [ LineTerminator ] Aggregate [ LineTerminator ] CollectionRangeVariableDeclaration  
 [ QueryOperator+ ]  
 [ LineTerminator ] Into [ LineTerminator ] ExpressionRangeVariableDeclarationList

GroupJoinQueryOperator ::=  
 [ LineTerminator ] Group Join [ LineTerminator ] CollectionRangeVariableDeclaration  
 [ JoinOrGroupJoinQueryOperator ] [ LineTerminator ] On [ LineTerminator ] JoinConditionList  
 [ LineTerminator ] Into [ LineTerminator ] ExpressionRangeVariableDeclarationList

ConditionalExpression ::=   
 If OpenParenthesis BooleanExpression Comma Expression Comma Expression CloseParenthesis |  
 If OpenParenthesis Expression Comma Expression CloseParenthesis

XMLLiteralExpression ::=  
 XMLDocument |  
 XMLElement | XMLProcessingInstruction |  
 XMLComment |  
 XMLCDATASection

XMLCharacter ::=  
 < Unicode tab character (0x0009) > |  
 < Unicode linefeed character (0x000A) > |  
 < Unicode carriage return character (0x000D) > |  
 < Unicode characters 0x0020 – 0xD7FF > |  
 < Unicode characters 0xE000 – 0xFFFD > |  
 < Unicode characters 0x10000 – 0x10FFFF >

XMLString ::= XMLCharacter+

XMLWhitespace ::= XMLWhitespaceCharacter+

XMLWhitespaceCharacter ::=  
 < Unicode carriage return character (0x000D) > |  
 < Unicode linefeed character (0x000A) > |  
 < Unicode space character (0x0020) > |  
 < Unicode tab character (0x0009) >

XMLNameCharacter ::= XMLLetter | XMLDigit | . | - | \_ | : | XMLCombiningCharacter | XMLExtender

XMLNameStartCharacter ::= XMLLetter | \_ | :

XMLName ::= XMLNameStartCharacter [ XMLNameCharacter+ ]

XMLLetter ::=   
 < Unicode character as defined in the Letter production of the XML 1.0 specification >

XMLDigit ::=  
 < Unicode character as defined in the Digit production of the XML 1.0 specification >

XMLCombiningCharacter ::=  
 < Unicode character as defined in the CombiningChar production of the XML 1.0 specification >

XMLExtender ::=  
 < Unicode character as defined in the Extender production of the XML 1.0 specification >

XMLEmbeddedExpression ::=  
 < % = [ LineTerminator ] Expression [ LineTerminator ] % >

XMLDocument ::=  
 XMLDocumentPrologue [ XMLMisc+ ] XMLDocumentBody [ XMLMisc+ ]

XMLDocumentPrologue ::=  
 < ? xml XMLVersion [ XMLEncoding ] [ XMLStandalone ] [ XMLWhitespace ] ? >

XMLVersion ::=  
 XMLWhitespace version [ XMLWhitespace ] = [ XMLWhitespace ] XMLVersionNumberValue

XMLVersionNumberValue ::=   
 SingleQuoteCharacter 1 . 0 SingleQuoteCharacter |  
 DoubleQuoteCharacter 1 . 0 DoubleQuoteCharacter

XMLEncoding ::=  
 XMLWhitespace encoding [ XMLWhitespace ] = [ XMLWhitespace ] XMLEncodingNameValue

XMLEncodingNameValue ::=   
 SingleQuoteCharacter XMLEncodingName SingleQuoteCharacter |  
 DoubleQuoteCharacter XMLEncodingName DoubleQuoteCharacter

XMLEncodingName ::= XMLLatinAlphaCharacter [ XMLEncodingNameCharacter+ ]

XMLEncodingNameCharacter ::=  
 XMLUnderscoreCharacter |  
 XMLLatinAlphaCharacter |  
 XMLNumericCharacter |  
 XMLPeriodCharacter |  
 XMLDashCharacter

XMLLatinAlphaCharacter ::=  
 < Unicode Latin alphabetic character (0x0041-0x005a, 0x0061-0x007a) >

XMLNumericCharacter ::= < Unicode digit character (0x0030-0x0039) >

XMLHexNumericCharacter ::=  
 XMLNumericCharacter |  
 < Unicode Latin hex alphabetic character (0x0041-0x0046, 0x0061-0x0066) >

XMLPeriodCharacter ::= < Unicode period character (0x002e) >

XMLUnderscoreCharacter ::= < Unicode underscore character (0x005f) >

XMLDashCharacter ::= < Unicode dash character (0x002d) >

XMLStandalone ::=  
 XMLWhitespace standalone [ XMLWhitespace ] = [ XMLWhitespace ] XMLYesNoValue

XMLYesNoValue ::=   
 SingleQuoteCharacter XMLYesNo SingleQuoteCharacter |  
 DoubleQuoteCharacter XMLYesNo DoubleQuoteCharacter

XMLYesNo ::= yes | no

XMLMisc ::=  
 XMLComment |  
 XMLProcessingInstruction |  
 XMLWhitespace

XMLDocumentBody ::= XMLElement | XMLEmbeddedExpression

XMLElement ::=  
 XMLEmptyElement |  
 XMLElementStart XMLContent XMLElementEnd

XMLEmptyElement ::=  
 < XMLQualifiedNameOrExpression [ XMLAttribute+ ] [ XMLWhitepace ] / >

XMLElementStart ::=  
 < XMLQualifiedNameOrExpression [ XMLAttribute+ ] [ XMLWhitespace ] >

XMLElementEnd ::=  
 < / > |  
 < / XMLQualifiedName [ XMLWhitespace ] >

XMLContent ::=  
 [ XMLCharacterData ] [ XMLNestedContent [ XMLCharacterData ] ]+

XMLCharacterData ::=  
 < Any XMLCharacterDataString that does not contain the string "]]>" >

XMLCharacterDataString ::=  
 < Any Unicode character except < or & >+

XMLNestedContent ::=  
 XMLElement |  
 XMLReference |  
 XMLCDATASection |  
 XMLProcessingInstruction |  
 XMLComment |  
 XMLEmbeddedExpression

XMLAttribute ::=  
 XMLWhitespace XMLAttributeName [ XMLWhitespace ] = [ XMLWhitespace ] XMLAttributeValue |  
 XMLWhitespace XMLEmbeddedExpression

XMLAttributeName ::=  
 XMLQualifiedNameOrExpression |  
 XMLNamespaceAttributeName

XMLAttributeValue ::=  
 DoubleQuoteCharacter [ XMLAttributeDoubleQuoteValueCharacter+ ] DoubleQuoteCharacter |  
 SingleQuoteCharacter [ XMLAttributeSingleQuoteValueCharacter+ ] SingleQuoteCharacter |  
 XMLEmbeddedExpression

XMLAttributeDoubleQuoteValueCharacter ::=   
 < Any XMLCharacter except <, &, or DoubleQuoteCharacter > |  
 XMLReference

XMLAttributeSingleQuoteValueCharacter ::=  
 < Any XMLCharacter except <, &, or SingleQuoteCharacter > |  
 XMLReference

XMLReference ::= XMLEntityReference | XMLCharacterReference

XMLEntityReference ::=  
 & XMLEntityName ;

XMLEntityName ::= lt | gt | amp | apos | quot

XMLCharacterReference ::=  
 & # XMLNumericCharacter+ ; |  
 & # x XMLHexNumericCharacter+ ;

XMLNamespaceAttributeName ::=  
 XMLPrefixedNamespaceAttributeName |  
 XMLDefaultNamespaceAttributeName

XMLPrefixedNamespaceAttributeName ::=  
 xmlns : XMLNamespaceName

XMLDefaultNamespaceAttributeName ::=  
 xmlns

XMLNamespaceName ::= XMLNamespaceNameStartCharacter [ XMLNamespaceNameCharacter+ ]

XMLNamespaceNameStartCharacter ::=  
 < Any XMLNameCharacter except : >

XMLNamespaceNameCharacter ::= XMLLetter | \_

XMLQualifiedNameOrExpression ::= XMLQualifiedName | XMLEmbeddedExpression

XMLQualifiedName ::=  
 XMLPrefixedName |  
 XMLUnprefixedName

XMLPrefixedName ::= XMLNamespaceName : XMLNamespaceName

XMLUnprefixedName ::= XMLNamespaceName

XMLProcessingInstruction ::=  
 < ? XMLProcessingTarget [ XMLWhitespace [ XMLProcessingValue ] ] ? >

XMLProcessingTarget ::=  
 < Any XMLName except a casing permutation of the string "xml" >

XMLProcessingValue ::=  
 < Any XMLString that does not contain the string "?>" >

XMLComment ::=  
 < ! - - [ XMLCommentCharacter+ ] - - >

XMLCommentCharacter ::=  
 < Any XMLCharacter except dash (0x002D) > |  
 - < Any XMLCharacter except dash (0x002D) >

XMLCDATASection ::=  
 < ! [ CDATA [ [ XMLCDATASectionString ] ] ] >

XMLCDATASectionString ::=  
 < Any XMLString that does not contain the string "]]>" >

XMLMemberAccessExpression ::=  
 Expression . [ LineTerminator ] < XMLQualifiedName > |  
 Expression . [ LineTerminator ] @ [ LineTerminator ] < XMLQualifiedName > |  
 Expression . [ LineTerminator ] @ [ LineTerminator ] IdentifierOrKeyword |  
 Expression . . . [ LineTerminator ] < XMLQualifiedName >

AwaitOperatorExpression ::= Await Expression

# Lista de cambios

En la siguiente sección se enumeran los cambios realizados a la especificación entre las versiones. Tras cada cambio se indican las secciones afectadas.

## Versión 7.1 a versión 8.0

Esta actualización trata los cambios de lenguaje realizados entre las versiones 7.1 y 8.0 del lenguaje.

### Cambios importantes

* Ahora se permite que existan varios bloques de atributos antes de una declaración (es decir, <a> <b> en lugar de solo <a,b>). [5.2, 6]
* Se agrega la instrucción Continue. [2.3, 10.11]
* Se agrega la instrucción Using. [2.3, 10, 10.13]
* Se agrega el operador IsNot. [2.3, 11.5.3]
* Se agrega el calificador Global que permite el enlace en el espacio de nombres global. [2.3, 4.7, 11.6]
* Se agregan comentarios sobre la documentación XML. [12]
* Las clases derivadas pueden volver a implementar interfaces implementadas por su clase base. [4.4, 4.4.1]
* Se agrega el operador TryCast. [2.3, 11.11]
* Los atributos pueden incluir argumentos de tipo Object o matrices unidimensionales. [5.1, 5.2.2]
* Se agrega una sección sobre compatibilidad del lenguaje. [1.2, 1.2.1, 1.2.2, 1.2.3]
* Se agrega la sobrecarga de operadores. [2.3, 4.1.1, 7.5.2, 7.6.1, 9.8, 9.8.1, 9.8.2, 9.8.3, 10.8.2, 10.9.2, 11.17.1, 11.11, 8.11, 8.11.1, 8.11.2, 11.12.3]
* Se agregan los pseudo-operadores IsTrue e IsFalse. [11.19, 10.8.1, 10.9.1, 10.10.1.2]
* Se agregan tipos de enteros sin signo. [2.2.1, 2.3, 2.4.2, 7.3, 11.11, 7.4, 8.2, 8.3, 8.7, 8.8, 8.9, 10.9.2, 11.2, 11.12.3, 11.13.1, 11.13.2, 11.13.3, 11.13.4, 11.13.5, 11.13.6, 11.13.7, 11.13.8, 11.14, 11.15, 11.16, 11.17, 11.17.1, 11.18, 11.8.1]
* Se agregan declaraciones de evento personalizadas. [9.4.1]
* Los descriptores de acceso de una propiedad pueden especificar un nivel de acceso más restrictivo que su propiedad contenedora. [9.7, 9.7.1, 9.7.2, 9.7.3]
* Se agregan tipos parciales. [2.3, 7.5, 7.6, 7.11]
* Se agregan instancias predeterminadas. [11.6.2, 11.6.2.1, 11.6.2.2]
* Se agregan tipos y métodos genéricos. [2.3, 2.4.7, 4.1.1, 4.4.1, 4.5.1, 4.6, 4.7.1, 4.7.2, 4.9, 4.9.1, 4.9.2, 5.1, 5.2.2, 6.1, 6.3.1, 6.3.2, 7, 7.2, 7.5, 7.5.1, 7.6, 7.8, 7.8.1, 7.9, 7.11, 7.12, 7.12.1, 8.6, 8.10, 9.1, 9.2.1, 9.2.2, 9.3.2, 9.4, 9.6, 9.8.1, 9.8.2, 9.8.3, 10.2, 10.9.3, 10.10.1.2, 11.1, 11.4.4, 11.4.5, 11.5.1, 11.5.2, 11.5.3, 11.6, 11.6.2, 11.8, 11.8.2, 11.8.5, 11.10.1, 11.10.2, 11.10.3, 12.2.16, 12.3]

### Cambios menores

* En la compilación condicional siempre se usan comparaciones de cadenas binarias. De lo contrario, las comparaciones de cadenas no funcionarían porque las comparaciones de cadenas de texto dependen de la referencia cultural en tiempo de ejecución. [3.1.2]
* Un tipo no puede heredar de otro tipo directa o indirectamente incluido en él. También se aclaran los ejemplos. [4.3]
* Se cambia la gramática y la especificación, de forma que los tipos enumerados pueden usar los equivalentes System de los tipos fundamentes como tipo subyacente. [7.4]
* Ahora se permite la conversión entre una matriz de un tipo enumerado y una matriz del tipo subyacente de la enumeración. [8.5, 8.8. 8.9]
* Al invalidar un método, se puede hacer con un método MustOverride; de esta forma se convierte en abstracto. [4.5.1]
* Un miembro de tipo puede controlar un evento de su propia clase mediante Handles. [9.2.6]
* Los métodos y las propiedades que se declaran como Overrides ahora suponen Overloads, lo que resulta más lógico que suponer Shadows. [4.3.3]
* Los campos y las variables locales pueden inicializar varias variables a la vez mediante la sintaxis As New. [9.6]
* Se quita la restricción por la que un bloque Catch interno no podía bifurcarse en un bloque Try externo. [10.10.1.2]
* Las clases no pueden heredar de System.MulticastDelegate. [7.5.1]
* Las variables compartidas de las estructuras pueden incluir inicializadores. [9.6.3]
* Se agrega una regla por la que se prefieren los tipos numéricos en lugar de los enumerados al efectuar una resolución de sobrecarga con respecto al literal 0. [11.8.1]
* Los inicializadores de tamaño de matriz pueden indicar de forma explícita un límite inferior de cero. [9.6.3.3]
* Se agrega una directiva de suma de comprobación externa. [3.4]
* Los inicializadores de tamaño de matriz de los campos pueden ser expresiones no constantes. [9.6.3.3]
* Ahora las palabras clave con caracteres de tipo se tratan como identificadores. [2.2]
* Las constantes, los campos, las propiedades, las variables locales y los parámetros con el mismo nombre que su tipo se pueden interpretar como el miembro o como el tipo en la búsqueda de miembros. [11.6.1]
* Se agrega una sección acerca de los tipos limitados por .NET Framework y se desplaza el tema sobre System.Void a esta sección. [7, 7.13]
* Al trabajar con un proyecto y un ensamblado referenciado que definen el mismo nombre completo, se prefiere el tipo del proyecto; en caso contrario, el nombre sería ambiguo. [4.7.2, 11.4.4]
* Las instrucciones On Error no se extienden a lo largo de la llamada a New en el inicio de un constructor. [10.10.2]
* No se permite la resolución de una llamada sobrecargada en una llamada enlazada en tiempo de ejecución si el tipo contenedor es una interfaz. [11.8.1]
* Overloads y Shadows no se permiten en un módulo estándar. [4.3.3]
* Cuando se realiza una búsqueda en interfaces, un nombre que se sombrea en una ruta de acceso de la jerarquía, se sombrea en todas sus rutas de acceso. Anteriormente, se producía un error de ambigüedad. [4.3.2]
* En el enlace por medio de importaciones, los tipos y los miembros de tipo tienen preferencia sobre los espacios de nombres. [4.7.2, 11.6]
* El cambio de la propiedad predeterminada de un tipo ya no requiere Shadows. [9.7.3]
* Las palabras clave contextuales Assembly, Ansi, Auto, Preserve, Unicode y Until ya no son reservadas. [2.3]

### Aclaraciones y erratas

* Se agrega una nota para indicar que la equivalencia de ancho completo y ancho medio solo funciona en tokens completos (es decir, no se pueden mezclar dentro de un token). [1.1]
* Existen lugares del lenguaje que permiten nombres de tipo normales pero no nombres de tipo de matriz. Se agregan aclaraciones en la gramática para indicarlos. [5.2, 7, 7.2, 7.5.1, 7.8.1, 7.9, 9.1, 9.2.5, 10.10.1.2, 11.10.1, 11.10.2, 11.10.3]
* La especificación indica incorrectamente que los signos de dos puntos solo se pueden usar como separadores en el nivel de instrucción. De hecho, se pueden usar prácticamente en cualquier ubicación. Se ha aclarado la gramática y la especificación. [6, 6.2.1, 6.2.2, 6.2.3, 6.3, 6.4.1, 7.2, 7.4, 7.4.1, 7.5, 7.5.1, 7.6, 7.7, 7.8, 7.8.1, 9.2.1, 9.2.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.7.1, 9.7.2, 10]
* Como parte del punto anterior, las etiquetas se convierten en instrucciones por sí mismas y se corrige la gramática. [10, 10.1]
* Se restringe el ámbito de la gramática para que las declaraciones de delegado sean más precisas. [7.10]
* La covarianza de matrices también incluye la implementación de interfaces además de la herencia. [8.5]
* Al implementar una interfaz o controlar un evento, los identificadores incluidos detrás de un punto pueden coincidir con palabras clave. [9.1, 9.2.6]
* Las declaraciones MustOverride se separan de las declaraciones normales de método y propiedad. [9.2.1, 9.7]
* La gramática era incorrecta en las declaraciones de variable; Dim es un modificador normal, no un elemento separado. [9.6]
* Se especifica la forma en que los atributos de los campos WithEvents se transmiten a los miembros sintéticos subyacentes. [9.6.2]
* La invocación de un evento no iniciará una excepción si no existen agentes de escucha. [10.5.1]
* Se amplía la sección RaiseEvent para que se eviten las condiciones de carrera al generar un evento. [10.5.1]
* Se aclara que RaiseEvent toma un identificador, no una expresión. [10.5.1]
* Se trata un caso excepcional de la asignación Mid. [10.6.3]
* Las instrucciones tras una línea If no son opcionales. [10.8.1]
* Se amplía la gramática de Do…Loop para que resulte más explícita. [10.9.1]
* Las variables de control de bucle en los bucles For pueden ser un tipo de matriz. [10.9.2]
* El modificador de tamaño no es opcional en una expresión de creación de matriz. [11.10.2]
* Los tipos de valor devuelto de las propiedades pueden contar con atributos. [9.7]
* Se separan las versiones de interfaz de las declaraciones de evento, propiedad y método para mejorar la claridad gramatical. [7.8.2, 9.2, 9.2.1, 9.4, 9.7]
* MyClass y MyBase no son independientes y se desplazan a la producción de expresiones calificadas. [11.4.3, 11.6]
* TypeOf…Is forma parte de la categoría principal de prioridad de operadores. [11.12.1]
* Las cláusulas Handles pueden incluir más de un identificador, pero solo son válidos dos identificadores. [9.2.6]
* La compilación condicional solo admite un subconjunto de expresiones constantes. [3.1]
* Se corrigen las referencias de System.Monitor a System.Threading.Monitor. [10.4]
* Se aclara que un compilador puede incluir declaraciones en un espacio de nombres concreto de forma predeterminada. [6.4]
* Dentro de un bloque Finally no se puede volver a iniciar una excepción (Throw sin ningún argumento). [10.10.1.3]
* La conversión de un tipo a sí mismo se considera ampliación. [8.8]
* Se intenta aclarar el comportamiento exacto de DirectCast. [11.11]
* El tipo de elemento de una colección en una instrucción For Each no necesita una conversión implícita a la expresión o variable de control de bucle; la conversión puede ser de cualquier clase. [10.9.3]
* Se aclara el comportamiento de la división con números decimales. [11.13.6]
* Se aclara que los operadores sobrecargados no se tienen en cuenta al convertir Nothing en una cadena vacía para el operador &. También se aclara que se aplica el mismo comportamiento al operador Like. [11.15, 11.16]
* Se aclara que las operaciones con parámetros Object pueden dar como resultado algo distinto de Integer. [11.12.2]
* Se agregan tablas de tipos de operadores explícitos. [11.12.3, 11.13.1, 11.13.2, 11.13.3, 11.13.4, 11.13.5, 11.13.6, 11.13.7, 11.13.8, 11.14, 11.15, 11.16, 11.17, 11.17.1, 11.18]
* Se intenta que la intención de las reglas de tipo “más específico” resulte más clara. [11.8.1]
* Se aclara que el sombreado de un método ParamArray por nombre y signatura solo sombrea dicha signatura, incluso si el método de sombreado coincide con la signatura no expandida del método ParamArray. [4.3.3]
* Se mueve la regla acerca de la preferencia de menos coincidencias de paramarray en lugar de más durante la resolución de sobrecarga a una ubicación anterior en el proceso para adaptarse al comportamiento del compilador (y deseado). [11.8.1, 11.8.2]
* Los inicializadores de tamaño de matriz y de elemento de matriz no se pueden combinar. [9.6.3.3]
* Al especificar límites en una expresión de creación de matriz y proporcionar un inicializador de elemento de matriz, los límites se deben especificar mediante expresiones constantes. [9.6.3.4]
* Se agrega un tema acerca de las conversiones boxing y unboxing, así como las limitaciones del suavizado de contorno en el diseño de tipos con conversión boxing. [8.6]
* Se intenta aclarar una regla críptica acerca de los tipos enumerados y los bucles For con variables de control de bucle de Object. [10.9.2]
* Se aclara que la modificación de una variable de control de bucle durante un bucle de Object no cambia el tipo del bucle. [10.9.2]
* Se indica que los delegados y los métodos externos pueden usar nombres de parámetro duplicados. [9.2.5]
* Las interfaces tienen una conversión de ampliación a Object. [8.4, 8.8]
* Las interfaces no heredan de Object. [7.8.1]
* Object es un tipo de referencia. No es un tipo que "no es un tipo de referencia ni un tipo de valor". [7]
* Se indica la posición de System.ValueType y System.Enum en la jerarquía de tipos de valor. [7.1]
* Se describen las conversiones de tipos primitivos que se permiten cuando se aplica una conversión boxing como Object. [8.6]
* Se amplía la explicación de los miembros de un delegado. [7.10]
* Se amplía el tema de las variables locales implícitas. [10.1.1, 10.2.1]
* Se aclara cómo funcionan los inicializadores estáticos. [10.2]
* Se indican los nombres sintéticos que se omiten en el enlace de nombres. [9.4, 9.4.1, 9.7.1, 9.7.2]
* Las excepciones detectadas en los bloques Try…Catch almacenan sus excepciones en el objeto Err. [10.10.1.2]
* Se describe la presencia de la conversión de identidad. [8]
* Se aclara cómo se tratan los accesos enlazados en tiempo de ejecución en los contextos de expresión. [11.1, 11.1.1, 11.3, 11.6, 11.8.1, 11.9]
* Se corrigen las reglas que indican cuándo se ejecutan los constructores compartidos. [9.3.2]

### Varios

* Se cambian las referencias al nombre del lenguaje de “Microsoft Visual Basic .NET” a “Microsoft Visual Basic”. El nombre oficial del lenguaje es “Visual Basic”.
* Los signos de puntuación y operadores de varios tokens como <= o >>= se mueven a la gramática léxica para mejorar la claridad. [2.5, 2.6, 5.2.2, 10.6.2, 10.8.2, 11.14]
* Se quita la producción de NamespaceOrTypeName porque no era realmente necesaria. [4.7, 6.3.1, 6.3.2, 7]
* Se quitan las producciones de variable local porque eran superfluas. [10.2]
* Se consolidan todas las producciones que solo incluían modificadores de acceso y Shadows en una única producción. [7, 7.4, 7.5, 7.6, 7.7, 7.8, 7.10]
* Con la llegada de un archivo de respuesta predeterminado para la línea de comandos, todos los proyectos importarán System de forma predeterminada, por lo que se quita de todos los ejemplos. [Demasiadas referencias para incluirlas todas]
* Se cambia el sufijo de las producciones de instrucciones de preprocesamiento de Element a Statement y el prefijo de las instrucciones de compilación condicional de Conditional a CC. [3.1, 3.1.1, 3.1.2, 3.2, 3.3]
* Se corrige el ejemplo de la sección de la instrucción Resume. [10.10.2.3]
* Se agrega Protected Friend a la producción del modificador. [4.6]
* Se agregan algunos ejemplos a las expresiones de creación de matriz. [11.10.2]

## Versión 8.0 a versión 8.0 (2nd Edition)

En esta actualización no existen cambios de lenguaje, solo incluye correcciones y erratas.

### Cambios menores

* Ahora se permite el uso de WithEvents en un tipo que no genera eventos. [9.6.2]
* Los errores en el número de dimensiones de ReDim se detectan en tiempo de compilación, no en tiempo de ejecución. [10.12.1]
* Las expresiones constantes no se pueden enlazar por medio de un contexto With. [11.2]
* El tipo de los parámetros Optional no se puede declarar como un parámetro de tipo sin restricciones. [9.2.5.3]
* El constructor predeterminado de una clase MustInherit es Protected, no Public. [9.3.1]
* TypeOf…Is produce un error en tiempo de compilación si la expresión nunca puede ser del tipo especificado. [11.5.2]
* Los constructores predeterminados llamarán a InitializeComponent si se emiten en una clase generada por el diseñador. [9.3.1]
* No se permite el acceso a los miembros de tipos de valor por medio de restricciones de tipo. [4.9.2]
* Se agrega el comportamiento especial en las conversiones boxing y unboxing de System.Nullable(Of T). [8.6]
* Un método puede resultar más aplicable que otro aunque los tipos de parámetro no se conviertan entre sí, siempre que al menos un tipo de parámetro resulte más aplicable. [11.8.1]
* Si se especifica una restricción Structure, se buscan miembros en System.ValueType. [4.9.2]
* No se permite la circularidad en la definición de estructuras. [7.6.1]

### Aclaraciones y erratas

* Se actualizan SubSignature y FunctionSignature para incluir las palabras clave Sub y Function de forma que aparezcan en las producciones de delegado. [9.2.1]
* Se agrega InvocationExpression a la producción de Expression. [11]
* Se quita la producción de VariableInitializer duplicada. [9.6.3.4]
* Se quita una indicación incorrecta acerca de que las instrucciones Catch pueden especificar ahora Object como tipo de excepción en una cláusula catch. [10.10.1.2]
* Se corrige el lenguaje acerca de la interacción entre los tipos genéricos y los miembros con acceso Protected. [4.6]
* Se corrige la producción de TypeArityList. [11.5.1]
* Se aclara que los tipos de valores con conversión boxing no se copian cuando se encuentran dentro de cualquier tipo, no solo de un tipo de valor con conversión boxing. [8.6]
* Se corrige la producción de ForEachStatement. [10.9.3]
* Se corrige la producción de ConversionOperatorModifier. [9.8.3]
* Se corrige la producción de MemberAccessExpression para incluir posibles parámetros genéricos. [11.6]
* Los desplazamientos de métodos externos se especifican con #, no con @. [9.2.2]
* Se quita la producción innecesaria de RegularInitializer. [9.6.3, 9.6.3.1]
* Se reúnen las producciones sintácticas de operador en una única producción, ya que la sintaxis real es más flexible que la implícita. [9.8, 9.8.1, 9.8.2, 9.8.3]
* Se aclara el uso de las expresiones booleanas en el lenguaje. [10.8.1, 10.9.1, 10.9.2, 10.10.1.2]
* Se indica que solo se puede llamar a los constructores Protected desde los constructores del tipo o los constructores de tipos derivados. [4.6]
* Se indica que las instancias predeterminadas solo se pueden crear mediante código generado por el compilador. [11.6.2.2]
* Se simplifican las producciones de la sintaxis de Imports. [6.3, 6.3.1, 6.3.2]
* Las reglas acerca de los tipos especiales que no se encuentran en el montón incluyen las llamadas a métodos de instancia de Object y System.ValueType. [7.13]
* Se aclara que las signaturas de ParamArray no se expanden cuando se realiza la ocultación de nombres. [4.3.3]
* EnumDeclaration toma NonArrayTypeName en lugar de tomar solo QualifiedName. [7.4]
* Se aclara la forma en que se asignan los operadores sobrecargados dentro y fuera de Visual Basic. [9.8.1, 9.8.2, 9.8.4]
* Se aclara cómo funciona el enlace con lenguajes que distinguen entre mayúsculas y minúsculas y declaraciones que solo se diferencian por el uso de mayúsculas y minúsculas. [4.1]

### Varios

* Se revisan todos los ejemplos de código para comprobar que son correctos y se actualizan cuando resulta necesario.

## Versión 8.0 (2nd Edition) a versión 9.0

Esta actualización trata los cambios de lenguaje realizados entre las versiones 8.0 y 9.0 del lenguaje.

### Cambios importantes

* Se agregan métodos de extensión. [9.2.7, 11.6, 11.8]
* Se agregan métodos parciales. [9.2.1, 9.2.8, 11.8, 11.10.3, 12.1]
* Se agrega la inicialización de miembros a las expresiones de creación de objetos. [9.6, 9.6.3.2, 9.6.3.4, 11.10.1]
* Se agrega la inferencia de tipo de variable local. [6.2, 6.2.2, 6.2.5, 10.2, 10.2.1, 10.9.2, 10.9.3, 10.13]
* Se agrega la construcción de tipos anónimos. [11.10.1, 11.10.4]
* Se agregan expresiones lambda y cierres. [7.13, 10.10.2.2, 11, 11.1, 11.1.1, 11.5.2, 11.8.1, 11.8.5, 11.10.3, 11.20, 11.20.1]
* Se agregan árboles de expresión. [11.1.1, 11.8.1, 11.8.5]
* Se permite que las expresiones de creación de delegados creen delegados a partir de firmas que no coinciden exactamente. [11.10.3]
* Se agregan tipos de valor que aceptan valores NULL. [2.5, 7, 7.1.1, 7.13, 8.6.1, 8.8, 8.9, 8.11, 9.2.5, 9.6, 9.6.3.2, 9.8, 9.8.3, 10.2, 10.9.2, 11.5.3, 11.10.3, 11.12.3, 11.16, 11.17, 11.19, 11.20]
* Se agregan expresiones de consulta. [11, 11.21, 11.21.1, 11.21.2, 11.21.3, 11.21.4, 11.21.5, 11.21.6, 11.21.7, 11.21.8, 11.21.9, 11.21.10, 11.21.11, 11.21.12, 11.21.13, 11.21.14]
* Se agrega el operador condicional. [3.1, 11, 11.2, 11.22]
* Se agregan importaciones de espacios de nombres XML. [6.3, 6.3.3, 11.5, 11.5.4]
* Se agregan expresiones literales XML. [2, 2.3, 11, 11.23, 11.23.1, 11.23.2, 11.23.3, 11.23.4, 11.23.5, 11.23.6, 11.23.7, 11.23.8]
* Se agregan expresiones de acceso a miembros XML. [11, 11.10.4, 11.24]

### Cambios menores

* Se agrega una sección que aclara el rol de las conversiones nativas. [8.12, 11.11]
* Ahora el lenguaje tiene en cuenta el atributo InternalsVisibleToAttribute. [4.5.1, 4.6, 7.5.2, 7.7.1, 7.12, 9.2.3]
* Ahora las clases de grupo pueden recopilar a partir de varios tipos, incluidos los tipos genéricos. [11.6.2.2]
* Los miembros de Object están disponibles en instancias de interfaces, pero se sombrean de forma implícita en los miembros de las interfaces. [7.8.2]
* Ahora se puede comparar la especificidad de los tipos de delegado en función de los tipos de valores devueltos en la resolución de sobrecarga. [11.8.1]
* Los argumentos de tipo, ya sean inferidos o no, deben satisfacer las restricciones para que un método se pueda aplicar en la resolución de sobrecarga. [11.8.2]
* Se amplía la inferencia de los argumentos de tipo para permitir que se resuelvan varias inferencias de un argumento de tipo. [8.13, 11.8.5]
* Los argumentos de tipo se pueden inferir a partir del tipo de valor devuelto de una expresión AddressOf. [11.8.5]

### Aclaraciones y erratas

* Se cambia la terminología del destino de un grupo de métodos o propiedades para que aparezca como “destino” en lugar del concepto más confuso “expresión de instancia asociada”. [10.6.1, 11.1, 11.1.1, 11.3, 11.4.5, 11.6, 11.8, 11.9, 11.10.3]
* Se aclara que la inferencia de tipos se realiza antes de la resolución de sobrecarga. [11.8.1]
* Se aclara cómo los parámetros ParamArray participan en la resolución de sobrecarga. [11.8.1, 11.8.2]
* Se agrega una regla no incluida por la que se prefieren más métodos derivados que métodos base con firmas idénticas en la resolución de sobrecarga. [11.8.1]
* Se corrigen las reglas para elegir métodos “menos genéricos” en la resolución de sobrecarga. [11.8.1]
* Se aclara que los inicializadores normales solo pueden inicializar una variable a la vez. [9.6.3.1]
* Se corrige el enlace de nombre simple y nombre calificado para que muestre el comportamiento adecuado en presencia de los argumentos de tipo. [4.7.1, 4.7.2, 11.4.4, 11.6]
* Se quita la regla falsa según la cual no se pueden crear delegados a partir de métodos declarados como MustOverride. [11.10.3]
* Se aclara la terminología con respecto al enlace en tiempo de ejecución y AddressOf. [11.4.5]
* Se corrigen errores en el control de matrices en la inferencia de argumentos de tipo. [11.8.5]
* Se aclara la inferencia de argumentos de tipo mediante tipos construidos. [11.8.5]
* Se aclaran las reglas relacionadas con las conversiones de referencia. [8.4]
* Se aclaran o corrigen las reglas relacionadas con las restricciones de parámetros de tipo. [8.8, 8.9, 8.10, 8.12]
* Se aclara que un nombre de variable con un inicializador de objetos no puede tener un modificador de tipo de matriz. [9.6.3.2]
* Se aclara el comportamiento del operador de concatenación. [11.16]
* Se aclara o corrige el comportamiento de las expresiones booleanas. [11.19]
* Se corrige el tipo de resultado incorrecto de los operadores en valores Boolean. [11.13.1, 11.13.2, 11.13.3, 11.13.4, 11.13.5, 11.13.6, 11.13.7, 11.18]
* Se agrega IsNot a la tabla de prioridad de operadores, con la misma prioridad que Is. [11.12.1]
* Se corrige la producción de CatchStatement para indicar que la cláusula As puede ser opcional. [10.10.1.2]
* Se corrige la producción de MemberAccessExpression, que ya no indica que el punto es opcional. [11.6]
* Se corrige la gramática de las expresiones de compilación condicional. [3.1]
* Se rehace la sección de restricciones de tipo para aclarar y permitir la covarianza de matrices para satisfacer las restricciones de tipo. [4.9.2]
* Se corrige que no se puede usar = ni <> en Object con Option Strict, pero sí se puede usar IsNot. [6.2.2]
* Se indica que se puede usar Date en expresiones constantes. [11.2]

### Varios

* Se quita la producción innecesaria de DelegateCreationExpression. [11.10, 11.10.3]
* Se cambia el término de “predefinida” a “intrínseca”. [8.11, 8.11.1, 8.11.2, 11.12.3, 11.15]
* Se quitan todas las referencias a “foo”, “bar” y “baz” en los ejemplos.

## Versión 9.0 a versión 10.0

### Cambios importantes

* Se agregan propiedades implementadas automáticamente. [9.7, 9.7.4]
* Se permiten las continuaciones de línea implícitas en algunos contextos. [2.1.2, 4.9.1, 4.9.2, 5.2, 5.2.2, 6.3, 6.3.1, 6.3.3, 7.2, 7.4.1, 7.8.1, 7.9, 7.12, 9.1, 9.2.1, 9.2.2, 9.2.5, 9.2.6, 9.3, 9.4, 9.4.1, 9.5, 9.6, 9.6.3.3, 9.6.3.4, 9.7, 9.7.2, 9.8, 10.5.1, 10.5.2, 10.6.1, 10.6.3, 10.8.2, 10.9.2, 10.9.3, 10.12.1, 10.12.2, 11.4.2, 11.4.4, 11.5.1, 11.5.2, 11.5.3, 11.5.4, 11.6, 11.8, 11.9, 11.10.1, 11.11, 11.13.3, 11.13.4, 11.13.5, 11.13.6, 11.13.7, 11.13.8, 11.14, 11.15, 11.16, 11.17, 11.17.1, 11.18, 11.20, 11.21.1, 11.21.4, 11.21.5, 11.21.6, 11.21.7, 11.21.8, 11.21.9, 11.21.10, 11.21.11, 11.21.12, 11.21.13, 11.21.14, 11.22, 11.23.2]
* Se agregan inicializadores de colección y literales de matriz. [9.6, 9.6.3, 9.6.3.3, 9.6.3.4 (eliminada), 9.7, 11.1, 11.1.1, 11.10, 11.10.1, 11.10.2, 11.10.4, 11.10.5]
* Se agregan expresiones lambda de varias líneas que pueden contener instrucciones e instrucciones lambda de línea única. [11.20]
* Se agrega la varianza genérica. [4.9.1, 4.9.3, 7.11, 8.4.1, 8.8, 8.9]

### Cambios menores

* Al enlazar nombres de tipos, los tipos inaccesibles se omiten. [4.7.1, 4.7.2, 11.4.4]
* Ya no es necesario que las estructuras tengan al menos una variable de instancia. [7.6.1]
* Ahora los parámetros opcionales pueden ser estructuras, tipos de referencia y parámetros de tipo sin restricciones. [9.2.5.3]
* La aplicación especial de paréntesis al valor devuelto de un método se restringe al caso en que solo existe un método accesible. [11.8]
* Las variables de control de bucle con un identificador definirán un nuevo identificador local si se usa la inferencia de tipos de variable local y el identificador no está definido o ser resuelve en un tipo [10.9.2, 10.9.3, 11.4.4]

### Aclaraciones y erratas

* Los operadores y los signos de puntuación de varios tokens se devuelven a la gramática sintáctica, ya que les afectan las reglas de espacio en blanco. [2.5, 2.6, 3.1, 5.2.2, 9.8, 10.6.2, 10.8.2, 11.8, 11.14, 11.18]
* Se corrige la producción de los operadores de comparación en instrucciones Select. [10.8.2]
* Se corrigen varios errores en la producción de propiedades. [9.7]
* Se corrige un error en la producción de literales de fecha. [2.4.6]
* Se corrige un error en la producción de referencias de caracteres XML. [11.23.4]
* Se corrige la afirmación que señala que los descriptores de acceso de propiedades pueden tener el mismo acceso que su propiedad contenedora. [9.7]
* Se corrige la ortografía de Wend y Variant en la tabla de palabras clave. [2.3]
* Se aclara que la reclasificación de los punteros de método y los métodos lambda tiene lugar en cualquier lugar donde se realiza una conversión. [11.1.1]
* Se corrige la afirmación que indica que los métodos lambda sin un tipo contextual se convierten en Func; en realidad se convierten en un tipo lambda anónimo que se puede convertir a tipos como Func. [11.1.1]
* Se quitan las producciones falsas de XMLFragment y XmlNames. [11.23, 11.23.1]
* Se corrige la producción de XMLAttribute para comenzar con espacio en blanco. [11.23.4]
* Se aclara que las variables locales declaradas de forma implícita en las expresiones lambda se declaran en el método que contiene la expresión y no en la propia expresión. [10.2.1]
* Se corrigen las producciones de nombre de tipo, de forma que los nombres de tipos genéricos se tratan correctamente. [4.7, 6.4.1, 7, 7.12, 9.2.6, 11.5.1]
* Se mejoran las descripciones y clasificaciones de las conversiones. [8, 8.1, 8.3, 8.4, 8.5, 8.8, 8.9, 8.12, 11.5.2]
* Se corrige el hecho de que la covarianza de matrices no solo funciona para la herencia y la implementación. [8.5, 8.8, 8.9]
* Se corrige la producción de XMLEncodingNameCharacter para incluir el carácter de guion. [11.23.3]
* Se aclara que puede producirse la asignación al grupo interno de cadenas (o puede que no). [2.4.4]
* Se agrega una sección sobre las conversiones de delegados anónimos. [8.4.2, 8.8, 8.9]
* Se define el concepto “conversión intrínseca”. [8.11]
* Se quita la indicación incorrecta de que las líneas que se excluyen en la compilación condicional deben ser léxicamente válidas. [3.1]
* Se aclara que los tipos de valores devueltos de un método invalidado y el método de invalidación deben coincidir. [4.5.1]
* Se quita la afirmación incorrecta de que solo se puede llamar a los constructores Protected desde los constructores de un tipo o los constructores de tipos derivados. [4.6]
* Se aclara que las estructuras satisfacen una restricción New. [4.9.1]
* Se aclara que las matrices implementan IList(Of T) en los tipos cuyo tipo de elemento tiene una identidad de ampliación o una conversión de referencia. [7.9]
* Se aclara que en los tipos enumerados con conversión boxing se puede aplicar una conversión unboxing a su tipo subyacente. [8.6]
* Se aclara que solo se pueden usar conversiones intrínsecas en los valores de parámetros opcionales. [9.2.5.3]
* Se aclara que las declaraciones locales se realizan de forma implícita si el nombre no se resuelve en nada más. [10.2.1]
* Se corrige la gramática de las listas de argumentos. [5.2.2, 11.8]
* Se aclara el uso de IDispatchConstantAttribute e IUnknownConstantAttribute en la coincidencia de listas de argumentos. [11.8.2.]
* Se corrige la dirección de la conversión de los tipos de parámetros al establecer correspondencias de firmas de expresiones de creación de delegado. [11.10.3]
* Se intenta aclarar la posición del valor NULL en el lenguaje. [7.1, 7.3, 8.4, 8.6, 10.10.1.3, 10.12.1, 11.1, 11.1.1, 11.4.1, 11.8.2, 11.14]
* El tipo de la expresión de una instrucción TryCast no tiene que ser un tipo de valor. [11.11]
* Else If permite un espacio. [2.5, 3.2, 10.8.1]
* El preprocesador permite un espacio en blanco. [3.1]
* Las expresiones de cadena constantes se evalúan en tiempo de compilación. [2.4.4]
* Se aclara la anidación de directivas Region. [3.3]
* Se aclara que los métodos de implementación deben coincidir con el tipo devuelto. [4.4.1]
* Se aclara que los tipos enumerados no pueden declarar parámetros de tipo. [4.9]
* Se aclara qué combinaciones incoherentes de restricciones de tipos no se permiten. [4.9.2]
* La clase Attribute no necesita AttributeUsage explícito. [5.1]
* Los parámetros de atributo posicionales pueden ser Optional o ParamArray. [5.2]
* IdentifierModifiers ? y () se pueden combinar. [7, 9.2.5, 9.6, 10.9.2]
* Las declaraciones de delegado pueden tener modificadores y atributos [7.10]
* Las declaraciones de tipos parciales anidadas pueden estar en contenedores no anidados. [7.12]
* Se aclaran las conversiones booleanas intrínsecas. [8.2]
* Las cláusulas Handles permiten la relajación de delegados sin argumentos y admiten MyClass y MyBase cuando especifiquen eventos. [9.2.6, 11.4.3]
* Las instrucciones deben comenzar con una palabra clave o identificador. [10]
* Siempre se usa la inferencia para las declaraciones de constantes locales que no tienen tipos explícitos. [10.2]
* Se aclara el tratamiento de las expresiones booleanas. [10.8.1, 10.9.1, 10.10.1.2, 11.19]
* Las instrucciones Next múltiples se combinan en una. [10.9.2, 10.9.3]
* Los modelos de colección coinciden con los métodos de extensión. [10.9.3]
* La expresión "Nothing" tiene la clasificación "default value" y puede reclasificarse. [11.1.1, 11.4.1]
* Hay límites en los tipos de lambdas que pueden convertirse en árboles de expresión. [11.1.1]
* Object es un tipo válido para una expresión constante. En las constantes se permiten las conversiones en y de la cadena null. Se aclara que las operaciones en constantes deben dar como resultado un tipo válido para una constante. [11.2]
* Se aclara el orden de evaluación de los argumentos de invocación. [11.3, 11.8]
* TypeArityList requiere paréntesis y "Of". [11.5.1]
* Las expresiones de acceso a miembros funcionan en tipos que admiten valores null. [11.6]
* E.New se permite cuando E es una enumeración. [11.6]
* Los operadores de igualdad y desigualdad no se permiten en Object con Option Strict On. [11.12.2]
* Se aclara el comportamiento de AndAlso y OrElse para valores booleanos que admiten valores null. [11.17.1]
* Los operadores Join y GroupJoin permiten operadores de consulta Join y GroupJoin anidados. [11.21, 11.21.5, 11.21.14]
* Solo el operador de consulta Let puede especificar el tipo de su expresión de variable de intervalo de expresión. [11.21]
* La cláusula "Group" del operador de consulta "Group By" es opcional. [11.21.12]

## Versión 10.0 a versión 11.0

### Cambios importantes

* Compatibilidad agregada para los métodos Async e Iterator y las palabras clave Await y Yield [6.1, 9.2.1, 9.7.5, 10, 10.1, 10.1.1-10.1.3, 10.10.1.1, 10.10.1.2, 10.10.2, 10.11, 10.14, 10.15, 11.1, 11.1.1, 11.8.5, 11.12.1, 11.20, 11.25]
* Compatibilidad agregada para los atributos CallerLineNumber, CallerFilePath y CallerMemberName en parámetros opcionales; se aclara el tratamiento especial alternativo específico de Microsoft de ciertos parámetros opcionales [11.8.2]
* Ahora los espacios de nombres pueden declararse explícitamente para incluirse en el espacio de nombres global [6.4.1]

### Cambios menores

* Se ha agregado una nueva regla de desempate a la resolución de sobrecarga: si se puede llamar a una sobrecarga sin tener que sustituir ningún valor predeterminado de parámetro opcional, y sin embargo a otra no, elija la primera [11.8.1]
* Ahora For Each crea una copia reciente de la variable de iteración en cada bucle de la iteración [10.9.3]
* Ahora el compilador permite que se declaren métodos que difieren solo en los parámetros opcionales, siempre que sus signaturas sean distintas [9.2]
* Se ha agregado una nueva regla de desempate a la resolución de sobrecarga: entre dos sobrecargas genéricas aplicables, elija aquella cuyo parámetro de tipo genérico sea “más profundo” [11.8.1.3]
* Compatibilidad agregada para IReadOnlyList e IReadOnlyCollection [7.9, 8.8, 8.9, 11.1.1]
* La generación de código de los eventos en los archivos winmd es ligeramente distinta [9.4.1]
* La resolución del operador binario entre Object y otro tipo, si no hay conversiones intrínsecas o definidas por el usuario, se enlazará en tiempo de ejecución en lugar de generar un error [11.12.3]

### Aclaraciones y erratas

* Se corrigen las reglas de resolución de sobrecarga: se cambia el nombre de "aplicabilidad" a "especificidad"; si un candidato de sobrecarga es una coincidencia exacta, selecciónelo sin tener en cuenta otras reglas de resolución de sobrecarga; se corrigen las reglas para la prioridad de especifidad entre tipos numéricos intrínsecos; se aclara que Option Strict se usa en la resolución de sobrecarga; se corrigen las reglas por las que se usa el enlace en tiempo de ejecución; se aclara que los métodos de instancia narrowing no excluirán los métodos de extensión; se agrega un análisis considerable sobre los niveles de relajación de delegados que se realizan al inicio de la resolución de sobrecarga, en lugar de como desempate final; se aclara la descripción del desempate de parámetro de tipo entre dos candidatos de método de extensión; se corrige la descripción del desempate relacionado con Sub/Función entre expresiones AddressOf y tipos de delegado; se aclara que la aplicabilidad también considera las restricciones genéricas [11.8.1, 11.8.2]
* Se corrigen las reglas sobre qué tipo numérico es más específico que otro en la resolución de sobrecarga y se describe el desempate final adicional relacionado con la especificidad de los tipos de parámetro [11.8.1.1]
* Se aclara que la palabra clave “Key” en FieldInitializer solo se permite en las expresiones de creación de objetos anónimos [11.10.1]
* Se ha corregido el ejemplo de código para la expresión de objeto anónimo [11.10.4]
* Se aclara que las instrucciones de invocación con listas de argumentos siempre tratarán a estos como argumentos de la invocación; además, las instrucciones de invocación deben comenzar con un identificador [10.7]
* Se aclara que el modelo de colección puede enlazarse a métodos o propiedades compartidos, y los inicializadores de colección pueden enlazarse a métodos Add compartidos [10.9.3, 11.10.1]
* Se aclaran las reglas de búsqueda de nombres en las instrucciones “For *identificador* =” y “For Each *identificador* In” [10.9.2, 10.9.3]
* Se aclara que los espacios de nombres anidados son “lógicos” en lugar de “léxicos” [4.7]; además, se aclaran las reglas para búsqueda de nombres de espacio de nombres o de tipos calificados o no calificados [4.7.1, 4.7.2]
* Se corrigen las reglas de búsqueda de métodos de extensión: el compilador busca los métodos de extensión en los espacios de nombres importados antes de buscar en los espacios de nombres envolventes [11.6.3]
* Se aclara que los tipos genéricos pueden limitarse a los tipos anidados dentro del tipo en el que se definen [7.5.2, 7.6.1, 7.8.2]
* Se aclaran las reglas para los alias de Imports: hacen referencia al tipo o el nombre de espacio de nombres en sí, en lugar de usar las reglas para resolución de nombres calificados; además, no pueden ser tipos genéricos abiertos [6.3.1]
* Se aclara una regla para la resolución de sobrecarga: si un candidato de sobrecarga es una coincidencia exacta, selecciónelo sin tener en cuenta otras reglas de resolución de sobrecarga [11.8.1]
* Se aclaran las reglas para las conversiones de matriz basadas en el tipo subyacente de la enumeración y para IList, IReadOnlyList, ICollection, IReadOnlyCollection, IEnumerable [7.9, 8.5, 8.8, 8.9]
* Se describe una ambigüedad introducida por la covarianza y la contravarianza [4.9.3]
* Se aclaran las reglas para la conversión de lambdas en tipos no delegados, incluidos Object, Delegate y MulticastDelegate [8.4.2]
* Se aclara la lista de conversiones widening y narrowing [8.8, 8.9]
* Se aclara que un método puede ser MustOverride y Overrides [9.2.3]
* Se corrige la gramática de GetTypeExpression [11.5.1]
* Se corrige la gramática de MemberAccessBase: no incluye MyBase o Myclass [11.6]
* Se corrigen las reglas de prioridad de aplicabilidad entre tipos numéricos intrínsecos, en la resolución de sobrecarga [11.8.1]
* Se permite una continuación de línea implícita después del punto en [xml.@attr](mailto:xml.@attr) [11.24]
* Se aclara que no se pueden invocar los métodos add\_X y remove\_X en los eventos [9.4]
* Se corrigen las reglas sobre qué tipo numérico tiene mayor aplicabilidad que otro en la resolución de sobrecarga [11.8.1.1]
* Se ha corregido el ejemplo de tipo anónimo [11.10.4]
* Se aclara el comportamiento de cortocircuito de los operadores lógicos con respecto a IsTrue e IsFalse [11.17.1]
* Se aclara exactamente cuando se crean las variables locales dentro de un bucle [10.1.5, 10.9, 10.9.2, 10.9.3]
* Se aclara la definición de "signatura de método"; no incluye el nombre del método [4.1.1]
* Se aclaran las reglas por las que los argumentos ByRef se realizan mediante el uso de alias o las opciones de copiar [9.2.5.2]
* Los miembros de tipo constante usan la inferencia de tipos [9.5]
* Se indican los casos en los que GoTo no se permite en determinados bloques [10.11]
* Se aclaran las reglas según las cuales la resolución de operador puede enlazarse en tiempo de ejecución [11.12.2, 11.12.3]
* Se aclaran las reglas de tipo de los parámetros opcionales que aceptan valores NULL [9.2.5.3]
* Se aclara que solamente Me (y no MyClass o MyBase) es una expresión de instancia válida; MyClass y MyBase solamente se permiten en expresiones de acceso a miembros y en cláusulas Handles [9.2.6, 11.4.3, 11.6]
* Se aclara que, en la reclasificación de expresiones, se aplican las consideraciones de si un elemento es widening o narrowing; en concreto, la reclasificación de lambda con tipos de parámetros más restringidos es narrowing, ya que se trata de una reclasificación de literales de matriz a tipos de elementos de matriz más restringidos [11.1.1]