The Microsoft® Visual Basic® Language Specification

Version 11.0

Paul Vick, Lucian Wischik
Microsoft Corporation

Table of Contents

1. Introduction

The Microsoft® Visual Basic® programming language is a high-level programming language for the Microsoft .NET Framework. Although it is designed to be an approachable and easy-to-learn language, it is also powerful enough to satisfy the needs of experienced programmers. The Visual Basic programming language has a syntax that is similar to English, which promotes the clarity and readability of Visual Basic code. Wherever possible, meaningful words or phrases are used instead of abbreviations, acronyms, or special characters. Extraneous or unneeded syntax is generally allowed but not required.

The Visual Basic programming language can be either a strongly typed or a loosely typed language. Loose typing defers much of the burden of type checking until a program is already running. This includes not only type checking of conversions but also of method calls, meaning that the binding of a method call can be deferred until run-time. This is useful when building prototypes or other programs in which speed of development is more important than execution speed. The Visual Basic programming language also provides strongly typed semantics that performs all type checking at compile-time and disallows run-time binding of method calls. This guarantees maximum performance and helps ensure that type conversions are correct. This is useful when building production applications in which speed of execution and execution correctness is important.

This document describes the Visual Basic language. It is meant to be a complete language description rather than a language tutorial or a user's reference manual.

1.1 Grammar Notation

This specification describes two grammars: a lexical grammar and a syntactic grammar. The lexical grammar defines how characters can be combined to form tokens; the syntactic grammar defines how the tokens can be combined to form Visual Basic programs. There are also several secondary grammars used for preprocessing operations like conditional compilation.

The grammars in this specification are written in ANTLR format -- see http://www.antlr.org/.

Case is unimportant in Visual Basic programs. For simplicity, all terminals will be given in standard casing, but any casing will match them. Terminals that are printable elements of the ASCII character set are represented by their corresponding ASCII characters. Visual Basic is also width insensitive when matching terminals, allowing full-width Unicode characters to match their half-width Unicode equivalents, but only on a whole-token basis. A token will not match if it contains mixed half-width and full-width characters.

Line breaks and indentation may be added for readability and are not part of the production.

1.2 Compatibility

An important feature of a programming language is compatibility between different versions of the language. If a newer version of a language does not accept the same code as a previous version of the language, or interprets it differently than the previous version, then a burden can be placed on a programmer when upgrading his code from one version of the language to another. As such, compatibility between versions must be preserved except when the benefit to language consumers is of a clear and overwhelming nature.

The following policy governs changes to the Visual Basic language between versions. The term language, when used in this context, refers only to the syntactic and semantic aspects of the Visual Basic language itself and does not include any .NET Framework classes included as a part of the `Microsoft.VisualBasic` namespace (and sub-namespaces). All classes in the .NET Framework are covered by a separate versioning and compatibility policy outside the scope of this document.

1.2.1 Kinds of compatibility breaks

In an ideal world, compatibility would be 100% between the existing version of Visual Basic and all future versions of Visual Basic. However, there may be situations where the need for a compatibility break may outweigh the cost it may impose on programmers. Such situations are:

- *New warnings.* Introducing a new warning is not, per se, a compatibility break. However, because many developers compile with "treat warnings as errors" turned on, extra care must be taken when introducing warnings.
- *New keywords.* Introducing new keywords may be necessary when introducing new language features. Reasonable efforts will be made to choose keywords that minimize the possibility of collision with users' identifiers and to use existing keywords where it makes sense. Help will be provided to upgrade projects from previous versions and escape any new keywords.
- *Compiler bugs.* When the compiler's behavior is at odds with a documented behavior in the language specification, fixing the compiler behavior to match the documented behavior may be necessary.
- *Specification bug.* When the compiler is consistent with the language specification but the language specification is clearly wrong, changing the language specification and the compiler behavior may be necessary. The phrase "clearly wrong" means that the documented behavior runs counter to what a clear and unambiguous majority of users would expect and produces highly undesirable behavior for users.
- *Specification ambiguity.* When the language specification should spell out what happens in a particular situation but doesn't, and the compiler handles the situation in a way that is either inconsistent or clearly wrong (using the same definition from the previous point), clarifying the specification and correcting the compiler behavior may be necessary. In other words, when the specification covers cases a, b, d and e, but omits any mention of what happens in case c, and the compiler behaves incorrectly in case c, it may be necessary to document what happens in case c and change the behavior of the compiler to match. (Note that if the specification was ambiguous as to what happens in a situation and the compiler behaves in a manner that is not clearly wrong, the compiler behavior becomes the de facto specification.)
- *Making run-time errors into compile-time errors.* In a situation where code is 100% guaranteed to fail at runtime (i.e. the user code has an unambiguous bug in it), it may be desirable to add a compile-time error that catches the situation.
- *Specification omission.* When the language specification does not specifically allow or disallow a particular situation and the compiler handles the situation in a way that is undesirable (if the compiler behavior was clearly wrong, it would a specification bug, not a specification omission), it may be necessary to clarify the specification and change the compiler behavior. In addition to the usual impact analysis, changes of this kind are further restricted to cases where the impact of the change is considered to be extremely minimal and the benefit to developers is very high.
- *New features.* In general, introducing new features should not change existing parts of the language specification or the existing behavior of the compiler. In the situation where introducing a new feature requires changing the existing language specification, such a compatibility break is reasonable only if the impact would be extremely minimal and the benefit of the feature is high.
- *Security.* In extraordinary situations, security concerns may necessitate a compatibility break, such as removing or modifying a feature that is inherently insecure and poses a clear security risk for users.

The following situations are not acceptable reasons for introducing compatibility breaks:

- *Undesirable or regrettable behavior.* Language design or compiler behavior which is reasonable but considered undesirable or regrettable in retrospect is not a justification for breaking backward compatibility. The language deprecation process, covered below, must be used instead.
- *Anything else.* Otherwise, compiler behavior remains backwards compatible.

1.2.2 Impact Criteria

When considering whether a compatibility break might be acceptable, several criteria are used to determine what the impact of the change might be. The greater the impact, the higher the bar for accepting the compatibility breaks.

The criteria are:

- What is the scope of the change? In other words, how many programs are likely to be affected? How many users are likely to be affected? How common will it be to write code that is affected by the change?
- Do any workarounds exist to get the same behavior prior to the change?
- How obvious is the change? Will users get immediate feedback that something has changed, or will their programs just execute differently?
- Can the change be reasonably addressed during upgrade? Is it possible to write a tool that can find the situation in which the change occurs with perfect accuracy and change the code to work around the change?
- What is the community feedback on the change?

1.2.3 Language deprecation

Over time, parts of the language or compiler may become deprecated. As discussed previously, it is not acceptable to break compatibility to remove such deprecated features. Instead, the following steps must be followed:

1. Given a feature that exists in version *A* of Visual Studio, feedback must be solicited from the user community on deprecation of the feature and full notice given before any final deprecation decision is made. The deprecation process may be reversed or abandoned at any point based on user community feedback.
2. full version (i.e. not a point release) *B* of Visual Studio must be released with compiler warnings that warn of deprecated usage. The warnings must be on by default and can be turned off. The deprecations must be clearly documented in the product documentation and on the web.
3. A full version *C* of Visual Studio must be released with compiler warnings that cannot be turned off.
4. A full version *D* of Visual Studio must subsequently be released with the deprecated compiler warnings converted into compiler errors. The release of *D* must occur after the end of the Mainstream Support Phase (5 years as of this writing) of release *A*.
5. Finally, a version *E* of Visual Studio may be released that removes the compiler errors.

Changes that cannot be handled within this deprecation framework will not be allowed.

2. Lexical Grammar

Compilation of a Visual Basic program first involves translating the raw stream of Unicode characters into an ordered set of lexical tokens. Because the Visual Basic language is not free-format, the set of tokens is then further divided into a series of logical lines. A *logical line* spans from either the start of the stream or a line terminator through to the next line terminator that is not preceded by a line continuation or through to the end of the stream.

Note. With the introduction of XML literal expressions in version 9.0 of the language, Visual Basic no longer has a distinct lexical grammar in the sense that Visual Basic code can be tokenized without regard to the syntactic context. This is due to the fact that XML and Visual Basic have different lexical rules and the set of lexical rules in use at any particular time depends on what syntactic construct is being processed at that moment. This specification retains this lexical grammar section as a guide to the lexical rules of regular Visual Basic code.

```
LogicalLineStart:
    | LogicalLine*
    ;

LogicalLine:
    | LogicalLineElement* Comment? LineTerminator
    ;

LogicalLineElement:
    | WhiteSpace
    | LineContinuation
    | Token
    ;

Token:
    | Identifier
    | Keyword
    | Literal
    | Separator
    | Operator
    ;
```

2.1 Characters and Lines

Visual Basic programs are composed of characters from the Unicode character set.

```
Character:
    | Any Unicode character except a LineTerminator
    ;
```

2.1.1 Line Terminators

Unicode line break characters separate logical lines.

```
LineTerminator:
    | Unicode 0x00D
    | Unicode 0x00A
    | CR
    | LF
    | Unicode 0x2028
    | Unicode 0x2029
    ;
```

2.1.2 Line Continuation

A *line continuation* consists of at least one white-space character that immediately precedes a single underscore character as the last character (other than white space) in a text line. A line continuation allows a logical line to span more than one physical line. Line continuations are treated as if they were white space, even though they are not.

```
LineContinuation:
    | WhiteSpace '_' WhiteSpace* LineTerminator
    ;
```

The following program shows some line continuations:

```
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
```

Some places in the syntactic grammar allow for *implicit line continuations*. When a line terminator is encountered

- after a comma (`,`), open parenthesis (`(`), open curly brace (`{`), or open embedded expression (`<%=`)
- after a member qualifier (`.` or `.@` or `...`), provided that something is being qualified (i.e. is not using an implicit `With` context)
- before a close parenthesis (`)`), close curly brace (`}`), or close embedded expression (`%>`)
- after a less-than (`<`) in an attribute context
- before a greater-than (`>`) in an attribute context
- after a greater-than (`>`) in a non-file-level attribute context
- before and after query operators (`Where`, `Order`, `Select`, etc.)
- after binary operators (`+`, `-`, `/`, `*`, etc.) in an expression context
- after assignment operators (`=`, `:=`, `+=`, `-=`, etc.) in any context.

the line terminator is treated as if it was a line continuation.

```
Comma:
    | ',' LineTerminator?
    ;

Period:
    | '.' LineTerminator?
    ;

OpenParenthesis:
    | '(' LineTerminator?
    ;

CloseParenthesis:
    | LineTerminator? ')'
    ;

OpenCurlyBrace:
```

```
    | '{' LineTerminator?
    ;

CloseCurlyBrace:
    | LineTerminator? '}'
    ;

Equals:
    | '=' LineTerminator?
    ;

ColonEquals:
    | ':' '=' LineTerminator?
    ;
```

For example, the previous example could also be written as:

```
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
```

Implicit line continuations will only ever be inferred directly before or after the specified token. They will not be inferred before or after a line continuation. For example:

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

y
```

Line continuations will not be inferred in conditional compilation contexts. (**Note.** This last restriction is required because text in conditional compilation blocks that are not compiled do not have to be syntactically valid. Thus, text in the block might accidentally get "picked up" by the conditional compilation statement, especially as the language gets extended in the future.)

2.1.3 White Space

White space serves only to separate tokens and is otherwise ignored. Logical lines containing only white space are ignored. (**Note.** Line terminators are not considered white space.)

```
WhiteSpace:
    | Unicode class Zs
    | Unicode Tab 0x0009
    ;
```

2.1.4 Comments

A *comment* begins with a single-quote character or the keyword `REM`. A single-quote character is either an ASCII single-quote character, a Unicode left single-quote character, or a Unicode right single-quote character. Comments can begin anywhere on a source line, and the end of the physical line ends the comment. The compiler ignores the characters between the beginning of the comment and the line terminator. Consequently, comments cannot extend across multiple lines by using line continuations.

```
Comment:
    | CommentMarker Character*
    ;

CommentMarker:
    | SingleQuoteCharacter
    | 'REM'
    ;

SingleQuoteCharacter:
    | '\''
    | Unicode 0x2018
    | Unicode 0x2019
    ;
```

2.2 Identifiers

An *identifier* is a name. Visual Basic identifiers conform to the Unicode Standard Annex 15 with one exception: identifiers may begin with an underscore (connector) character. If an identifier begins with an underscore, it must contain at least one other valid identifier character to disambiguate it from a line continuation.

```
Identifier:
    | NonEscapedIdentifier TypeCharacter?
    | Keyword TypeCharacter
    | EscapedIdentifier
    ;

NonEscapedIdentifier:
    | Any IdentifierName but not Keyword
    ;

EscapedIdentifier:
    | '[' IdentifierName ']'
    ;

IdentifierName:
    | IdentifierStart IdentifierCharacter*
    ;

IdentifierStart:
    | AlphaCharacter
    | UnderscoreCharacter IdentifierCharacter
    ;

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

AlphaCharacter:
    | Unicode classes Lu,Ll,Lt,Lm,Lo,Nl
    ;

NumericCharacter:
    | Unicode decimal digit class Nd
    ;
```

```
CombiningCharacter:
    | Unicode combining character classes Mn, Mc
    ;

FormattingCharacter:
    | Unicode formatting character class Cf
    ;

UnderscoreCharacter:
    | Unicode connection character class Pc
    ;

IdentifierOrKeyword:
    | Identifier
    | Keyword
    ;
```

Regular identifiers may not match keywords, but escaped identifiers or identifiers with a type character can. An *escaped identifier* is an identifier delimited by square brackets. Escaped identifiers follow the same rules as regular identifiers except that they may match keywords and may not have type characters.

This example defines a class named `class` with a shared method named `shared` that takes a parameter named `boolean` and then calls the method.

```
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
```

Identifiers are case insensitive, so two identifiers are considered to be the same identifier if they differ only in case. (**Note.** The Unicode Standard one-to-one case mappings are used when comparing identifiers and any locale-specific case mappings are ignored.)

2.2.1 Type Characters

A *type character* denotes the type of the preceding identifier. The type character is not considered part of the identifier.

```
TypeCharacter:
    | IntegerTypeCharacter
    | LongTypeCharacter
    | DecimalTypeCharacter
    | SingleTypeCharacter
    | DoubleTypeCharacter
    | StringTypeCharacter
    ;

IntegerTypeCharacter:
    | '%'
```

```
    ;

LongTypeCharacter:
    | '&'
    ;

DecimalTypeCharacter:
    | '@'
    ;

SingleTypeCharacter:
    | '!'
    ;

DoubleTypeCharacter:
    | '#'
    ;

StringTypeCharacter:
    | '$'
    ;
```

If a declaration includes a type character, the type character must agree with the type specified in the declaration itself; otherwise, a compile-time error occurs. If the declaration omits the type (for example, if it does not specify an `As` clause), the type character is implicitly substituted as the type of the declaration.

No white space may come between an identifier and its type character. There are no type characters for `Byte`, `SByte`, `UShort`, `Short`, `UInteger` or `ULong`, due to a lack of suitable characters.

Appending a type character to an identifier that conceptually does not have a type (for example, a namespace name) or to an identifier whose type disagrees with the type of the type character causes a compile-time error.

The following example shows the use of type characters:

```
' 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
```

The type character `!` presents a special problem in that it can be used both as a type character and as a separator in the language. To remove ambiguity, a `!` character is a type character as long as the character that follows it cannot start an identifier. If it can, then the `!` character is a separator, not a type character.

2.3 Keywords

A *keyword* is a word that has special meaning in a language construct. All keywords are reserved by the language and may not be used as identifiers unless the identifiers are escaped. (**Note.** `EndIf`, `GoSub`, `Let`, `Variant`, and `Wend` are retained as keywords, although they are no longer used in Visual Basic.)

```
Keyword:
    | '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'
    ;
```

2.4 Literals

A *literal* is a textual representation of a particular value of a type. Literal types include Boolean, integer, floating point, string, character, and date.

```
Literal:
    | BooleanLiteral
    | IntegerLiteral
    | FloatingPointLiteral
    | StringLiteral
    | CharacterLiteral
    | DateLiteral
```

```
    | Nothing
    ;
```

2.4.1 Boolean Literals

`True` and `False` are literals of the `Boolean` type that map to the true and false state, respectively.

```
BooleanLiteral:
    | 'True' | 'False'
    ;
```

2.4.2 Integer Literals

Integer literals can be decimal (base 10), hexadecimal (base 16), or octal (base 8). A decimal integer literal is a string of decimal digits (0-9). A hexadecimal literal is `&H` followed by a string of hexadecimal digits (0-9, A-F). An octal literal is `&O` followed by a string of octal digits (0-7). Decimal literals directly represent the decimal value of the integral literal, whereas octal and hexadecimal literals represent the binary value of the integer literal (thus, `&H8000S` is -32768, not an overflow error).

```
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'
    ;
```

The type of a literal is determined by its value or by the following type character. If no type character is specified, values in the range of the `Integer` type are typed as `Integer`; values outside the range for `Integer` are typed as `Long`. If an integer literal's type is of insufficient size to hold the integer literal, a compile-time error results. (**Note.** There isn't a type character for `Byte` because the most natural character would be `B`, which is a legal character in a hexadecimal literal.)

2.4.3 Floating-Point Literals

A floating-point literal is an integer literal followed by an optional decimal point (the ASCII period character) and mantissa, and an optional base 10 exponent. By default, a floating-point literal is of type `Double`. If the `Single`, `Double`, or `Decimal` type character is specified, the literal is of that type. If a floating-point literal's type is of insufficient size to hold the floating-point literal, a compile-time error results.

Note. It is worth noting that the `Decimal` data type can encode trailing zeros in a value. The specification currently makes no comment about whether trailing zeros in a `Decimal` literal should be honored by a compiler.

```
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:
    | '+'
    | '-'
    ;
```

2.4.4 String Literals

A string literal is a sequence of zero or more Unicode characters beginning and ending with an ASCII double-quote character, a Unicode left double-quote character, or a Unicode right double-quote character. Within a string, a sequence of two double-quote characters is an escape sequence representing a double quote in the string.

```
StringLiteral:
    | DoubleQuoteCharacter StringCharacter* DoubleQuoteCharacter
    ;

DoubleQuoteCharacter:
    | '"'
    | unicode left double-quote 0x201c
    | unicode right double-quote 0x201D
    ;

StringCharacter:
    | Any character except DoubleQuoteCharacter
    | DoubleQuoteCharacter DoubleQuoteCharacter
    ;
```

A string constant is of the `String` type.

```
        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
```

The compiler is allowed to replace a constant string expression with a string literal. Each string literal does not necessarily result in a new string instance. When two or more string literals that are equivalent according to the string equality operator using binary comparison semantics appear in the same program, these string literals may refer to the same string instance. For instance, the output of the following program may return `True` because the two literals may refer to the same string instance.

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

2.4.5 Character Literals

A character literal represents a single Unicode character of the `Char` type. Two double-quote characters is an escape sequence representing the double-quote character.

```
CharacterLiteral:
    | DoubleQuoteCharacter StringCharacter DoubleQuoteCharacter 'C'
    ;
```

```
Module Test
    Sub Main()

        ' This prints out: a.
        Console.WriteLine("a"c)

        ' This prints out: ".
        Console.WriteLine(""""c)
    End Sub
End Module
```

2.4.6 Date Literals

A date literal represents a particular moment in time expressed as a value of the `Date` type.

```
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'
    ;
```

The literal may specify both a date and a time, just a date, or just a time. If the date value is omitted, then January 1 of the year 1 in the Gregorian calendar is assumed. If the time value is omitted, then 12:00:00 AM is assumed.

To avoid problems with interpreting the year value in a date value, the year value cannot be two digits. When expressing a date in the first century AD/CE, leading zeros must be specified.

A time value may be specified either using a 24-hour value or a 12-hour value; time values that omit an `AM` or `PM` are assumed to be 24-hour values. If a time value omits the minutes, the literal `0` is used by default. If a time value omits the seconds, the literal `0` is used by default. If both minutes and second are omitted, then `AM` or `PM` must be specified. If the date value specified is outside the range of the `Date` type, a compile-time error occurs.

The following example contains several date literals.

```
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.
```

2.4.7 Nothing

`Nothing` is a special literal; it does not have a type and is convertible to all types in the type system, including type parameters. When converted to a particular type, it is the equivalent of the default value of that type.

```
Nothing:
    | 'Nothing'
    ;
```

2.5 Separators

The following ASCII characters are separators:

```
Separator:
    | '(' | ')' | '{' | '}' | '!' | '#' | ',' | '.' | ':' | '?'
    ;
```

2.6 Operator Characters

The following ASCII characters or character sequences denote operators:

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

3. Preprocessing Directives

Once a file has been lexically analyzed, several kinds of source preprocessing occur. The most important, conditional compilation, determines which source is processed by the syntactic grammar; two other types of directives -- external source directives and region directives -- provide meta-information about the source but have no effect on compilation.

3.1 Conditional Compilation

Conditional compilation controls whether sequences of logical lines are translated into actual code. At the beginning of conditional compilation, all logical lines are enabled; however, enclosing lines in conditional compilation statements may selectively disable those lines within the file, causing them to be ignored during the rest of the compilation process.

```
CCStart:
    | 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 ')'
    ;
```

For example, the program

```
#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
```

produces the exact same sequence of tokens as the program

```
Class C
    Sub F()
    End Sub

    Sub I()
    End Sub
End Class
```

The constant expressions allowed in conditional compilation directives are a subset of general constant expressions.

The preprocessor allows whitespace and explicit line continuations before and after every token.

3.1.1 Conditional Constant Directives

Conditional constant statements define constants that exist in a separate conditional compilation declaration space scoped to the source file.

```
CCConstantDeclaration:
    | '#' 'Const' Identifier '=' CCExpression LineTerminator
    ;
```

The declaration space is special in that no explicit declaration of conditional compilation constants is necessary -- conditional constants can be implicitly defined in a conditional compilation directive.

Prior to being assigned a value, a conditional compilation constant has the value `Nothing`. When a conditional compilation constant is assigned a value, which must be a constant expression, the type of the constant becomes the type of the value being assigned to it. A conditional compilation constant may be redefined multiple times throughout a source file.

For example, the following code prints only the string `about to print value` and the value of `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
```

The compilation environment may also define conditional constants in a conditional compilation declaration space.

3.1.2 Conditional Compilation Directives

Conditional compilation directives control conditional compilation.

```
CCIfGroup:
    | '#' 'If' CCExpression 'Then'? LineTerminator CCStatement*
      CCElseIfGroup* CCElseGroup? '#' 'End' 'If' LineTerminator
    ;

CCElseIfGroup:
    | '#' ElseIf CCExpression 'Then'? LineTerminator CCStatement*
    ;

CCElseGroup:
    | '#' 'Else' LineTerminator CCStatement*
    ;
```

Conditional constants can only reference constant expressions and conditional compilation constants. Each of the constant expressions within a single conditional compilation group is evaluated and converted to the `Boolean` type in textual order from first to last until one of the conditional expressions evaluates to `True`. If an expression is not convertible to `Boolean`, a compile-time error results. Permissive semantics and binary string comparisons are always used when evaluating conditional compilation constant expressions, regardless of any `Option` directives or compilation environment settings.

All lines enclosed by the group, including nested conditional compilation directives, are disabled except for lines between the statement containing the `True` expression and the next conditional statement of the group, or lines between the `Else` statement and the `End If` statement if an `Else` appears in the group and all of the expressions evaluate to `False`.

In this example, the call to `WriteToLog` in the `Trace` conditional compilation directive is not processed because the surrounding `Debug` conditional compilation directive evaluates to `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
```

3.2 External Source Directives

A source file may include external source directives that indicate a mapping between source lines and text external to the source.

```
ESDStart:
    | ExternalSourceStatement*
    ;

ExternalSourceStatement:
    | ExternalSourceGroup
    | LogicalLine
    ;

ExternalSourceGroup:
    | '#' 'ExternalSource' '(' StringLiteral ',' IntLiteral ')' LineTerminator
      LogicalLine* '#' 'End' 'ExternalSource' LineTerminator
    ;
```

External source directives have no effect on compilation and may not be nested. For example:

```
Module Test
    Sub Main()

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

    End Sub
End Module
```

3.3 Region Directives

Region directives group lines of source code but have no other effect on compilation. The entire group can be collapsed and hidden, or expanded and viewed, in the integrated development environment (IDE).

```
RegionStart:
    | RegionStatement*
    ;
```

```
RegionStatement:
    | RegionGroup
    | LogicalLine
    ;

RegionGroup:
    | '#' 'Region' StringLiteral LineTerminator
      RegionStatement*
      '#' 'End' 'Region' LineTerminator
    ;
```

Regions may be nested. Region directives are special in that they can neither start nor terminate within a method body, and they must respect the block structure of the program. For example:

```
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
```

3.4 External Checksum Directives

A source file may include an external checksum directive that indicates what checksum should be emitted for a file referenced in an external source directive. In all other respects external source directives have no effect on compilation.

```
ExternalChecksumStart:
    | ExternalChecksumStatement*
    ;

ExternalChecksumStatement:
    | '#' 'ExternalChecksum' '('
      StringLiteral ',' StringLiteral ',' StringLiteral
      ')' LineTerminator
    ;
```

An external checksum directive contains the filename of the external file, a globally unique identifier (GUID) associated with the file and the checksum for the file. The GUID is specified as a string constant of the form "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}", where x is a hexadecimal digit. The checksum is specified as a string constant of the form "xxxx...", where x is a hexadecimal digit. The number of digits in a checksum must be an even number.

An external file may have multiple external checksum directives associated with it provided that all of the GUID and checksum values match exactly. If the name of the external file matches the name of a file being compiled, the checksum is ignored in favor of the compiler's checksum calculation.

For example:

```
#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
```

4. General Concepts

This chapter covers a number of concepts that are required to understand the semantics of the Microsoft Visual Basic language. Many of the concepts should be familiar to Visual Basic programmers or C/C++ programmers, but their precise definitions may differ.

4.1 Declarations

A Visual Basic program is made up of named entities. These entities are introduced through *declarations* and represent the "meaning" of the program.

At a top level, *namespaces* are entities that organize other entities, such as nested namespaces and types. *Types* are entities that describe values and define executable code. Types may contain nested types and type members. *Type members* are constants, variables, methods, operators, properties, events, enumeration values, and constructors.

An entity that can contain other entities defines a *declaration space*. Entities are introduced into a declaration space either through declarations or inheritance; the containing declaration space is called the entities' *declaration context*. Declaring an entity in a declaration space in turn defines a new declaration space that can contain further nested entity declarations; thus, the declarations in a program form a hierarchy of declaration spaces.

Except in the case of overloaded type members, it is invalid for declarations to introduce identically named entities of the same kind into the same declaration context. Additionally, a declaration space may never contain different kinds of entities with the same name; for example, a declaration space may never contain a variable and a method by the same name.

Note. It may be possible in other languages to create a declaration space that contains different kinds of entities with the same name (for example, if the language is case sensitive and allows different declarations based on casing). In that situation, the most accessible entity is considered bound to that name; if more than one type of entity is most accessible then the name is ambiguous. `Public` is more accessible than `Protected Friend`, `Protected Friend` is more accessible than `Protected` or `Friend`, and `Protected` or `Friend` is more accessible than `Private`.

The declaration space of a namespace is "open ended," so two namespace declarations with the same fully qualified name contribute to the same declaration space. In the example below, the two namespace declarations contribute to the same declaration space, in this case declaring two classes with the fully qualified names `Data.Customer` and `Data.Order`:

```
Namespace Data
    Class Customer
    End Class
End Namespace

Namespace Data
    Class Order
    End Class
End Namespace
```

Because the two declarations contribute to the same declaration space, a compile-time error would occur if each contained a declaration of a class with the same name.

4.1.1 Overloading and Signatures

The only way to declare identically named entities of the same kind in a declaration space is through *overloading*. Only methods, operators, instance constructors, and properties may be overloaded.

Overloaded type members must possess unique signatures. The signature of a type member consists of the number of type parameters, and the number and types of the member's parameters. Conversion operators also include the return type of the operator in the signature.

The following are not part of a member's signature, and hence cannot be overloaded on:

- Modifiers to a type member (for example, `Shared` or `Private`).
- Modifiers to a parameter (for example, `ByVal` or `ByRef`).
- The names of the parameters.
- The return type of a method or operator (except for conversion operators) or the element type of a property.
- Constraints on a type parameter.

The following example shows a set of overloaded method declarations along with their signatures. This declaration would not be valid since several of the method declarations have identical signatures.

```
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
```

It is valid to define a generic type that may contain members with identical signatures based on the type arguments supplied. Overload resolution rules are used to try and disambiguate between such overloads, although there may be situations in which it is impossible to disambiguate. For example:

```
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
```

4.2 Scope

The *scope* of an entity's name is the set of all declaration spaces within which it is possible to refer to that name without qualification. In general, the scope of an entity's name is its entire declaration context; however, an entity's declaration may contain nested declarations of entities with the same name. In that case, the nested entity *shadows*, or hides, the outer entity, and access to the shadowed entity is only possible through qualification.

Shadowing through nesting occurs in namespaces or types nested within namespaces, in types nested within other types, and in the bodies of type members. Shadowing through the nesting of declarations always occurs implicitly; no explicit syntax is required.

In the following example, within the `F` method, the instance variable `i` is shadowed by the local variable `i`, but within the `G` method, `i` still refers to the instance variable.

```
Class Test
    Private i As Integer = 0

    Sub F()
        Dim i As Integer = 1
    End Sub

    Sub G()
        i = 1
    End Sub
End Class
```

When a name in an inner scope hides a name in an outer scope, it shadows all overloaded occurrences of that name. In the following example, the call `F(1)` invokes the `F` declared in `Inner` because all outer occurrences of `F` are hidden by the inner declaration. For the same reason, the call `F("Hello")` is in error.

```
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
```

4.3 Inheritance

An inheritance relationship is one in which one type (the *derived* type) derives from another (the *base* type), such that the derived type's declaration space implicitly contains the accessible non-constructor type members and nested types of its base type. In the following example, class `A` is the base class of `B`, and `B` is derived from `A`.

```
Class A
End Class

Class B
```

```
    Inherits A
End Class
```

Since `A` does not explicitly specify a base class, its base class is implicitly `Object`.

The following are important aspects of inheritance:

- Inheritance is transitive. If type *C* is derived from type *B*, and type *B* is derived from type *A*, type *C* inherits the type members declared in type *B* as well as the type members declared in type *A*.
- A derived type extends, but cannot narrow, its base type. A derived type can add new type members, and it can shadow inherited type members, but it cannot remove the definition of an inherited type member.
- Because an instance of a type contains all of the type members of its base type, a conversion always exists from a derived type to its base type.
- All types must have a base type, except for the type `Object`. Thus, `Object` is the ultimate base type of all types, and all types can be converted to it.
- Circularity in derivation is not permitted. That is, when a type `B` derives from a type `A`, it is an error for type `A` to derive directly or indirectly from type `B`.
- A type may not directly or indirectly derive from a type nested within it.

The following example produces a compile-time error because the classes circularly depend on each other.

```
Class A
    Inherits B
End Class

Class B
    Inherits C
End Class

Class C
    Inherits A
End Class
```

The following example also produces a compile-time error because `B` indirectly derives from its nested class `C` through class `A`.

```
Class A
    Inherits B.C
End Class

Class B
    Inherits A

    Public Class C
    End Class
End Class
```

The next example does not produce an error because class `A` does not derive from class `B`.

```
Class A
    Class B
        Inherits A
    End Class
End Class
```

4.3.1 MustInherit and NotInheritable Classes

A `MustInherit` class is an incomplete type that can act only as a base type. A `MustInherit` class cannot be instantiated, so it is an error to use the `New` operator on one. It is valid to declare variables of `MustInherit` classes; such variables can only be assigned `Nothing` or a value that is of a class derived from the `MustInherit` class.

When a regular class is derived from a `MustInherit` class, the regular class must override all inherited `MustOverride` members. For example:

```
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
```

The `MustInherit` class `A` introduces a `MustOverride` method `F`. Class `B` introduces an additional method `G`, but does not provide an implementation of `F`. Class `B` must therefore also be declared `MustInherit`. Class `C` overrides `F` and provides an actual implementation. Since there are no outstanding `MustOverride` members in class `C`, it is not required to be `MustInherit`.

A `NotInheritable` class is a class from which another class cannot be derived. `NotInheritable` classes are primarily used to prevent unintended derivation.

In this example, class `B` is in error because it attempts to derive from the `NotInheritable` class `A`. A class cannot be marked both `MustInherit` and `NotInheritable`.

```
NotInheritable Class A
End Class

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

4.3.2 Interfaces and Multiple Inheritance

Unlike other types, which only derive from a single base type, an interface may derive from multiple base interfaces. Because of this, an interface can inherit an identically named type member from different base interfaces. In such a case, the multiply-inherited name is not available in the derived interface, and referring to any of those type members through the derived interface causes a compile-time error, regardless of signatures or overloading. Instead, conflicting type members must be referenced through a base interface name.

In the following example, the first two statements cause compile-time errors because the multiply-inherited member `Count` is not available in interface `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
```

As illustrated by the example, the ambiguity is resolved by casting `x` to the appropriate base interface type. Such casts have no run-time costs; they merely consist of viewing the instance as a less-derived type at compile time.

When a single type member is inherited from the same base interface through multiple paths, the type member is treated as if it were only inherited once. In other words, the derived interface only contains one instance of each type member inherited from a particular base interface. For example:

```
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
```

If a type member name is shadowed in one path through the inheritance hierarchy, then the name is shadowed in all paths. In the following example, the `IBase.F` member is shadowed by the `ILeft.F` member, but is not shadowed in `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
```

The invocation `d.F(1)` selects `ILeft.F`, even though `IBase.F` appears to not be shadowed in the access path that leads through `IRight`. Because the access path from `IDerived` to `ILeft` to `IBase` shadows `IBase.F`, the member is also shadowed in the access path from `IDerived` to `IRight` to `IBase`.

4.3.3 Shadowing

A derived type shadows the name of an inherited type member by re-declaring it. Shadowing a name does not remove the inherited type members with that name; it merely makes all of the inherited type members with that name unavailable in the derived class. The shadowing declaration may be any type of entity.

Entities than can be overloaded can choose one of two forms of shadowing. *Shadowing by name* is specified using the `Shadows` keyword. An entity that shadows by name hides everything by that name in the base class, including all overloads. *Shadowing by name and signature* is specified using the `Overloads` keyword. An entity that shadows by name and signature hides everything by that name with the same signature as the entity. For example:

```
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
```

Shadowing a method with a `ParamArray` argument by name and signature hides only the individual signature, not all possible expanded signatures. This is true even if the signature of the shadowing method matches the unexpanded signature of the shadowed method. The following example:

```
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
```

prints `Base`, even though `Derived.F` has the same signature as the unexpanded form of `Base.F`.

Conversely, a method with a `ParamArray` argument only shadows methods with the same signature, not all possible expanded signatures. The following example:

```
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
```

prints `Base`, even though `Derived.F` has an expanded form that has the same signature as `Base.F`.

A shadowing method or property that does not specify `Shadows` or `Overloads` assumes `Overloads` if the method or property is declared `Overrides`, `Shadows` otherwise. If one member of a set of overloaded entities specifies the `Shadows` or `Overloads` keyword, they all must specify it. The `Shadows` and `Overloads` keywords cannot be specified at the same time. Neither `Shadows` nor `Overloads` can be specified in a standard module; members in a standard module implicitly shadow members inherited from `Object`.

It is valid to shadow the name of a type member that has been multiply-inherited through interface inheritance (and which is thereby unavailable), thus making the name available in the derived interface.

For example:

```
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
```

Because methods are allowed to shadow inherited methods, it is possible for a class to contain several `Overridable` methods with the same signature. This does not present an ambiguity problem, since only the most-derived method is visible. In the following example, the `C` and `D` classes contain two `Overridable` methods with the same signature:

```
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
```

There are two `Overridable` methods here: one introduced by class `A` and the one introduced by class `C`. The method introduced by class `C` hides the method inherited from class `A`. Thus, the `Overrides` declaration in class `D` overrides the method introduced by class `C`, and it is not possible for class `D` to override the method introduced by class `A`. The example produces the output:

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

It is possible to invoke the hidden `Overridable` method by accessing an instance of class `D` through a less-derived type in which the method is not hidden.

It is not valid to shadow a `MustOverride` method, because in most cases this would make the class unusable. For example:

```
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
```

In this case, the class `MoreDerived` is required to override the `MustOverride` method `Base.F`, but because the class `Derived` shadows `Base.F`, this is not possible. There is no way to declare a valid descendent of `Derived`.

In contrast to shadowing a name from an outer scope, shadowing an accessible name from an inherited scope causes a warning to be reported, as in the following example:

```
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
```

The declaration of method `F` in class `Derived` causes a warning to be reported. Shadowing an inherited name is specifically not an error, since that would preclude separate evolution of base classes. For example, the above situation might have come about because a later version of class `Base` introduced a method `F` that was not present in an earlier version of the class. Had the above situation been an error, *any* change made to a base class in a separately versioned class library could potentially cause derived classes to become invalid.

The warning caused by shadowing an inherited name can be eliminated through use of the `Shadows` or `Overloads` modifier:

```
Class Base
    Public Sub F()
    End Sub
End Class

Class Derived
    Inherits Base

    Public Shadows Sub F() 'OK.
    End Sub
End Class
```

The `Shadows` modifier indicates the intention to shadow the inherited member. It is not an error to specify the `Shadows` or `Overloads` modifier if there is no type member name to shadow.

A declaration of a new member shadows an inherited member only within the scope of the new member, as in the following example:

```
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
```

In the example above, the declaration of method `F` in class `Derived` shadows the method `F` that was inherited from class `Base`, but since the new method `F` in class `Derived` has `Private` access, its scope does not extend to class `MoreDerived`. Thus, the call `F()` in `MoreDerived.G` is valid and will invoke `Base.F`. In the case of overloaded type members, the entire set of overloaded type members is treated as if they all had the most permissive access for the purposes of shadowing.

```
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
```

In this example, even though the declaration of `F()` in `Derived` is declared with `Private` access, the overloaded `F(Integer)` is declared with `Public` access. Therefore, for the purpose of shadowing, the name `F` in `Derived` is treated as if it was `Public`, so both methods shadow `F` in `Base`.

4.4 Implementation

An *implementation* relationship exists when a type declares that it implements an interface and the type implements all the type members of the interface. A type that implements a particular interface is convertible to that interface. Interfaces cannot be instantiated, but it is valid to declare variables of interfaces; such variables can only be assigned a value that is of a class that implements the interface. For example:

```
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
```

A type implementing an interface with multiply-inherited type members must still implement those methods, even though they cannot be accessed directly from the derived interface being implemented. For example:

```
    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
```

Even `MustInherit` classes must provide implementations of all the members of implemented interfaces; however, they can defer implementation of these methods by declaring them as `MustOverride`. For example:

```
    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
```

A type may choose to re-implement an interface that its base type implements. To re-implement the interface, the type must explicitly state that it implements the interface. A type re-implementing an interface may choose to re-implement only some, but not all, of the members of the interface -- any members not re-implemented continue to use the base type's implementation. For example:

```
    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
```

This example prints:

```
    TestDerived.DerivedTest1
    TestBase.Test2
```

When a derived type implements an interface whose base interfaces are implemented by the derived type's base types, the derived type can choose to only implement the interface's type members that are not already implemented by the base types. For example:

```
    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
```

An interface method can also be implemented using an overridable method in a base type. In that case, a derived type may also override the overridable method and alter the implementation of the interface. For example:

```
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
```

4.4.1 Implementing Methods

A type *implements* a type member of an implemented interface by supplying a method with an `Implements` clause. The two type members must have the same number of parameters, all of the types and modifiers of the parameters must match, including the default value of optional parameters, the return type must match, and all of the constraints on method parameters must match. For example:

```
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
```

A single method may implement any number of interface type members if they all meet the above criteria. For example:

```
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
```

When implementing a method in a generic interface, the implementing method must supply the type arguments that correspond to the interface's type parameters. For example:

```
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
```

Note that it is possible that a generic interface may not be implementable for some set of type arguments.

```
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
```

4.5 Polymorphism

Polymorphism provides the ability to vary the implementation of a method or property. With polymorphism, the same method or property can perform different actions depending on the run-time type of the instance that invokes

it. Methods or properties that are polymorphic are called *overridable*. By contrast, the implementation of a non-overridable method or property is invariant; the implementation is the same whether the method or property is invoked on an instance of the class in which it is declared or an instance of a derived class. When a non-overridable method or property is invoked, the compile-time type of the instance is the determining factor. For example:

```
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
```

An overridable method may also be `MustOverride`, which means that it provides no method body and must be overridden. `MustOverride` methods are only allowed in `MustInherit` classes.

In the following example, the class `Shape` defines the abstract notion of a geometrical shape object that can paint itself:

```
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
```

The `Paint` method is `MustOverride` because there is no meaningful default implementation. The `Ellipse` and `Box` classes are concrete `Shape` implementations. Because these classes are not `MustInherit`, they are required to override the `Paint` method and provide an actual implementation.

It is an error for a base access to reference a `MustOverride` method, as the following example demonstrates:

```
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
```

An error is reported for the `MyBase.F()` invocation because it references a `MustOverride` method.

4.5.1 Overriding Methods

A type may *override* an inherited overridable method by declaring a method with the same name and , signature, and marking the declaration with the `Overrides` modifier.There are additional requirements on overriding methods, listed below. Whereas an `Overridable` method declaration introduces a new method, an `Overrides` method declaration replaces the inherited implementation of the method.

An overriding method may be declared `NotOverridable`, which prevents any further overriding of the method in derived types. In effect, `NotOverridable` methods become non-overridable in any further derived classes.

Consider the following example:

```
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
```

In the example, class `B` provides two `Overrides` methods: a method `F` that has the `NotOverridable` modifier and a method `G` that does not. Use of the `NotOverridable` modifier prevents class `C` from further overriding method `F`.

An overriding method may also be declared `MustOverride`, even if the method that it is overriding is not declared `MustOverride`. This requires that the containing class be declared `MustInherit` and that any further derived classes that are not declared `MustInherit` must override the method. For example:

```
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
```

In the example, class `B` overrides `A.F` with a `MustOverride` method. This means that any classes derived from `B` will have to override `F`, unless they are declared `MustInherit` as well.

A compile-time error occurs unless all of the following are true of an overriding method:

- The declaration context contains a single accessible inherited method with the same signature and return type (if any) as the overriding method.
- The inherited method being overridden is overridable. In other words, the inherited method being overridden is not `Shared` or `NotOverridable`.
- The accessibility domain of the method being declared is the same as the accessibility domain of the inherited method being overridden. There is one exception: a `Protected Friend` method must be overridden by a `Protected` method if the other method is in another assembly that the overriding method does not have `Friend` access to.
- The parameters of the overriding method match the overridden method's parameters in regards to usage of the `ByVal`, `ByRef`, `ParamArray`, and `Optional` modifiers, including the values provided for optional parameters.
- The type parameters of the overriding method match the overridden method's type parameters in regards to type constraints.

When overriding a method in a base generic type, the overriding method must supply the type arguments that correspond to the base type parameters. For example:

```
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
```

Note that it is possible that an overridable method in a generic class may not be able to be overridden for some sets of type arguments. If the method is declared `MustOverride`, this means that some inheritance chains may not be possible. For example:

```
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
```

An override declaration can access the overridden base method using a base access, as in the following example:

```
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
```

In the example, the invocation of `MyBase.PrintVariables()` in class `Derived` invokes the `PrintVariables` method declared in class `Base`. A base access disables the overridable invocation mechanism and simply treats the base method as a non-overridable method. Had the invocation in `Derived` been written `CType(Me, Base).PrintVariables()`, it would recursively invoke the `PrintVariables` method declared in `Derived`, not the one declared in `Base`.

Only when it includes an `Overrides` modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply shadows the inherited method, as in the example below:

```
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
```

In the example, the method `F` in class `Derived` does not include an `Overrides` modifier and therefore does not override method `F` in class `Base`. Rather, method `F` in class `Derived` shadows the method in class `Base`, and a warning is reported because the declaration does not include a `Shadows` or `Overloads` modifier.

In the following example, method `F` in class `Derived` shadows the overridable method `F` inherited from class `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
```

Since the new method `F` in class `Derived` has `Private` access, its scope only includes the class body of `Derived` and does not extend to class `MoreDerived`. The declaration of method `F` in class `MoreDerived` is therefore permitted to override the method `F` inherited from class `Base`.

When an `Overridable` method is invoked, the most derived implementation of the instance method is called, based on the type of the instance, regardless of whether the call is to the method in the base class or the derived class. The most derived implementation of an `Overridable` method `M` with respect to a class `R` is determined as follows:

- If `R` contains the introducing `Overridable` declaration of `M`, this is the most derived implementation of `M`.
- Otherwise, if `R` contains an override of `M`, this is the most derived implementation of `M`.
- Otherwise, the most derived implementation of `M` is the same as that of the direct base class of `R`.

4.6 Accessibility

A declaration specifies the *accessibility* of the entity it declares. An entity's accessibility does not change the scope of an entity's name. The *accessibility domain* of a declaration is the set of all declaration spaces in which the declared entity is accessible.

The five access types are `Public`, `Protected`, `Friend`, `Protected Friend`, and `Private`. `Public` is the most permissive access type, and the four other types are all subsets of `Public`. The least permissive access type is `Private`, and the four other access types are all supersets of `Private`.

```
AccessModifier:
    | 'Public'
    | 'Protected'
    | 'Friend'
    | 'Private'
```

```
    | 'Protected' 'Friend'
    ;
```

The access type for a declaration is specified via an optional access modifier, which can be `Public`, `Protected`, `Friend`, `Private`, or the combination of `Protected` and `Friend`. If no access modifier is specified, the default access type depends on the declaration context; the permitted access types also depend on the declaration context.

- Entities declared with the `Public` modifier have `Public` access. There are no restrictions on the use of `Public` entities.
- Entities declared with the `Protected` modifier have `Protected` access. `Protected` access can only be specified on members of classes (both regular type members and nested classes) or on `Overridable` members of standard modules and structures (which must, by definition, be inherited from `System.Object` or `System.ValueType`). A `Protected` member is accessible to a derived class, provided that either the member is not an instance member, or the access takes place through an instance of the derived class. `Protected` access is not a superset of `Friend` access.
- Entities declared with the `Friend` modifier have `Friend` access. An entity with `Friend` access is accessible only within the program that contains the entity declaration or any assemblies that have been given `Friend` access through the `System.Runtime.CompilerServices.InternalsVisibleToAttribute` attribute.
- Entities declared with the `Protected Friend` modifiers have the union of `Protected` and `Friend` access.
- Entities declared with the `Private` modifier have `Private` access. A `Private` entity is accessible only within its declaration context, including any nested entities.

The accessibility in a declaration does not depend on the accessibility of the declaration context. For example, a type declared with `Private` access may contain a type member with `Public` access.

The following code demonstrates various accessibility domains:

```
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
```

The classes and members in this example have the following accessibility domains:

- The accessibility domain of `A` and `A.X` is unlimited.
- The accessibility domain of `A.Y`, `B`, `B.X`, `B.Y`, `B.C`, `B.C.X`, and `B.C.Y` is the containing program.
- The accessibility domain of `A.Z` is `A`.

- The accessibility domain of `B.Z`, `B.D`, `B.D.X`, and `B.D.Y` is `B`, including `B.C` and `B.D`.
- The accessibility domain of `B.C.Z` is `B.C`.
- The accessibility domain of `B.D.Z` is `B.D`.

As the example illustrates, the accessibility domain of a member is never larger than that of a containing type. For example, even though all `X` members have `Public` declared accessibility, all but `A.X` have accessibility domains that are constrained by a containing type.

Access to `Protected` instance members must be through an instance of the derived type so that unrelated types cannot gain access to each other's protected members. For example:

```
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
```

In the above example, the class `Guest` only has access to the protected `Password` field if it is qualified with an instance of `Guest`. This prevents `Guest` from gaining access to the `Password` field of an `Employee` object simply by casting it to `User`.

For the purposes of `Protected` member access in generic types, the declaration context includes type parameters. This means that a derived type with one set of type arguments does not have access to the `Protected` members of a derived type with a different set of type arguments. For example:

```
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
```

Note. The C# language (and possibly other languages) allows a generic type to access `Protected` members regardless of what type arguments are supplied. This should be kept in mind when designing generic classes that contain `Protected` members.

4.6.1 Constituent Types

The *constituent types* of a declaration are the types that are referenced by the declaration. For example, the type of a constant, the return type of a method and the parameter types of a constructor are all constituent types. The

accessibility domain of a constituent type of a declaration must be the same as or a superset of the accessibility domain of the declaration itself. For example:

```
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
```

4.7 Type and Namespace Names

Many language constructs require a namespace or type to be specified; these can be specified by using a qualified form of the namespace or type's name. A *qualified name* consists of a series of identifiers separated by periods; the identifier on the right side of a period is resolved in the declaration space specified by the identifier on the left side of the period.

The *fully qualified name* of a namespace or type is a qualified name that contains the name of all containing namespaces and types. In other words, the fully qualified name of a namespace or type is `N.T`, where `T` is the name of the entity and `N` is the fully qualified name of its containing entity.

The example below shows several namespace and type declarations together with their associated fully qualified names in in-line comments.

```
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 that the namespace X.Y has been declared in two different locations in the source code, but these two partial declarations constitute just a single namespace called X.Y which contains both class D and class E.

In some situations, a qualified name may begin with the keyword `Global`. The keyword represents the unnamed outermost namespace, which is useful in situations where a declaration shadows an enclosing namespace. The `Global` keyword allows "escaping" out to the outermost namespace in that situation. For example:

```
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
```

In the above example, the first method call is invalid because the identifier `System` binds to the class `System`, not the namespace `System`. The only way to access the `System` namespace is to use `Global` to escape out to the outermost namespace. `Global` cannot be used in an `Imports` statement or `Namespace` declaration.

Because other languages may introduce types and namespaces that match keywords in the language, Visual Basic recognizes keywords to be part of a qualified name as long as they follow a period. Keywords used in this way are treated as identifiers. For example, the qualified identifier `X.Default.Class` is a valid qualified identifier, while `Default.Class` is not.

4.7.1 Qualified Name Resolution for namespaces and types

Given a qualified namespace or type name of the form `N.R(Of A)`, where `R` is the rightmost identifier in the qualified name and `A` is an optional type argument list, the following steps describe how to determine to which namespace or type the qualified name refers:

1. Resolve `N`, using the rules for either qualified or unqualified name resolution.
2. If resolution of `N` fails, or resolves to a type parameter, a compile-time error occurs.
3. Otherwise, if `R` matches the name of a namespace in N and no type arguments were supplied, or `R` matches an accessible type in `N` with the same number of type parameters as type arguments, if any, then the qualified name refers to that namespace or type.
4. Otherwise, if `N` contains one or more standard modules, and `R` matches the name of an accessible type with the same number of type parameters as type arguments, if any, in exactly one standard module, then the qualified name refers to that type. If `R` matches the name of accessible types with the same number of type parameters as type arguments, if any, in more than one standard module, a compile-time error occurs.
5. Otherwise, a compile-time error occurs.

Note. An implication of this resolution process is that type members do not shadow namespaces or types when resolving namespace or type names.

4.7.2 Unqualified Name Resolution for namespaces and types

Given an unqualified name `R(Of A)`, where `A` is an optional type argument list, the following steps describe how to determine to which namespace or type the unqualified name refers:

1. If R matches the name of a type parameter of the current method, and no type arguments were supplied, then the unqualified name refers to that type parameter.
2. For each nested type containing the name reference, starting from the innermost type and going to the outermost:
    a. If R matches the name of a type parameter in the current type and no type arguments were supplied, then the unqualified name refers to that type parameter.
    b. Otherwise, if R matches the name of an accessible nested type with the same number of type parameters as type arguments, if any, then the unqualified name refers to that type.
3. For each nested namespace containing the name reference, starting from the innermost namespace and going to the outermost namespace:
    a. If R matches the name of a nested namespace in the current namespace and no type argument list is supplied, then the unqualified name refers to that nested namespace.
    b. Otherwise, if R matches the name of an accessible type with the same number of type parameters as type arguments, if any, in the current namespace, then the unqualified name refers to that type.
    c. Otherwise, if the namespace contains one or more accessible standard modules, and R matches the name of an accessible nested type with the same number of type parameters as type arguments, if any, in exactly one standard module, then the unqualified name refers to that nested type. If R matches the name of accessible nested types with the same number of type parameters as type arguments, if any, in more than one standard module, a compile-time error occurs.
4. If the source file has one or more import aliases, and R matches the name of one of them, then the unqualified name refers to that import alias. If a type argument list is supplied, a compile-time error occurs.
5. If the source file containing the name reference has one or more imports:
    a. If R matches the name of an accessible type with the same number of type parameters as type arguments, if any, in exactly one import, then the unqualified name refers to that type. If R matches the name of an accessible type with the same number of type parameters as type arguments, if any, in more than one import and all are not the same type, a compile-time error occurs.
    b. Otherwise, if no type argument list was supplied and R matches the name of a namespace with accessible types in exactly one import, then the unqualified name refers to that namespace. If no type argument list was supplied and R matches the name of a namespace with accessible types in more than one import and all are not the same namespace, a compile-time error occurs.
    c. Otherwise, if the imports contain one or more accessible standard modules, and R matches the name of an accessible nested type with the same number of type parameters as type arguments, if any, in exactly one standard module, then the unqualified name refers to that type. If R matches the name of accessible nested types with the same number of type parameters as type arguments, if any, in more than one standard module, a compile-time error occurs.
6. If the compilation environment defines one or more import aliases, and R matches the name of one of them, then the unqualified name refers to that import alias. If a type argument list is supplied, a compile-time error occurs.
7. If the compilation environment defines one or more imports:
    a. If R matches the name of an accessible type with the same number of type parameters as type arguments, if any, in exactly one import, then the unqualified name refers to that type. If R matches the name of an accessible type with the same number of type parameters as type arguments, if any, in more than one import, a compile-time error occurs.
    b. Otherwise, if no type argument list was supplied and R matches the name of a namespace with accessible types in exactly one import, then the unqualified name refers to that namespace. If no type argument list

was supplied and `R` matches the name of a namespace with accessible types in more than one import, a compile-time error occurs.

c. Otherwise, if the imports contain one or more accessible standard modules, and `R` matches the name of an accessible nested type with the same number of type parameters as type arguments, if any, in exactly one standard module, then the unqualified name refers to that type. If `R` matches the name of accessible nested types with the same number of type parameters as type arguments, if any, in more than one standard module, a compile-time error occurs.

8. Otherwise, a compile-time error occurs.

Note. An implication of this resolution process is that type members do not shadow namespaces or types when resolving namespace or type names.

Normally, a name can only occur once in a particular namespace. However, because namespaces can be declared across multiple .NET assemblies, it is possible to have a situation where two assemblies define a type with the same fully qualified name. In that case, a type declared in the current set of source files is preferred over a type declared in an external .NET assembly. Otherwise, the name is ambiguous and there is no way to disambiguate the name.

4.8 Variables

A *variable* represents a storage location. Every variable has a type that determines what values can be stored in the variable. Because Visual Basic is a type-safe language, every variable in a program has a type and the language guarantees that values stored in variables are always of the appropriate type. Variables are always initialized to the default value of their type before any reference to the variable can be made. It is not possible to access uninitialized memory.

4.9 Generic Types and Methods

Types (except for standard modules and enumerated types) and methods can declare *type parameters*, which are types that will not be provided until an instance of the type is declared or the method is invoked. Types and methods with type parameters are also known as *generic types* and *generic methods*, respectively, because the type or method must be written generically, without specific knowledge of the types that will be supplied by code that uses the type or method.

Note. At this time, even though methods and delegates can be generic, properties, events and operators cannot be generic themselves. They may, however, use type parameters from the containing class.

From the perspective of the generic type or method, a type parameter is a placeholder type that will be filled in with an actual type when the type or method is used. Type arguments are substituted for the type parameters in the type or method at the point at which the type or method is used. For example, a generic stack class could be implemented as:

```
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
```

Declarations that use the `Stack(Of ItemType)` class must supply a type argument for the type parameter `ItemType`. This type is then filled in wherever `ItemType` is used within the class:

```
    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
```

4.9.1 Type Parameters

Type parameters may be supplied on type or method declarations. Each type parameter is an identifier which is a place-holder for a type argument that is supplied to create a constructed type or method. By contrast, a type argument is the actual type that is substituted for the type parameter when a generic type or method is used.

```
TypeParameterList:
    | OpenParenthesis 'Of' TypeParameter ( Comma TypeParameter  )* CloseParenthesis
    ;

TypeParameter:
    | VarianceModifier? Identifier TypeParameterConstraints?
    ;

VarianceModifier:
    | 'In' | 'Out'
    ;
```

Each type parameter in a type or method declaration defines a name in the declaration space of that type or method. Thus, it cannot have the same name as another type parameter, a type member, a method parameter, or a local variable. The scope of a type parameter on a type or method is the entire type or method. Because type parameters are scoped to the entire type declaration, nested types can use outer type parameters. This also means that type parameters must always be specified when accessing types nested inside generic types:

```
    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
```

Unlike other members of a class, type parameters are not inherited. Type parameters in a type can only be referred to by their simple name; in other words, they cannot be qualified with the containing type name. Although it is bad programming style, the type parameters in a nested type can hide a member or type parameter declared in the outer type:

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

Types and methods may be overloaded based on the number of type parameters (or *arity*) that the types or methods declare. For example, the following declarations are legal:

```
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
```

In the case of types, overloads are always matched against the number of type arguments specified. This is useful when using both generic and non-generic classes together in the same program:

```
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
```

Rules for methods overloaded on type parameters are covered in the section on method overload resolution.

Within the containing declaration, type parameters are considered full types. Since a type parameter can be instantiated with many different actual type arguments, type parameters have slightly different operations and restrictions than other types as described below:

- A type parameter cannot be used directly to declare a base class or interface.
- The rules for member lookup on type parameters depend on the constraints, if any, applied to the type parameter.
- The available conversions for a type parameter depend on the constraints, if any, applied to the type parameters.

- In the absence of a `Structure` constraint, a value with a type represented by a type parameter can be compared with `Nothing` using `Is` and `IsNot`.
- A type parameter can only be used in a `New` expression if the type parameter is constrained by a `New` or a `Structure` constraint.
- A type parameter cannot be used anywhere within an attribute exception within a `GetType` expression.
- Type parameters can be used as type arguments to other generic types and parameters.

The following example is a generic type that extends the `Stack(Of ItemType)` class:

```
Class MyStack(Of ItemType)
    Inherits Stack(Of ItemType)

    Public ReadOnly Property Size() As Integer
        Get
            Return CurrentIndex
        End Get
    End Property
End Class
```

When a declaration supplies a type argument to `MyStack`, the same type argument will be applied to `Stack` as well.

As a type, type parameters are purely a compile-time construct. At run-time, each type parameter is bound to a run-time type that was specified by supplying a type argument to the generic declaration. Thus, the type of a variable declared with a type parameter will, at run-time, be a non-generic type or a specific constructed type. The run-time execution of all statements and expressions involving type parameters uses the actual type that was supplied as the type argument for that parameter.

4.9.2 Type Constraints

Because a type argument can be any type in the type system, a generic type or method cannot make any assumptions about a type parameter. Thus, the members of a type parameter are considered to be the members of the type `Object`, since all types derive from `Object`.

In the case of a collection like `Stack(Of ItemType)`, this fact may not be a particularly important restriction, but there may be cases where a generic type may wish to make an assumption about the types that will be supplied as type arguments. *Type constraints* can be placed on type parameters that restrict which types can be supplied as a type parameter and allow generic types or methods to assume more about type parameters.

```
TypeParameterConstraints:
    | 'As' Constraint
    | 'As' OpenCurlyBrace ConstraintList CloseCurlyBrace
    ;

ConstraintList:
    | Constraint ( Comma Constraint  )*
    ;

Constraint:
    | TypeName
    | 'New'
    | 'Structure'
    | 'Class'
    ;
```

```
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
```

In this example, the `DisposableStack(Of ItemType)` constrains its type parameter to only types that implement the interface `System.IDisposable`. As a result, it can implement a `Dispose` method that disposes any objects still left in the queue.

A type constraint must be one of the special constraints `Class`, `Structure`, or `New`, or it must be a type `T` where:

- `T` is a class, an interface, or a type parameter.
- `T` is not `NotInheritable`.
- `T` is not one of, or a type inherited from one of, the following special types: `System.Array`, `System.Delegate`, `System.MulticastDelegate`, `System.Enum`, or `System.ValueType`.
- `T` is not `Object`. Since all types derive from `Object`, such a constraint would have no effect if it were permitted.
- `T` must be at least as accessible as the generic type or method being declared.

Multiple type constraints can be specified for a single type parameter by enclosing the type constraints in curly braces (`{}`).. Only one type constraint for a given type parameter can be a class. It is an error to combine a `Structure` special constraint with a named class constraint or the `Class` special constraint.

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

Type constraints can use the containing types or any of the containing types' type parameters. In the following example, the constraint requires that the type argument supplied implements a generic interface using itself as a type argument:

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

The special type constraint `Class` constrains the supplied type argument to any reference type.

Note. The special type constraint `Class` can be satisfied by an interface. And a structure can implement an interface. Therefore, the constraint `(Of T As U, U As Class)` might be satisfied with "T" a structure (which does not satisfy the `Class` special constraint), and "U" an interface that it implements (which does satisfy the `Class` special constraint).

The special type constraint `Structure` constrains the supplied type argument to any value type except `System.Nullable(Of T)`.

Note. Structure constraints do not allow `System.Nullable(Of T)` so that it is not possible to supply `System.Nullable(Of T)` as a type argument to itself.

The special type constraint `New` requires that the supplied type argument must have an accessible parameterless constructor and cannot be declared `MustInherit`. For example:

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

A class type constraint requires that the supplied type argument must either be that type as or inherit from it. An interface type constraint requires that the supplied type argument must implement that interface. A type parameter constraint requires that the supplied type argument must derive from or implement all of the bounds given for the matching type parameter. For example:

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

In this example, the type parameter `S` on `AddRange` is constrained to the type parameter `T` of `List`. This means that a `List(Of Control)` would constrain `AddRange`'s type parameter to any type that is or inherits from `Control`.

A type parameter constraint `Of S As T` is resolved by transitively adding all of `T`'s constraints onto `S`, other than the special constraints (`Class`, `Structure`, `New`). It is an error to have circular constraints (e.g. `Of S As T, T As S`). It is an error to have a type parameter constraint which itself has the `Structure` constraint. After adding constraints, it is possible that a number of special situations may occur:

- If multiple class constraints exist, the most derived class is considered to be the constraint. If one or more class constraints have no inheritance relationship, the constraint is unsatisfiable and it is an error.
    - If a type parameter combines a `Structure` special constraint with a named class constraint or the `Class` special constraint, it is an error. A class constraint may be `NotInheritable`, in which case no derived types of that constraint are accepted and it is an error.

The type may be one of, or a type inherited from, the following special types: `System.Array`, `System.Delegate`, `System.MulticastDelegate`, `System.Enum`, or `System.ValueType`. In that case, only the type, or a type inherited from it, is accepted. A type parameter constrained to one of these types can only use the conversions allowed by the `DirectCast` operator. For example:

```
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
```

Additionally, a type parameter constrained to a value type due to one of the above relaxations cannot call any methods defined on that value type. For example:

```
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
```

If the constraint, after substitution, ends up as an array type, any covariant array type is allowed as well. For example:

```
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
```

A type parameter with a class or interface constraint is considered to have the same members as that class or interface constraint. If a type parameter has multiple constraints, then the type parameter is considered to have the union of all the members of the constraints. If there are members with the same name in more than one constraint, then members are hidden in the following order: the class constraint hides members in interface constraints, which hide members in `System.ValueType` (if `Structure` constraint is specified), which hides members in `Object`. If a member with the same name appears in more than one interface constraint the member is unavailable (as in multiple interface inheritance) and the type parameter must be cast to the desired interface. For example:

```
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
```

When supplying type parameters as type arguments, the type parameters must satisfy the constraints of the matching type parameters.

```
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
```

Values of a constrained type parameter can be used to access the instance members, including instance methods, specified in the constraint.

```
Interface IPrintable
    Sub Print()
End Interface

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

4.9.3 Type Parameter Variance

A type parameter in an interface or a delegate type declaration can optionally specify a *variance modifier*. Type parameters with variance modifiers restrict how the type parameter can be used in the interface or delegate type but allow a generic interface or delegate type to be converted to another generic type with variant compatible type arguments. For example:

```
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
```

Generic interfaces that have type parameters with variance modifiers have several restrictions:

- They cannot contain an event declaration that specifies a parameter list (but a custom event declaration or an event declaration with a delegate type is allowed).

- They cannot contain a nested class, structure, or enumerated type.

Note. These restrictions are due to the fact that types nested in generic types implicitly copy the generic parameters of their parent. In the case of nested classes, structures, or enumerated types, those kinds of types cannot have variance modifiers on their type parameters. In the case of an event declaration with a parameter list, the generated nested delegate class could have confusing errors when a type that appears to be used in an `In` position (i.e. a parameter type) is actually used in an `Out` position (i.e. the type of the event).

A type parameter that is declared with the Out modifier is *covariant*. Informally, a covariant type parameter can only be used in an output position -- i.e. a value that is being returned from the interface or delegate type -- and cannot be used in an input position. A type `T` is considered to be *valid covariantly* if:

- `T` is a class, structure, or enumerated type.
- `T` is non-generic delegate or interface type.
- `T` is an array type whose element type is valid covariantly.
- `T` is a type parameter which was not declared as an `Out` type parameter.
- `T` is a constructed interface or delegate type `X(Of P1,...,Pn)` with type arguments `A1,...,An` such that:
  - If `Pi` was declared as an Out type parameter then `Ai` is valid covariantly.
  - If `Pi` was declared as an In type parameter then `Ai` is valid contravariantly.

The following must be valid covariantly in an interface or delegate type:

- The base interface of an interface.
- The return type of a function or the delegate type.
- The type of a property if there is a `Get` accessor.
- The type of any `ByRef` parameter.

For example:

```
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
```

Note. `Out` is not a reserved word.

A type parameter that is declared with the In modifier is *contravariant*. Informally, a contravariant type parameter can only be used in an input position -- i.e. a value that is being passed in to the interface or delegate type -- and cannot be used in an output position. A type `T` is considered to be *valid contravariantly* if:

- `T` is a class, structure, or enumerated type.
- `T` is a non-generic delegate or interface type.
- `T` is an array type whose element type is valid contravariantly.

- `T` is a type parameter which was not declared as an In type parameter.
- `T` is a constructed interface or delegate type `X(Of P1,...,Pn)` with type arguments `A1,...,An` such that:
  - If `Pi` was declared as an `Out` type parameter then `Ai` is valid contravariantly.
  - If `Pi` was declared as an `In` type parameter then `Ai` is valid covariantly.

The following must be valid contravariantly in an interface or delegate type:

- The type of a parameter.
- A type constraint on a method type parameter.
- The type of a property if it has a `Set` accessor.
- The type of an event.

For example:

```
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
```

In the case where a type must be valid be contravariantly and covariantly (such as a property with both a `Get` and `Set` accessor or a `ByRef` parameter), a variant type parameter cannot be used.

Co- and contra-variance give rise to a "diamond ambiguity problem". Consider the following code:

```
Class 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()
```

The class `C` can be converted to `IEnumerable(Of Object)` in two ways, both through covariant conversion from `IEnumerable(Of String)` and through covariant conversion from `IEnumerable(Of Exception)`. The CLR does not specify which of the two methods will be called by `c.GetEnumerator()`. In general, whenever a class is declared to implement a covariant interface with two different generic arguments that have a common supertype (e.g. in this case `String` and `Exception` have the common supertype `Object`), or a class is declared to implement a contravariant interface with two different generic arguments that have a common subtype, then ambiguity is likely to arise. The compiler gives a warning on such declarations.

5. Attributes

The Visual Basic language enables the programmer to specify modifiers on declarations, which represent information about the entities being declared. For example, affixing a class method with the modifiers `Public`, `Protected`, `Friend`, `Protected Friend`, or `Private` specifies its accessibility.

In addition to the modifiers defined by the language, Visual Basic also enables programmers to create new modifiers, called *attributes*, and to use them when declaring new entities. These new modifiers, which are defined through the declaration of attribute classes, are then assigned to entities through *attribute blocks*.

Note. Attributes may be retrieved at run time through the .NET Framework's reflection APIs. These APIs are outside the scope of this specification.

For instance, a framework might define a `Help` attribute that can be placed on program elements such as classes and methods to provide a mapping from program elements to documentation, as the following example demonstrates:

```
<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
```

The example defines an attribute class named `HelpAttribute`, or `Help` for short, that has one positional parameter (`UrlValue`) and one named argument (`Topic`).

The next example shows several uses of the attribute:

```
<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
```

The next example checks to see if `Class1` has a `Help` attribute, and writes out the associated `Topic` and `Url` values if the attribute is present.

```
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
```

5.1 Attribute Classes

An *attribute class* is a non-generic class that derives from `System.Attribute` and is not `MustInherit`. The attribute class may have a `System.AttributeUsage` attribute that declares what the attribute is valid on, whether it may be used multiple times in a declaration, and whether it is inherited. The following example defines an attribute class named `SimpleAttribute` that can be placed on class declarations and interface declarations:

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

The next example shows a few uses of the `Simple` attribute. Although the attribute class is named `SimpleAttribute`, uses of this attribute may omit the `Attribute` suffix, thus shortening the name to `Simple`:

```
<Simple> Class Class1
End Class

<Simple> Interface Interface1
End Interface
```

If the attribute lacks a `System.AttributeUsage`, then the attribute can be placed onany target (equivalent to `AttributeTargets.All`). The `System.AttributeUsage` attribute has a variable initializer, `AllowMultiple`, which specifies whether the indicated attribute can be specified more than once for a given declaration. If `AllowMultiple` for an attribute is `True`, it is a *multiple-use attribute class*, and can be specified more than once on a declaration. If `AllowMultiple` for an attribute is `False` or unspecified for an attribute, it is a *single-use attribute class*, and can be specified at most once on a declaration.

The following example defines a multiple-use attribute class named `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
```

The example shows a class declaration with two uses of the `Author` attribute:

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

The `System.AttributeUsage` attribute has a public instance variable, `Inherited`, that specifies whether the attribute, when specified on a base type, is also inherited by types that derive from this base type. If the `Inherited` public

instance variable is not initialized, a default value of `False` is used. Properties and events do not inherit attributes, although the methods defined by properties and events do. Interfaces do not inherit attributes.

If a single-use attribute is both inherited and specified on a derived type, the attribute specified on the derived type overrides the inherited attribute. If a multiple-use attribute is both inherited and specified on a derived type, both attributes are specified on the derived type. For example:

```
<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
```

The positional parameters of the attribute are defined by the parameters of the public constructors of the attribute class. Positional parameters must be `ByVal` and may not specify `ByRef`. Public instance variables and properties are defined by public read-write properties or instance variables of the attribute class. The types that can be used in positional parameters and public instance variables and properties are restricted to attribute types. A type is an attribute type if it is one of the following:

- Any primitive type except for `Date` and `Decimal`.
- The type `Object`.
- The type `System.Type`.
- An enumerated type, provided that it and the types in which it is nested (if any) have `Public` accessibility.
- A one-dimensional array of one of the previous types in this list.

5.2 Attribute Blocks

Attributes are specified in *attribute blocks*. Each attribute block is delimited by angle brackets ("<>"), and multiple attributes can be specified in a comma-separated list within an attribute block or in multiple attribute blocks. The order in which attributes are specified is not significant. For example, the attribute blocks `<A, B>`, `<B, A>`, `<A> <B>` and `<B> <A>` are all equivalent.

```
Attributes:
    | AttributeBlock+
    ;
```

```
AttributeBlock:
    | LineTerminator? '<' AttributeList LineTerminator? '>' LineTerminator?
    ;

AttributeList:
    | Attribute ( Comma Attribute  )*
    ;

Attribute:
    | ( AttributeModifier ':'  )? SimpleTypeName
      ( OpenParenthesis AttributeArguments? CloseParenthesis  )?
    ;

AttributeModifier:
    | 'Assembly' | 'Module'
    ;
```

An attribute may not be specified on a kind of declaration it does not support, and single-use attributes may not be specified more than once in an attribute block. The example below causes errors both because it attempts to use `HelpString` on the interface `Interface1` and more than once on the declaration of `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
```

An attribute consists of an optional attribute modifier, an attribute name, an optional list of positional arguments, and variable/property initializers. If there are no parameters or initializers, the parentheses may be omitted. If an attribute has a modifier, it must be in an attribute block at the top of a source file.

If a source file contains an attribute block at the top of the file that specifies attributes for the assembly or module that will contain the source file, each attribute in the attribute block must be prefixed by both the `Assembly` or `Module` modifier and a colon.

5.2.1 Attribute Names

The name of an attribute specifies an attribute class. By convention, attribute classes are named with the suffix `Attribute`. Uses of an attribute may either include or omit this suffix. Consequently the name of an attribute class that corresponds to an attribute identifier is either the identifier itself or the concatenation of the qualified identifier and `Attribute`. When the compiler resolves an attribute name, it appends `Attribute` to the name and tries the lookup. If that lookup fails, the compiler tries the lookup without the suffix. For example, uses of an attribute class `SimpleAttribute` may omit the `Attribute` suffix, thus shortening the name to `Simple`:

```
<Simple> Class Class1
End Class

<Simple> Interface Interface1
End Interface
```

The example above is semantically equivalent to the following:

```
<SimpleAttribute> Class Class1
End Class

<SimpleAttribute> Interface Interface1
End Interface
```

In general, attributes named with the suffix `Attribute` are preferred. The following example shows two attribute classes named `T` and `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
```

Both the attribute block `<T>` and the attribute block `<TAttribute>` refer to the attribute class named `TAttribute`. It is not possible to use `T` as an attribute until you remove the declaration for class `TAttribute`.

5.2.2 Attribute Arguments

Arguments to an attribute may take two forms: *positional arguments* and *instance variable/property initializers*. Any positional arguments to the attribute must precede the instance variable/property initializers. A positional argument consists of a constant expression, a one-dimensional array-creation expression or a `GetType` expression. An instance variable/property initializer consists of an identifier, which can match keywords, followed by a colon and equal sign, and terminated by a constant expression or a `GetType` expression.

Given an attribute with attribute class `T`, positional argument list `P`, and instance variable/property initializer list `N`, these steps determine whether the arguments are valid:

1. Follow the compile-time processing steps for compiling an expression of the form `New T(P)`. This either results in a compile-time error or determines a constructor on `T` that is most applicable to the argument list.

2. If the constructor determined in step 1 has parameters that are not attribute types or is inaccessible at the declaration site, a compile-time error occurs.
3. For each instance variable/property initializer `Arg` in `N`, let `Name` be the identifier of the instance variable/property initializer `Arg`. `Name` must identify a non-`Shared`, writeable, `Public` instance variable or parameterless property on `T` whose type is an attribute type. If `T` has no such instance variable or property, a compile-time error occurs.

For example:

```
<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
```

Type parameters cannot be used anywhere in attribute arguments. However, constructed types may be used:

```
<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? ( Comma AttributeArgumentExpression?  )*
    ;

VariablePropertyInitializerList:
    | VariablePropertyInitializer ( Comma VariablePropertyInitializer  )*
    ;

VariablePropertyInitializer:
    | IdentifierOrKeyword ColonEquals AttributeArgumentExpression
    ;

AttributeArgumentExpression:
    | ConstantExpression
    | GetTypeExpression
    | ArrayExpression
    ;
```

6. Source Files and Namespaces

A Visual Basic program consists of one or more source files. When a program is compiled, all of the source files are processed together; thus, source files can depend on each other, possibly in a circular fashion, without any forward-declaration requirement. The textual order of declarations in the program text is generally of no significance.

A source file consists of an optional set of option statements, import statements, and attributes, which are followed by a namespace body. The attributes, which must each have either the `Assembly` or `Module` modifier, apply to the .NET assembly or module produced by the compilation. The body of the source file functions as an implicit namespace declaration for the global namespace, meaning that all declarations at the top level of a source file are placed in the global namespace. For example:

FileA.vb:

```
Class A
End Class
```

FileB.vb:

```
Class B
End Class
```

The two source files contribute to the global namespace, in this case declaring two classes with the fully qualified names `A` and `B`. Because the two source files contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

Note. The compilation environment may override the namespace declarations into which a source file is implicitly placed.

Except where noted, statements within a Visual Basic program can be terminated either by a line terminator or by a colon.

```
Start:
    | OptionStatement* ImportsStatement* AttributesStatement* NamespaceMemberDeclaration*
    ;

StatementTerminator:
    | LineTerminator
    | ':'
    ;

AttributesStatement:
    | Attributes StatementTerminator
    ;
```

6.1 Program Startup and Termination

Program startup occurs when the execution environment executes a designated method, which is referred to as the program's *entry point*. This entry point method, which must always be named `Main`, must be shared, cannot be contained in a generic type, cannot have the async modifier, and must have one of the following signatures:

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

The accessibility of the entry point method is irrelevant. If a program contains more than one suitable entry point, the compilation environment must designate one as the entry point. Otherwise, a compile-time error occurs. The compilation environment may also create an entry point method if one does not exist.

When a program begins, if the entry point has a parameter, the argument supplied by the execution environment contains the command-line arguments to the program represented as strings. If the entry point has a return type of `Integer`, then the value returned from the function is returned to the execution environment as the result of the program.

In all other respects, entry point methods behave in the same manner as other methods. When execution leaves the invocation of the entry point method made by the execution environment, the program terminates.

6.2 Compilation Options

A source file can specify compilation options in the source code using *option statements*.

```
OptionStatement:
    | OptionExplicitStatement
    | OptionStrictStatement
    | OptionCompareStatement
    | OptionInferStatement
    ;
```

An `Option` statement applies only to the source file in which it appears, and only one of each type of `Option` statement may appear in a source file. For example:

```
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.
```

There are four compilation options: strict type semantics, explicit declaration semantics, comparison semantics, and local variable type inference semantics. If a source file does not include a particular `Option` statement, then the compilation environment determines which particular set of semantics will be used. There is also a fifth compilation option, integer overflow checks, which can only be specified through the compilation environment.

6.2.1 Option Explicit Statement

The `Option Explicit` statement determines whether local variables may be implicitly declared. The keywords `On` or `Off` may follow the statement; if neither is specified, the default is `On`. If no statement is specified in a file, the compilation environment determines which will be used.

```
OptionExplicitStatement:
    | 'Option' 'Explicit' OnOff? StatementTerminator
    ;

OnOff:
    | 'On' | 'Off'
    ;
```

Note. `Explicit` and `Off` are not reserved words.

```
Option Explicit Off

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

In this example, the local variable `x` is implicitly declared by assigning to it. The type of `x` is `Object`.

6.2.2 Option Strict Statement

The `Option Strict` statement determines whether conversions and operations on `Object` are governed by strict or permissive type semantics and whether types are implicitly typed as `Object` if no `As` clause is specified. The statement may be followed by the keywords `On` or `Off`; if neither is specified, the default is `On`. If no statement is specified in a file, the compilation environment determines which will be used.

```
OptionStrictStatement:
    | 'Option' 'Strict' OnOff? StatementTerminator
    ;
```

Note. `Strict` and `Off` are not reserved words.

```
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
```

Under strict semantics, the following are disallowed:

- Narrowing conversions without an explicit cast operator.
- Late binding.
- Operations on type `Object` other than `TypeOf...Is`, `Is`, and `IsNot`.
- Omitting the `As` clause in a declaration that does not have an inferred type.

6.2.3 Option Compare Statement

The `Option Compare` statement determines the semantics of string comparisons. String comparisons are carried out either using binary comparisons (in which the binary Unicode value of each character is compared) or text comparisons (in which the lexical meaning of each character is compared using the current culture). If no statement is specified in a file, the compilation environment controls which type of comparison will be used.

```
OptionCompareStatement:
    | 'Option' 'Compare' CompareOption StatementTerminator
    ;

CompareOption:
    | 'Binary' | 'Text'
    ;
```

Note. `Compare`, `Binary`, and `Text` are not reserved words.

```
Option Compare Text

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

In this case, the string comparison is done using a text comparison that ignores case differences. If `Option Compare Binary` had been specified, then this would have printed `False`.

6.2.4 Integer Overflow Checks

Integer operations can either be checked or not checked for overflow conditions at run time. If overflow conditions are checked and an integer operation overflows, a `System.OverflowException` exception is thrown. If overflow conditions are not checked, integer operation overflows do not throw an exception. The compilation environment determines whether this option is on or off.

6.2.5 Option Infer Statement

The `Option Infer` statement determines whether local variable declarations that have no `As` clause have an inferred type or use `Object`. The statement may be followed by the keywords `On` or `Off`; if neither is specified, the default is `On`. If no statement is specified in a file, the compilation environment determines which will be used.

```
OptionInferStatement:
    | 'Option' 'Infer' OnOff? StatementTerminator
    ;
```

Note. `Infer` and `Off` are not reserved words.

```
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
```

6.3 Imports Statement

`Imports` statements import the names of entities into a source file, allowing the names to be referenced without qualification, or import a namespace for use in XML expressions.

```
ImportsStatement:
    | 'Imports' ImportsClauses StatementTerminator
    ;

ImportsClauses:
    | ImportsClause ( Comma ImportsClause  )*
    ;

ImportsClause:
    | AliasImportsClause
    | MembersImportsClause
    | XMLNamespaceImportsClause
    ;
```

Within member declarations in a source file that contains an `Imports` statement, the types contained in the given namespace can be referenced directly, as seen in the following example:

```
Imports N1.N2

Namespace N1.N2
    Class A
    End Class
End Namespace
```

```
Namespace N3
    Class B
        Inherits A
    End Class
End Namespace
```

Here, within the source file, the type members of namespace `N1.N2` are directly available, and thus class `N3.B` derives from class `N1.N2.A`.

`Imports` statements must appear after any `Option` statements but before any type declarations. The compilation environment may also define implicit `Imports` statements.

`Imports` statements make names available in a source file, but do not declare anything in the global namespace's declaration space. The scope of the names imported by an `Imports` statement extends over the namespace member declarations contained in the source file. The scope of an `Imports` statement specifically does not include other `Imports` statements, nor does it include other source files. `Imports` statements may not refer to one another.

In this example, the last `Imports` statement is in error because it is not affected by the first import alias.

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

Namespace N1.N2
End Namespace
```

Note. The namespace or type names that appear in `Imports` statements are always treated as if they are fully qualified. That is, the leftmost identifier in a namespace or type name always resolves in the global namespace and the rest of the resolution proceeds according to normal name resolution rules. This is the only place in the language that applies such a rule; the rule ensures that a name cannot be completely hidden from qualification. Without the rule, if a name in the global namespace were hidden in a particular source file, it would be impossible to specify any names from that namespace in a qualified way.

In this example, the `Imports` statement always refers to the global `System` namespace, and not the class in the source file.

```
Imports System   ' Imports the namespace, not the class.

Class System
End Class
```

6.3.1 Import Aliases

An *import alias* defines an alias for a namespace or type.

```
AliasImportsClause:
    | Identifier Equals TypeName
    ;
```

```
Imports A = N1.N2.A

Namespace N1.N2
    Class A
    End Class
End Namespace

Namespace N3
    Class B
        Inherits A
    End Class
End Namespace
```

Here, within the source file, `A` is an alias for `N1.N2.A`, and thus class `N3.B` derives from class `N1.N2.A`. The same effect can be obtained by creating an alias `R` for `N1.N2` and then referencing `R.A`:

```
Imports R = N1.N2

Namespace N3
    Class B
        Inherits R.A
    End Class
End Namespace
```

The identifier of an import alias must be unique within the declaration space of the global namespace (not just the global namespace declaration in the source file in which the import alias is defined), even though it does not declare a name in the global namespace's declaration space.

Note. Declarations in a module do not introduce names into the containing declaration space. Thus, it is valid for a declaration in a module to have the same name as an import alias, even though the declaration's name will be accessible in the containing declaration space.

```
' Error: Alias A conflicts with typename A
Imports A = N3.A

Class A
End Class

Namespace N3
    Class A
    End Class
End Namespace
```

Here, the global namespace already contains a member `A`, so it is an error for an import alias to use that identifier. It is likewise an error for two or more import aliases in the same source file to declare aliases by the same name.

An import alias can create an alias for any namespace or type. Accessing a namespace or type through an alias yields exactly the same result as accessing the namespace or type through its declared name.

```
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
```

Here, the names `N1.N2.A`, `R1.N2.A`, and `R2.A` are equivalent, and all refer to the class whose fully qualified name is `N1.N2.A`.

The import specifies the exact name of the namespace or type to which it is creating an alias. This must be the exact fully qualified name of that namespace or type: it does not use the normal rules for qualified name resolution (which for instance allow access to the members of a base class through a derived class).

If an import alias points to a type or namespace which cannot be resolved by these rules, then the import statement is ignored (and the compiler gives a warning).

Also, the reference cannot be to an open generic type -- all generic types must have valid type arguments supplied, and all type arguments must be resolvable by the rules above. Any incorrect binding of a generic type is an 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
```

Declarations in the source file may shadow the import alias name.

```
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
```

In the preceding example the reference to `R.A` in the declaration of `B` causes an error because `R` refers to `N3.R`, not `N1.N2`.

An import alias makes an alias available within a particular source file, but it does not contribute any new members to the underlying declaration space. In other words, an import alias is not transitive, but rather affects only the source file in which it occurs.

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
```

In the above example, because the scope of the import alias that introduces `R` only extends to declarations in the source file in which it is contained, `R` is unknown in the second source file.

6.3.2 Namespace Imports

A *namespace import* imports all of the members of a namespace or type, allowing the identifier of each member of the namespace or type to be used without qualification. In the case of types, a namespace import only allows access to the shared members of the type without requiring qualification of the class name. In particular, it allows the members of enumerated types to be used without qualification.

```
MembersImportsClause:
    | TypeName
    ;
```

For example:

```
Imports Colors

Enum Colors
    Red
    Green
    Blue
End Enum

Module M1
    Sub Main()
        Dim c As Colors = Red
    End Sub
End Module
```

Unlike an import alias, a namespace import has no restrictions on the names it imports and may import namespaces and types whose identifiers are already declared within the global namespace. The names imported by a regular import are shadowed by import aliases and declarations in the source file.

In the following example, `A` refers to `N3.A` rather than `N1.N2.A` within member declarations in the `N3` namespace.

```
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
```

When more than one imported namespace contains members by the same name (and that name is not otherwise shadowed by an import alias or declaration), a reference to that name is ambiguous and causes a compile-time error.

```
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
```

In the above example, both `N1` and `N2` contain a member `A`. Because `N3` imports both, referencing `A` in `N3` causes a compile-time error. In this situation, the conflict can be resolved either through qualification of references to `A`, or by introducing an import alias that picks a particular `A`, as in the following example:

```
Imports N1
Imports N2
Imports A = N1.A

Namespace N3
    Class B
        Inherits A ' A means N1.A.
    End Class
End Namespace
```

Only namespaces, classes, structures, enumerated types, and standard modules may be imported.

6.3.3 XML Namespace Imports

An *XML namespace import* defines a namespace or the default namespace for unqualified XML expressions contained within the compilation unit.

```
XMLNamespaceImportsClause:
    | '<' XMLNamespaceAttributeName XMLWhitespace? Equals XMLWhitespace?
      XMLNamespaceValue '>'
    ;

XMLNamespaceValue:
    | DoubleQuoteCharacter XMLAttributeDoubleQuoteValueCharacter* DoubleQuoteCharacter
    | SingleQuoteCharacter XMLAttributeSingleQuoteValueCharacter* SingleQuoteCharacter
    ;
```

For example:

```
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
```

An XML namespace, including the default namespace, can only be defined once for a particular set of imports. For example:

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

6.4 Namespaces

Visual Basic programs are organized using namespaces. Namespaces both internally organize a program as well as organize the way program elements are exposed to other programs.

Unlike other entities, namespaces are open-ended, and may be declared multiple times within the same program and across many programs, with each declaration contributing members to the same namespace. In the following example, the two namespace declarations contribute to the same declaration space, declaring two classes with the fully qualified names `N1.N2.A` and `N1.N2.B`.

```
Namespace N1.N2
    Class A
    End Class
End Namespace

Namespace N1.N2
    Class B
    End Class
End Namespace
```

Because the two declarations contribute to the same declaration space, it would be an error if each contained a declaration of a member with the same name.

There is a global namespace that has no name and whose nested namespaces and types can always be accessed without qualification. The scope of a namespace member declared in the global namespace is the entire program text. Otherwise, the scope of a type or namespace declared in a namespace whose fully qualified name is `N` is the program text of each namespace whose corresponding namespace's fully qualified name begins with `N` or is `N` itself. (Note that a compiler can choose to put declarations in a particular namespace by default. This does not alter the fact that there is still a global, unnamed namespace.)

In this example, the class `B` can see the class `A` because `B`'s namespace `N1.N2.N3` is conceptually nested within the namespace `N1.N2`.

```
Namespace N1.N2
    Class A
    End Class
End Namespace

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

6.4.1 Namespace Declarations

There are three forms of namespace declaration.

```
NamespaceDeclaration:
    | 'Namespace' NamespaceName StatementTerminator
      NamespaceMemberDeclaration*
      'End' 'Namespace' StatementTerminator
    ;
```

```
NamespaceName:
    | RelativeNamespaceName
    | 'Global'
    | 'Global' '.' RelativeNamespaceName
    ;

RelativeNamespaceName:
    | Identifier ( Period IdentifierOrKeyword  )*
    ;
```

The first form starts with the keyword `Namespace` followed by a relative namespace name. If the relative namespace name is qualified, the namespace declaration is treated as if it is lexically nested within namespace declarations corresponding to each name in the qualified name. For example, the following two namespaces are semantically equivalent:

```
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
```

The second form starts with the keywords `Namespace Global`. It is treated as if all its member declarations were lexically placed in the global unnamed namespace -- regardless of any defaults provided by the compilation environment. This form of namespace declaration may not be lexically nested within any other namespace declaration.

The third form starts with the keywords `Namespace Global` followed by a qualified identifier `N`. It is treated as if it were a namespace declaration of the first form "`Namespace N`" that was lexically placed in the global unnamed namespace -- regardless of any defaults provided by the compilation environment. This form of namespace declaration may not be lexically nested within any other namespace declaration.

```
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
```

When dealing with the members of a namespace, it is not important where a particular member is declared. If two programs define an entity with the same name in the same namespace, attempting to resolve the name in the namespace causes an ambiguity error.

Namespaces are by definition `Public`, so a namespace declaration cannot include any access modifiers.

6.4.2 Namespace Members

Namespace members can only be namespace declarations and type declarations. Type declarations may have `Public` or `Friend` access. The default access for types is `Friend` access.

```
NamespaceMemberDeclaration:
    | NamespaceDeclaration
    | TypeDeclaration
    ;

TypeDeclaration:
    | ModuleDeclaration
    | NonModuleDeclaration
    ;

NonModuleDeclaration:
    | EnumDeclaration
    | StructureDeclaration
    | InterfaceDeclaration
    | ClassDeclaration
    | DelegateDeclaration
    ;
```

7. Types

The two fundamental categories of types in Visual Basic are *value types* and *reference types*. Primitive types (except strings), enumerations, and structures are value types. Classes, strings, standard modules, interfaces, arrays, and delegates are reference types.

Every type has a *default value*, which is the value that is assigned to variables of that type upon initialization.

```
TypeName:
    | ArrayTypeName
    | NonArrayTypeName
    ;

NonArrayTypeName:
    | SimpleTypeName
    | NullableTypeName
    ;

SimpleTypeName:
    | QualifiedTypeName
    | BuiltInTypeName
    ;

QualifiedTypeName:
    | Identifier TypeArguments? ( Period IdentifierOrKeyword TypeArguments? )*
    | 'Global' Period IdentifierOrKeyword TypeArguments?
      ( Period IdentifierOrKeyword TypeArguments? )*
    ;

TypeArguments:
    | OpenParenthesis 'Of' TypeArgumentList CloseParenthesis
    ;

TypeArgumentList:
    | TypeName ( Comma TypeName  )*
    ;

BuiltInTypeName:
    | 'Object'
    | PrimitiveTypeName
    ;

TypeModifier:
    | AccessModifier
    | 'Shadows'
    ;

IdentifierModifiers:
    | NullableNameModifier? ArrayNameModifier?
    ;
```

7.1 Value Types and Reference Types

Although value types and reference types can be similar in terms of declaration syntax and usage, their semantics are distinct.

Reference types are stored on the run-time heap; they may only be accessed through a reference to that storage. Because reference types are always accessed through references, their lifetime is managed by the .NET Framework. Outstanding references to a particular instance are tracked and the instance is destroyed only when no more references remain. A variable of reference type contains a reference to a value of that type, a value of a more derived type, or a null value. A *null value* refers to nothing; it is not possible to do anything with a null value except assign it. Assignment to a variable of a reference type creates a copy of the reference rather than a copy of the referenced value. For a variable of a reference type, the default value is a null value.

Value types are stored directly on the stack, either within an array or within another type; their storage can only be accessed directly. Because value types are stored directly within variables, their lifetime is determined by the lifetime of the variable that contains them. When the location containing a value type instance is destroyed, the value type instance is also destroyed. Value types are always accessed directly; it is not possible to create a reference to a value type. Prohibiting such a reference makes it impossible to refer to a value class instance that has been destroyed. Because value types are always `NotInheritable`, a variable of a value type always contains a value of that type. Because of this, the value of a value type cannot be a null value, nor can it reference an object of a more derived type. Assignment to a variable of a value type creates a copy of the value being assigned. For a variable of a value type, the default value is the result of initializing each variable member of the type to its default value.

The following example shows the difference between reference types and value types:

```
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
```

The output of the program is:

```
Values: 0, 123
Refs: 123, 123
```

The assignment to the local variable `val2` does not impact the local variable `val1` because both local variables are of a value type (the type `Integer`) and each local variable of a value type has its own storage. In contrast, the assignment `ref2.Value = 123;` affects the object that both `ref1` and `ref2` reference.

One thing to note about the .NET Framework type system is that even though structures, enumerations and primitive types (except for `String`) are value types, they all inherit from reference types. Structures and the primitive types inherit from the reference type `System.ValueType`, which inherits from `Object`. Enumerated types inherit from the reference type `System.Enum`, which inherits from `System.ValueType`.

7.1.1 Nullable Value Types

For value types, a `?` modifier can be added to a type name to represent the *nullable* version of that type.

```
NullableTypeName:
    | NonArrayTypeName '?'
    ;

NullableNameModifier:
```

```
    | '?'
    ;
```

A nullable value type can contain the same values as the non-nullable version of the type as well as the null value. Thus, for a nullable value type, assigning `Nothing` to a variable of the type sets the value of the variable to the null value, not the zero value of the value type. For example:

```
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)
```

A variable may also be declared to be of a nullable value type by putting a nullable type modifier on the variable name. For clarity, it is not valid to have a nullable type modifier on both a variable name and a type name in the same declaration. Since nullable types are implemented using the type `System.Nullable(Of T)`, the type `T?` is synonymous to the type `System.Nullable(Of T)`, and the two names can be used interchangeably. The `?` modifier cannot be placed on a type that is already nullable; thus, it is not possible to declare the type `Integer??` or `System.Nullable(Of Integer)?`.

A nullable value type `T?` has the members of `System.Nullable(Of T)` as well as any operators or conversions *lifted* from the underlying type `T` into the type `T?`. Lifting copies operators and conversions from the underlying type, in most cases substituting nullable value types for non-nullable value types. This allows many of the same conversions and operations that apply to `T` to apply to `T?` as well.

7.2 Interface Implementation

Structure and class declarations may declare that they implement a set of interface types through one or more `Implements` clauses.

```
TypeImplementsClause:
    | 'Implements' TypeImplements StatementTerminator
    ;

TypeImplements:
    | NonArrayTypeName ( Comma NonArrayTypeName  )*
    ;
```

All the types specified in the `Implements` clause must be interfaces, and the type must implement all members of the interfaces. For example:

```
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
```

A type that implements an interface also implicitly implements all of the interface's base interfaces. This is true even if the type does not explicitly list all base interfaces in the `Implements` clause. In this example, the `TextBox` structure implements both `IControl` and `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
```

Declaring that a type implements an interface in and of itself does not declare anything in the declaration space of the type. Thus, it is valid to implement two interfaces with a method by the same name.

Types cannot implement a type parameter on its own, although it may involve the type parameters that are in scope.

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

        ...
    End Class
```

Generic interfaces can be implemented multiple times using different type arguments. However, a generic type cannot implement a generic interface using a type parameter if the supplied type parameter (regardless of type constraints) could overlap with another implementation of that interface. For example:

```
    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
```

7.3 Primitive Types

The *primitive types* are identified through keywords, which are aliases for predefined types in the `System` namespace. A primitive type is completely indistinguishable from the type it aliases: writing the reserved word `Byte` is exactly the same as writing `System.Byte`. Primitive types are also known as *intrinsic types*.

```
PrimitiveTypeName:
    | NumericTypeName
    | 'Boolean'
    | 'Date'
    | 'Char'
    | 'String'
    ;

NumericTypeName:
    | IntegralTypeName
    | FloatingPointTypeName
    | 'Decimal'
    ;

IntegralTypeName:
    | 'Byte' | 'SByte' | 'UShort' | 'Short' | 'UInteger'
    | 'Integer' | 'ULong' | 'Long'
    ;

FloatingPointTypeName:
    | 'Single' | 'Double'
    ;
```

Because a primitive type aliases a regular type, every primitive type has members. For example, `Integer` has the members declared in `System.Int32`. Literals can be treated as instances of their corresponding types.

- The primitive types differ from other structure types in that they permit certain additional operations:
- Primitive types permit values to be created by writing literals. For example, `123I` is a literal of type `Integer`.
- It is possible to declare constants of the primitive types.
- When the operands of an expression are all primitive type constants, it is possible for the compiler to evaluate the expression at compile time. Such an expression is known as a constant expression.

Visual Basic defines the following primitive types:

- The integral value types `Byte` (1-byte unsigned integer), `SByte` (1-byte signed integer), `UShort` (2-byte unsigned integer), `Short` (2-byte signed integer), `UInteger` (4-byte unsigned integer), `Integer` (4-byte signed integer), `ULong` (8-byte unsigned integer), and `Long` (8-byte signed integer). These types map to `System.Byte`, `System.SByte`, `System.UInt16`, `System.Int16`, `System.UInt32`, `System.Int32`, `System.UInt64` and `System.Int64`, respectively. The default value of an integral type is equivalent to the literal `0`.
- The floating-point value types `Single` (4-byte floating point) and `Double` (8-byte floating point). These types map to `System.Single` and `System.Double`, respectively. The default value of a floating-point type is equivalent to the literal `0`.
- The `Decimal` type (16-byte decimal value), which maps to `System.Decimal`. The default value of decimal is equivalent to the literal `0D`.

- The `Boolean` value type, which represents a truth value, typically the result of a relational or logical operation. The literal is of type `System.Boolean`. The default value of the `Boolean` type is equivalent to the literal `False`.
- The `Date` value type, which represents a date and/or a time and maps to `System.DateTime`. The default value of the `Date` type is equivalent to the literal `# 01/01/0001 12:00:00AM #`.
- The `Char` value type, which represents a single Unicode character and maps to `System.Char`. The default value of the `Char` type is equivalent to the constant expression `ChrW(0)`.
- The `String` reference type, which represents a sequence of Unicode characters and maps to `System.String`. The default value of the `String` type is a null value.

7.4 Enumerations

Enumerations are value types that inherit from `System.Enum` and symbolically represent a set of values of one of the primitive integral types.

```
EnumDeclaration:
    | Attributes? TypeModifier* 'Enum' Identifier
      ( 'As' NonArrayTypeName  )? StatementTerminator
      EnumMemberDeclaration+
      'End' 'Enum' StatementTerminator
    ;
```

For an enumeration type `E`, the default value is the value produced by the expression `CType(0, E)`.

The underlying type of an enumeration must be an integral type that can represent all the enumerator values defined in the enumeration. If an underlying type is specified, it must be `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, or one of their corresponding types in the `System` namespace. If no underlying type is explicitly specified, the default is `Integer`.

The following example declares an enumeration with an underlying type of `Long`:

```
Enum Color As Long
    Red
    Green
    Blue
End Enum
```

A developer might choose to use an underlying type of `Long`, as in the example, to enable the use of values that are in the range of `Long`, but not in the range of `Integer`, or to preserve this option for the future.

7.4.1 Enumeration Members

The members of an enumeration are the enumerated values declared in the enumeration and the members inherited from class `System.Enum`.

The scope of an enumeration member is the enumeration declaration body. This means that outside of an enumeration declaration, an enumeration member must always be qualified (unless the type is specifically imported into a namespace through a namespace import).

Declaration order for enumeration member declarations is significant when constant expression values are omitted. Enumeration members implicitly have `Public` access only; no access modifiers are allowed on enumeration member declarations.

```
EnumMemberDeclaration:
    | Attributes? Identifier ( Equals ConstantExpression  )? StatementTerminator
    ;
```

7.4.2 Enumeration Values

The enumerated values in an enumeration member list are declared as constants typed as the underlying enumeration type, and they can appear wherever constants are required. An enumeration member definition with `=` gives the associated member the value indicated by the constant expression. The constant expression must evaluate to an integral type that is implicitly convertible to the underlying type and must be within the range of values that can be represented by the underlying type. The following example is in error because the constant values `1.5`, `2.3`, and `3.3` are not implicitly convertible to the underlying integral type `Long` with strict semantics.

```
Option Strict On

Enum Color As Long
    Red = 1.5
    Green = 2.3
    Blue = 3.3
End Enum
```

Multiple enumeration members may share the same associated value, as shown below:

```
Enum Color
    Red
    Green
    Blue
    Max = Blue
End Enum
```

The example shows an enumeration that has two enumeration members -- `Blue` and `Max` -- that have the same associated value.

If the first enumerator value definition in the enumeration has no initializer, the value of the corresponding constant is `0`. An enumeration value definition without an initializer gives the enumerator the value obtained by increasing the value of the previous enumeration value by `1`. This increased value must be within the range of values that can be represented by the underlying type.

```
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
```

The example above prints the enumeration values and their associated values. The output is:

```
Red = 0
Green = 10
Blue = 11
```

The reasons for the values are as follows:

- The enumeration value `Red` is automatically assigned the value `0` (since it has no initializer and is the first enumeration value member).
- The enumeration value `Green` is explicitly given the value `10`.
- The enumeration value `Blue` is automatically assigned the value one greater than the enumeration value that textually precedes it.

The constant expression may not directly or indirectly use the value of its own associated enumeration value (that is, circularity in the constant expression is not allowed). The following example is invalid because the declarations of `A` and `B` are circular.

```
Enum Circular
    A = B
    B
End Enum
```

`A` depends on `B` explicitly, and `B` depends on `A` implicitly.

7.5 Classes

A *class* is a data structure that may contain data members (constants, variables, and events), function members (methods, properties, indexers, operators, and constructors), and nested types. Classes are reference types.

```
ClassDeclaration:
    | Attributes? ClassModifier* 'Class' Identifier TypeParameterList? StatementTerminator
      ClassBase?
      TypeImplementsClause*
      ClassMemberDeclaration*
      'End' 'Class' StatementTerminator
    ;

ClassModifier:
    | TypeModifier
    | 'MustInherit'
    | 'NotInheritable'
    | 'Partial'
    ;
```

The following example shows a class that contains each kind of member:

```
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
```

The following example shows uses of these members:

```
    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
```

There are two class-specific modifiers, `MustInherit` and `NotInheritable`. It is invalid to specify them both.

7.5.1 Class Base Specification

A class declaration may include a base type specification that defines the direct base type of the class.

```
ClassBase:
    | 'Inherits' NonArrayTypeName StatementTerminator
    ;
```

If a class declaration has no explicit base type, the direct base type is implicitly `Object`. For example:

```
Class Base
End Class

Class Derived
    Inherits Base
End Class
```

Base types cannot be a type parameter on its own, although it may involve the type parameters that are in scope.

```
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
```

Classes may only derive from `Object` and classes. It is invalid for a class to derive from `System.ValueType`, `System.Enum`, `System.Array`, `System.MulticastDelegate` or `System.Delegate`. A generic class cannot derive from `System.Attribute` or from a class that derives from it.

Every class has exactly one direct base class, and circularity in derivation is prohibited. It is not possible to derive from a `NotInheritable` class, and the accessibility domain of the base class must be the same as or a superset of the accessibility domain of the class itself.

7.5.2 Class Members

The members of a class consist of the members introduced by its class member declarations and the members inherited from its direct base class.

```
ClassMemberDeclaration:
    | NonModuleDeclaration
    | EventMemberDeclaration
    | VariableMemberDeclaration
    | ConstantMemberDeclaration
    | MethodMemberDeclaration
    | PropertyMemberDeclaration
    | ConstructorMemberDeclaration
```

```
    | OperatorDeclaration
    ;
```

A class member declaration may have `Public`, `Protected`, `Friend`, `Protected Friend`, or `Private` access. When a class member declaration does not include an access modifier, the declaration defaults to `Public` access, unless it is a variable declaration; in that case it defaults to `Private` access.

The scope of a class member is the class body in which the member declaration occurs, plus the constraint list of that class (if it is generic and has constraints). If the member has `Friend` access, its scope extends to the class body of any derived class in the same program or any assembly that has been given `Friend` access, and if the member has `Public`, `Protected`, or `Protected Friend` access, its scope extends to the class body of any derived class in any program.

7.6 Structures

Structures are value types that inherit from `System.ValueType`. Structures are similar to classes in that they represent data structures that can contain data members and function members. Unlike classes, however, structures do not require heap allocation.

```
StructureDeclaration:
    | Attributes? StructureModifier* 'Structure' Identifier
      TypeParameterList? StatementTerminator
      TypeImplementsClause*
      StructMemberDeclaration*
      'End' 'Structure' StatementTerminator
    ;

StructureModifier:
    | TypeModifier
    | 'Partial'
    ;
```

In the case of classes, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With structures, the variables each have their own copy of the non-`Shared` data, so it is not possible for operations on one to affect the other, as the following example illustrates:

```
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
```

Given the above declaration the following code outputs the value `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
```

The assignment of `a` to `b` creates a copy of the value, and `b` is thus unaffected by the assignment to `a.x`. Had `Point` instead been declared as a class, the output would be `100` because `a` and `b` would reference the same object.

7.6.1 Structure Members

The members of a structure are the members introduced by its structure member declarations and the members inherited from `System.ValueType`.

```
StructMemberDeclaration:
    | NonModuleDeclaration
    | VariableMemberDeclaration
    | ConstantMemberDeclaration
    | EventMemberDeclaration
    | MethodMemberDeclaration
    | PropertyMemberDeclaration
    | ConstructorMemberDeclaration
    | OperatorDeclaration
    ;
```

Every structure implicitly has a `Public` parameterless instance constructor that produces the default value of the structure. As a result, it is not possible for a structure type declaration to declare a parameterless instance constructor. A structure type is, however, permitted to declare *parameterized* instance constructors, as in the following example:

```
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
```

Given the above declaration, the following statements both create a `Point` with `x` and `y` initialized to zero.

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

Because structures directly contain their field values (rather than references to those values), structures cannot contain fields that directly or indirectly reference themselves. For example, the following code is not valid:

```
Structure S1
    Dim f1 As S2
End Structure

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

Normally, a structure member declaration may only have `Public`, `Friend`, or `Private` access, but when overriding members inherited from `Object`, `Protected` and `Protected Friend` access may also be used. When a structure member declaration does not include an access modifier, the declaration defaults to `Public` access. The scope of a member declared by a structure is the structure body in which the declaration occurs, plus the constraints of that structure (if it was generic and had constraints).

7.7 Standard Modules

A *standard module* is a type whose members are implicitly `Shared` and scoped to the declaration space of the standard module's containing namespace, rather than just to the standard module declaration itself. Standard modules may never be instantiated. It is an error to declare a variable of a standard module type.

```
ModuleDeclaration:
    | Attributes? TypeModifier* 'Module' Identifier StatementTerminator
```

```
    ModuleMemberDeclaration*
    'End' 'Module' StatementTerminator
  ;
```

A member of a standard module has two fully qualified names, one without the standard module name and one with the standard module name. More than one standard module in a namespace may define a member with a particular name; unqualified references to the name outside of either module are ambiguous. For example:

```
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
```

A module may only be declared in a namespace and may not be nested in another type. Standard modules may not implement interfaces, they implicitly derive from `Object`, and they have only `Shared` constructors.

7.7.1 Standard Module Members

The members of a standard module are the members introduced by its member declarations and the members inherited from `Object`. Standard modules may have any type of member except instance constructors. All standard module type members are implicitly `Shared`.

```
ModuleMemberDeclaration:
    | NonModuleDeclaration
    | VariableMemberDeclaration
    | ConstantMemberDeclaration
    | EventMemberDeclaration
    | MethodMemberDeclaration
    | PropertyMemberDeclaration
    | ConstructorMemberDeclaration
    ;
```

Normally, a standard module member declaration may only have `Public`, `Friend`, or `Private` access, but when overriding members inherited from `Object`, the `Protected` and `Protected Friend` access modifiers may be specified. When a standard module member declaration does not include an access modifier, the declaration defaults to `Public` access, unless it is a variable, which defaults to `Private` access.

As previously noted, the scope of a standard module member is the declaration containing the standard module declaration. Members inherited from `Object` are not included in this special scoping; those members have no scope

and must always be qualified with the name of the module. If the member has `Friend` access, its scope extends only to namespace members declared in the same program or assemblies that have been given `Friend` access.

7.8 Interfaces

Interfaces are reference types that other types implement to guarantee that they support certain methods. An interface is never directly created and has no actual representation -- other types must be converted to an interface type. An interface defines a contract. A class or structure that implements an interface must adhere to its contract.

```
InterfaceDeclaration:
    | Attributes? TypeModifier* 'Interface' Identifier
      TypeParameterList? StatementTerminator
      InterfaceBase*
      InterfaceMemberDeclaration*
      'End' 'Interface' StatementTerminator
    ;
```

The following example shows an interface that contains a default property `Item`, an event `E`, a method `F`, and a property `P`:

```
        Interface IExample
            Default Property Item(index As Integer) As String

            Event E()

            Sub F(value As Integer)

            Property P() As String
        End Interface
```

Interfaces may employ multiple inheritance. In the following example, the interface `IComboBox` inherits from both `ITextBox` and `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
```

Classes and structures can implement multiple interfaces. In the following example, the class `EditBox` derives from the class `Control` and implements both `IControl` and `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
```

7.8.1 Interface Inheritance

The base interfaces of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on. If an interface declaration has no explicit interface base, then there is no base interface for the type -- interfaces do not inherit from `Object` (although they do have a widening conversion to `Object`).

```
InterfaceBase:
    | 'Inherits' InterfaceBases StatementTerminator
    ;

InterfaceBases:
    | NonArrayTypeName ( Comma NonArrayTypeName  )*
    ;
```

In the following example, the base interfaces of `IComboBox` are `IControl`, `ITextBox`, and `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
```

An interface inherits all members of its base interfaces. In other words, the `IComboBox` interface above inherits members `SetText` and `SetItems` as well as `Paint`.

A class or structure that implements an interface also implicitly implements all of the interface's base interfaces.

If an interface appears more than once in the transitive closure of the base interfaces, it only contributes its members to the derived interface once. A type implementing the derived interface only has to implement the methods of the multiply defined base interface once. In the following example, `Paint` only needs to be implemented once, even though the class implements `IComboBox` and `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
```

An `Inherits` clause has no effect on other `Inherits` clauses. In the following example, `IDerived` must qualify the name of `INested` with `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
```

The accessibility domain of a base interface must be the same as or a superset of the accessibility domain of the interface itself.

7.8.2 Interface Members

The members of an interface consist of the members introduced by its member declarations and the members inherited from its base interfaces.

```
InterfaceMemberDeclaration:
    | NonModuleDeclaration
    | InterfaceEventMemberDeclaration
    | InterfaceMethodMemberDeclaration
    | InterfacePropertyMemberDeclaration
    ;
```

Although interfaces do not inherit members from `Object`, because every class or structure that implements an interface does inherit from `Object`, the members of `Object`, including extension methods, are considered members of an interface and can be called on an interface directly without requiring a cast to `Object`. For example:

```
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).

7.9 Arrays

An *array* is a reference type that contains variables accessed through *indices* corresponding in a one-to-one fashion with the order of the variables in the array. The variables contained in an array, also called the *elements* of the array, must all be of the same type, and this type is called the *element type* of the array.

```
ArrayTypeName:
    | NonArrayTypeName ArrayTypeModifiers
    ;

ArrayTypeModifiers:
    | ArrayTypeModifier+
    ;

ArrayTypeModifier:
    | OpenParenthesis RankList? CloseParenthesis
    ;

RankList:
    | Comma*
    ;

ArrayNameModifier:
    | ArrayTypeModifiers
    | ArraySizeInitializationModifier
    ;
```

The elements of an array come into existence when an array instance is created, and cease to exist when the array instance is destroyed. Each element of an array is initialized to the default value of its type. The type `System.Array` is the base type of all array types and may not be instantiated. Every array type inherits the members declared by the `System.Array` type and is convertible to it (and `Object`). A one-dimensional array type with element `T` also implements the interfaces `System.Collections.Generic.IList(Of T)` and `IReadOnlyList(Of T)`; if `T` is a reference type, then the array type also implements `IList(Of U)` and `IReadOnlyList(Of U)` for any `U` that has a widening reference conversion from `T`.

An array has a *rank* that determines the number of indices associated with each array element. The rank of an array determines the number of *dimensions* of the array. For example, an array with a rank of one is called a single-dimensional array, and an array with a rank greater than one is called a multidimensional array.

The following example creates a single-dimensional array of integer values, initializes the array elements, and then prints each of them out:

```
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
```

The program outputs the following:

```
arr(0) = 0
arr(1) = 1
arr(2) = 4
arr(3) = 9
arr(4) = 16
arr(5) = 25
```

Each dimension of an array has an associated length. Dimension lengths are not part of the type of the array, but rather are established when an instance of the array type is created at run time. The length of a dimension determines the valid range of indices for that dimension: for a dimension of length `N`, indices can range from zero to `N-1`. If a dimension is of length zero, there are no valid indices for that dimension. The total number of elements in an array is the product of the lengths of each dimension in the array. If any of the dimensions of an array has a length of zero, the array is said to be empty. The element type of an array can be any type.

Array types are specified by adding a modifier to an existing type name. The modifier consists of a left parenthesis, a set of zero or more commas, and a right parenthesis. The type modified is the element type of the array, and the number of dimensions is the number of commas plus one. If more than one modifier is specified, then the element type of the array is an array. The modifiers are read left to right, with the leftmost modifier being the outermost array. In the example

```
Module Test
    Dim arr As Integer(,)(,,)()
End Module
```

the element type of `arr` is a two-dimensional array of three-dimensional arrays of one-dimensional arrays of `Integer`.

A variable may also be declared to be of an array type by putting an array type modifier or an array-size initialization modifier on the variable name. In that case, the array element type is the type given in the declaration, and the array dimensions are determined by the variable name modifier. For clarity, it is not valid to have an array type modifier on both a variable name and a type name in the same declaration.

The following example shows a variety of local variable declarations that use array types with `Integer` as the element type:

```
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
```

An array type name modifier extends to all sets of parentheses that follow it. This means that in the situations where a set of arguments enclosed in parenthesis is allowed after a type name, it is not possible to specify the arguments for an array type name. For example:

```
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
```

In the last case, `(3)` is interpreted as part of the type name rather than as a set of constructor arguments.

7.10 Delegates

A *delegate* is a reference type that refers to a `Shared` method of a type or to an instance method of an object.

```
DelegateDeclaration:
    | Attributes? TypeModifier* 'Delegate' MethodSignature StatementTerminator
    ;

MethodSignature:
    | SubSignature
    | FunctionSignature
    ;
```

The closest equivalent of a delegate in other languages is a function pointer, but whereas a function pointer can only reference `Shared` functions, a delegate can reference both `Shared` and instance methods. In the latter case, the delegate stores not only a reference to the method's entry point, but also a reference to the object instance with which to invoke the method.

The delegate declaration may not have a `Handles` clause, an `Implements` clause, a method body, or an `End` construct. The parameter list of the delegate declaration may not contain `Optional` or `ParamArray` parameters. The accessibility domain of the return type and parameter types must be the same as or a superset of the accessibility domain of the delegate itself.

The members of a delegate are the members inherited from class `System.Delegate`. A delegate also defines the following methods:

- A constructor that takes two parameters, one of type `Object` and one of type `System.IntPtr`.
- An `Invoke` method that has the same signature as the delegate.
- A `BeginInvoke` method whose signature is the delegate signature, with three differences. First, the return type is changed to `System.IAsyncResult`. Second, two additional parameters are added to the end of the parameter list: the first of type `System.AsyncCallback` and the second of type `Object`. And finally, all `ByRef` parameters are changed to be `ByVal`.
- An `EndInvoke` method whose return type is the same as the delegate. The parameters of the method are only the delegate parameters exactly that are `ByRef` parameters, in the same order they occur in the delegate signature. In addition to those parameters, there is an additional parameter of type `System.IAsyncResult` at the end of the parameter list.

There are three steps in defining and using delegates: declaration, instantiation, and invocation.

Delegates are declared using delegate declaration syntax. The following example declares a delegate named `SimpleDelegate` that takes no arguments:

```
Delegate Sub SimpleDelegate()
```

The next example creates a `SimpleDelegate` instance and then immediately calls it:

```
Module Test
    Sub F()
        System.Console.WriteLine("Test.F")
    End Sub

    Sub Main()
        Dim d As SimpleDelegate = AddressOf F
        d()
    End Sub
End Module
```

There is not much point in instantiating a delegate for a method and then immediately calling via the delegate, as it would be simpler to call the method directly. Delegates show their usefulness when their anonymity is used. The next example shows a `MultiCall` method that repeatedly calls a `SimpleDelegate` instance:

```
Sub MultiCall(d As SimpleDelegate, count As Integer)
    Dim i As Integer

    For i = 0 To count - 1
        d()
    Next i
End Sub
```

It is unimportant to the `MultiCall` method what the target method for the `SimpleDelegate` is, what accessibility this method has, or whether the method is `Shared` or not. All that matters is that the signature of the target method is compatible with `SimpleDelegate`.

7.11 Partial types

Class and structure declarations can be *partial* declarations. A partial declaration may or may not fully describe the declared type within the declaration. Instead, the declaration of the type may be spread across multiple partial declarations within the program; partial types cannot be declared across program boundaries. A partial type declaration specifies the `Partial` modifier on the declaration. Then, any other declarations in the program for a type with the same fully-qualified name will be merged together with the partial declaration at compile-time to form a single type declaration. For example, the following code declares a single class `Test` with members `Test.C1` and `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
```

When combining partial type declarations, at least one of the declarations must have a `Partial` modifier, otherwise a compile-time error results.

Note. Although it is possible to specify `Partial` on only one declaration among many partial declarations, it is better form to specify it on all partial declarations. In the situation where one partial declaration is visible but one or more partial declarations are hidden (such as the case of extending tool-generated code), it is acceptable to leave the `Partial` modifier off of the visible declaration but specify it on the hidden declarations.

Only classes and structures can be declared using partial declarations. The arity of a type is considered when matching partial declarations together: two classes with the same name but different numbers of type parameters are not considered to be partial declarations of the same time. Partial declarations can specify attributes, class modifiers, `Inherits` statement or `Implements` statement. At compile time, all of the pieces of the partial declarations are combined together and used as a part of the type declaration. If there are any conflicts between attributes, modifiers, bases, interfaces, or type members, a compile-time error results. For example:

```
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
```

The previous example declares a type `Test1` that is `Public`, inherits from `Object` and implements `System.IDisposable` and `System.IComparable`. The partial declarations of `Test2` will cause a compile-time error because one of the declarations says that `Test2` is `Public` and another says that `Test2` is `Private`.

Partial types with type parameters can declare constraints and variance for the type parameters, but the constraints and variance from each partial declaration must match. Thus, constraints and variance are special in that they are not automatically combined like other modifiers:

```
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
```

The fact that a type is declared using multiple partial declarations does not affect the name lookup rules within the type. As a result, a partial type declaration can use members declared in other partial type declarations, or may implement methods on interfaces declared in other partial type declarations. For example:

```
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
```

Nested types can have partial declarations as well. For example:

```
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
```

Initializers within a partial declaration will still be executed in declaration order; however, there is no guaranteed order of execution for initializers that occur in separate partial declarations.

7.12 Constructed Types

A generic type declaration, by itself, does not denote a type. Instead, a generic type declaration can be used as a "blueprint" to form many different types by applying type arguments. A generic type that has type arguments applied to it is called a *constructed type*. The type arguments in a constructed type must always satisfy the constraints placed on the type parameters they match to.

A type name might identify a constructed type even though it doesn't specify type parameters directly. This can occur where a type is nested within a generic class declaration, and the instance type of the containing declaration is implicitly used for name lookup:

```
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
```

A constructed type `C(Of T1,...,Tn)` is accessible when the generic type and all the type arguments are accessible. For instance, if the generic type `C` is `Public` and all of the type arguments `T1,...,Tn` are `Public`, then the constructed type is `Public`. If either the type name or one of the type arguments is `Private`, however, then the accessibility of the constructed type is `Private`. If one type argument of the constructed type is `Protected` and another type argument is `Friend`, then the constructed type is accessible only in the class and its subclasses in this assembly or any assembly that has been given `Friend` access. In other words, the accessibility domain for a constructed type is the intersection of the accessibility domains of its constituent parts.

Note. The fact that the accessibility domain of constructed type is the intersection of its constituted parts has the interesting side effect of defining a new accessibility level. A constructed type that contains an element that is `Protected` and an element that is `Friend` can only be accessed in contexts that can access *both* `Friend` *and* `Protected` members. However, there is no way to express this accessibility level in the language, as the accessibility `Protected Friend` means that an entity can be accessed in a context that can access *either* `Friend` *or* `Protected` members.

The base, implemented interfaces and members of constructed types are determined by substituting the supplied type arguments for each occurrence of the type parameter in the generic type.

7.12.1 Open Types and Closed Types

A constructed type for who one or more type arguments are type parameters of a containing type or method is called an *open type*. This is because some of the type parameters of the type are still not known, so the actual shape of the type is not yet fully known. In contrast, a generic type whose type arguments are all non-type parameters is called a *closed type*. The shape of a closed type is always fully known. For example:

```
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
```

The constructed type `Base(Of Integer, V)` is an open type because although the type parameter `T` has been supplied, the type parameter `U` has been supplied another type parameter. Thus, the full shape of the type is not yet known. The constructed type `Derived(Of Double)`, however, is a closed type because all type parameters in the inheritance hierarchy have been supplied.

Open types are defined as follows:

- A type parameter is an open type.
- An array type is an open type if its element type is an open type.
- A constructed type is an open type if one or more of its type arguments are an open type.
- A closed type is a type that is not an open type.

Because the program entry point cannot be in a generic type, all types used at run-time will be closed types.

7.13 Special Types

The .NET Framework contains a number of classes that are treated specially by the .NET Framework and by the Visual Basic language:

The type `System.Void`, which represents a void type in the .NET Framework, can be directly referenced only in `GetType` expressions.

The types `System.RuntimeArgumentHandle`, `System.ArgIterator` and `System.TypedReference` all can contain pointers into the stack and so cannot appear on the .NET Framework heap. Therefore, they cannot be used as array element types, return types, field types, generic type arguments, nullable types, `ByRef` parameter types, the type of a value being converted to `Object` or `System.ValueType`, the target of a call to instance members of `Object` or `System.ValueType`, or lifted into a closure.

8. Conversions

Conversion is the process of changing a value from one type to another. For example, a value of type `Integer` can be converted to a value of type `Double`, or a value of type `Derived` can be converted to a value of type `Base`, assuming that `Base` and `Derived` are both classes and `Derived` inherits from `Base`. Conversions may not require the value itself to change (as in the latter example), or they may require significant changes in the value representation (as in the former example).

Conversions may either be widening or narrowing. A *widening conversion* is a conversion from a type to another type whose value domain is at least as big, if not bigger, than the original type's value domain. Widening conversions should never fail. A *narrowing conversion* is a conversion from a type to another type whose value domain is either smaller than the original type's value domain or sufficiently unrelated that extra care must be taken when doing the conversion (for example, when converting from `Integer` to `String`). Narrowing conversions, which may entail loss of information, can fail.

The identity conversion (i.e. a conversion from a type to itself) and default value conversion (i.e. a conversion from `Nothing`) are defined for all types.

8.1 Implicit and Explicit Conversions

Conversions can be either *implicit* or *explicit*. Implicit conversions occur without any special syntax. The following is an example of implicit conversion of an `Integer` value to a `Long` value:

```
Module Test
    Sub Main()
        Dim intValue As Integer = 123
        Dim longValue As Long = intValue

        Console.WriteLine(intValue & " = " & longValue)
    End Sub
End Module
```

Explicit conversions, on the other hand, require cast operators. Attempting to do an explicit conversion on a value without a cast operator causes a compile-time error. The following example uses an explicit conversion to convert a `Long` value to an `Integer` value.

```
Module Test
    Sub Main()
        Dim longValue As Long = 134
        Dim intValue As Integer = CInt(longValue)

        Console.WriteLine(longValue & " = " & intValue)
    End Sub
End Module
```

The set of implicit conversions depends on the compilation environment and the `Option Strict` statement. If strict semantics are being used, only widening conversions may occur implicitly. If permissive semantics are being used, all widening and narrowing conversions (in other words, all conversions) may occur implicitly.

8.2 Boolean Conversions

Although `Boolean` is not a numeric type, it does have narrowing conversions to and from the numeric types as if it were an enumerated type. The literal `True` converts to the literal `255` for `Byte`, `65535` for `UShort`, `4294967295` for

`UInteger`, `18446744073709551615` for `ULong`, and to the expression `-1` for `SByte`, `Short`, `Integer`, `Long`, `Decimal`, `Single`, and `Double`. The literal `False` converts to the literal `0`. A zero numeric value converts to the literal `False`. All other numeric values convert to the literal `True`.

There is a narrowing conversion from Boolean to String, converting to either `System.Boolean.TrueString` or `System.Boolean.FalseString`. There is also a narrowing conversion from `String` to `Boolean`: if the string was equal to `TrueString` or `FalseString` (in the current culture, case-insensitively) then it uses the appropriate value; otherwise it attempts to parse the string as a numeric type (in hex or octal if possible, otherwise as a float) and uses the above rules; otherwise it throws `System.InvalidCastException`.

8.3 Numeric Conversions

Numeric conversions exist between the types `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single` and `Double`, and all enumerated types. When being converted, enumerated types are treated as if they were their underlying types. When converting to an enumerated type, the source value is not required to conform to the set of values defined in the enumerated type. For example:

```
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
```

Numeric conversions are processed at run-time as follows:

- For a conversion from a numeric type to a wider numeric type, the value is simply converted to the wider type. Conversions from `UInteger`, `Integer`, `ULong`, `Long`, or `Decimal` to `Single` or `Double` are rounded to the nearest `Single` or `Double` value. While this conversion may cause a loss of precision, it will never cause a loss of magnitude.
- For a conversion from an integral type to another integral type, or from `Single`, `Double`, or `Decimal` to an integral type, the result depends on whether integer overflow checking is on:

  If integer overflow is being checked:

  - If the source is an integral type, the conversion succeeds if the source argument is within the range of the destination type. The conversion throws a `System.OverflowException` exception if the source argument is outside the range of the destination type.
  - If the source is `Single`, `Double`, or `Decimal`, the source value is rounded up or down to the nearest integral value, and this integral value becomes the result of the conversion. If the source value is equally close to two integral values, the value is rounded to the value that has an even number in the least significant digit position. If the resulting integral value is outside the range of the destination type, a `System.OverflowException` exception is thrown.

    If integer overflow is not being checked:

  - If the source is an integral type, the conversion always succeeds and simply consists of discarding the most significant bits of the source value.

    - If the source is `Single`, `Double`, or `Decimal`, the conversion always succeeds and simply consists of rounding the source value towards the nearest integral value. If the source value is equally close to two integral values, the value is always rounded to the value that has an even number in the least significant digit position.
- For a conversion from `Double` to `Single`, the `Double` value is rounded to the nearest `Single` value. If the `Double` value is too small to represent as a `Single`, the result becomes positive zero or negative zero. If the `Double` value is too large to represent as a `Single`, the result becomes positive infinity or negative infinity. If the `Double` value is `NaN`, the result is also `NaN`.
- For a conversion from `Single` or `Double` to `Decimal`, the source value is converted to `Decimal` representation and rounded to the nearest number after the 28th decimal place if required. If the source value is too small to represent as a `Decimal`, the result becomes zero. If the source value is `NaN`, infinity, or too large to represent as a `Decimal`, a `System.OverflowException` exception is thrown.
- For a conversion from `Double` to `Single`, the `Double` value is rounded to the nearest `Single` value. If the `Double` value is too small to represent as a `Single`, the result becomes positive zero or negative zero. If the `Double` value is too large to represent as a `Single`, the result becomes positive infinity or negative infinity. If the `Double` value is `NaN`, the result is also `NaN`.

8.4 Reference Conversions

Reference types may be converted to a base type, and vice versa. Conversions from a base type to a more derived type only succeed at run time if the value being converted is a null value, the derived type itself, or a more derived type.

Class and interface types can be cast to and from any interface type. Conversions between a type and an interface type only succeed at run time if the actual types involved have an inheritance or implementation relationship. Because an interface type will always contain an instance of a type that derives from `Object`, an interface type can also always be cast to and from `Object`.

Note. It is not an error to convert a `NotInheritable` classes to and from interfaces that it does not implement because classes that represent COM classes may have interface implementations that are not known until run time.

If a reference conversion fails at run time, a `System.InvalidCastException` exception is thrown.

8.4.1 Reference Variance Conversions

Generic interfaces or delegates may have variant type parameters that allow conversions between compatible variants of the type. Therefore, at runtime a conversion from a class type or an interface type to an interface type that is variant compatible with an interface type it inherits from or implements will succeed. Similarly, delegate types can be cast to and from variant compatible delegate types. For example, the delegate type

```
Delegate Function F(Of In A, Out R)(a As A) As R
```

would allow a conversion from `F(Of Object, Integer)` to `F(Of String, Integer)`. That is, a delegate `F` which takes `Object` may be safely used as a delegate `F` which takes `String`. When the delegate is invoked, the target method will be expecting an object, and a string is an object.

A generic delegate or interface type `S(Of S1,...,Sn)` is said to be *variant compatible* with a generic interface or delegate type `T(Of T1,...,Tn)` if:

- `S` and `T` are both constructed from the same generic type `U(Of U1,...,Un)`.
- For each type parameter `Ux`:
    - If the type parameter was declared without variance then `Sx` and `Tx` must be the same type.

- If the type parameter was declared `In` then there must be a widening identity, default, reference, array, or type parameter conversion from `Sx` to `Tx`.
- If the type parameter was declared `Out` then there must be a widening identity, default, reference, array, or type parameter conversion from `Tx` to `Sx`.

When converting from a class to a generic interface with variant type parameters, if the class implements more than one variant compatible interface the conversion is ambiguous if there is not a non-variant conversion. For example:

```
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
```

8.4.2 Anonymous Delegate Conversions

When an expression classified as a lambda method is reclassified as a value in a context where there is no target type (for example, `Dim x = Function(a As Integer, b As Integer) a + b`), or where the target type is not a delegate type, the type of the resulting expression is an anonymous delegate type equivalent to the signature of the lambda method. This anonymous delegate type has a conversion to any compatible delegate type: a compatible delegate type is any delegate type that can be created using a delegate creation expression with the anonymous delegate type's `Invoke` method as a parameter. For example:

```
' 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
```

Note that the types `System.Delegate` and `System.MulticastDelegate` are not themselves considered delegate types (even though all delegate types inherit from them). Also, note that the conversion from anonymous delegate type to a compatible delegate type is not a reference conversion.

8.5 Array Conversions

Besides the conversions that are defined on arrays by virtue of the fact that they are reference types, several special conversions exist for arrays.

For any two types `A` and `B`, if they are both reference types or type parameters not known to be value types, and if `A` has a reference, array, or type parameter conversion to `B`, a conversion exists from an array of type `A` to an array of type `B` with the same rank. This relationship is known as *array covariance*. Array covariance in particular means that an element of an array whose element type is `B` may actually be an element of an array whose element type is `A`, provided that both `A` and `B` are reference types and that `B` has a reference conversion or array conversion to `A`. In the following example, the second invocation of `F` causes a `System.ArrayTypeMismatchException` exception to be thrown because the actual element type of `b` is `String`, not `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
```

Because of array covariance, assignments to elements of reference type arrays include a run-time check that ensures that the value being assigned to the array element is actually of a permitted type.

```
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
```

In this example, the assignment to `array(i)` in method `Fill` implicitly includes a run-time check that ensures that the object referenced by the variable `value` is either `Nothing` or an instance of a type that is compatible with the actual element type of array `array`. In method `Main`, the first two invocations of method `Fill` succeed, but the third invocation causes a `System.ArrayTypeMismatchException` exception to be thrown upon executing the first assignment to `array(i)`. The exception occurs because an `Integer` cannot be stored in a `String` array.

If one of the array element types is a type parameter whose type turns out to be a value type at runtime, a `System.InvalidCastException` exception will be thrown. For example:

```
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
```

Conversions also exist between an array of an enumerated type and an array of the enumerated type's underlying type or an array of another enumerated type with the same underlying type, provided the arrays have the same rank.

```
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
```

In this example, an array of `Color` is converted to and from an array of `Byte`, `Color`'s underlying type. The conversion to an array of `Integer`, however, will be an error because `Integer` is not the underlying type of `Color`.

A rank-1 array of type `A()` also has an array conversion to the collection interface types `IList(Of B)`, `IReadOnlyList(Of B)`, `ICollection(Of B)`, `IReadOnlyCollection(Of B)` and `IEnumerable(Of B)` found in `System.Collections.Generic`, so long as one of the following is true:

- `A` and `B` are both reference types or type parameters not known to be value types; and `A` has a widening reference, array or type parameter conversion to `B`; or
- `A` and `B` are both enumerated types of the same underlying type; or
- one of `A` and `B` is an enumerated type, and the other is its underlying type.

Any array of type A with any rank also has an array conversion to the non-generic collection interface types `IList`, `ICollection` and `IEnumerable` found in `System.Collections`.

It is possible to iterate over the resulting interfaces using `For Each`, or through invoking the `GetEnumerator` methods directly. In the case of rank-1 arrays converted generic or non-generic forms of `IList` or `ICollection`, it is also possible to get elements by index. In the case of rank-1 arrays converted to generic or non-generic forms of `IList`, it is also possible to set elements by index, subject to the same runtime array covariance checks as described above. The behavior of all other interface methods is undefined by the VB language specification; it is up to the underlying runtime.

8.6 Value Type Conversions

A value type value can be converted to one of its base reference types or an interface type that it implements through a process called *boxing*. When a value type value is boxed, the value is copied from the location where it lives onto the .NET Framework heap. A reference to this location on the heap is then returned and can be stored in a

reference type variable. This reference is also referred to as a *boxed* instance of the value type. The boxed instance has the same semantics as a reference type instead of a value type.

Boxed value types can be converted back to their original value type through a process called *unboxing*. When a boxed value type is unboxed, the value is copied from the heap into a variable location. From that point on, it behaves as if it was a value type. When unboxing a value type, the value must be a null value or an instance of the value type. Otherwise a `System.InvalidCastException` exception is thrown. If the value is an instance of an enumerated type, that value can also be unboxed to the enumerated type's underlying type or another enumerated type that has the same underlying type. A null value is treated as if it were the literal `Nothing`.

To support nullable value types well, the value type `System.Nullable(Of T)` is treated specially when doing boxing and unboxing. Boxing a value of type `Nullable(Of T)` results in a boxed value of type `T` if the value's `HasValue` property is `True` or a value of `Nothing` if the value's `HasValue` property is `False`. Unboxing a value of type `T` to `Nullable(Of T)` results in an instance of `Nullable(Of T)` whose `Value` property is the boxed value and whose `HasValue` property is `True`. The value `Nothing` can be unboxed to `Nullable(Of T)` for any `T` and results in a value whose `HasValue` property is `False`. Because boxed value types behave like reference types, it is possible to create multiple references to the same value. For the primitive types and enumerated types, this is irrelevant because instances of those types are *immutable*. That is, it is not possible to modify a boxed instance of those types, so it is not possible to observe the fact that there are multiple references to the same value.

Structures, on the other hand, may be mutable if its instance variables are accessible or if its methods or properties modify its instance variables. If one reference to a boxed structure is used to modify the structure, then all references to the boxed structure will see the change. Because this result may be unexpected, when a value typed as `Object` is copied from one location to another boxed value types will automatically be cloned on the heap instead of merely having their references copied. For example:

```
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
```

The output of the program is:

```
Values: 0, 123
Refs: 123, 123
```

The assignment to the field of the local variable `val2` does not impact the field of the local variable `val1` because when the boxed `Struct1` was assigned to `val2`, a copy of the value was made. In contrast, the assignment `ref2.Value = 123` affects the object that both `ref1` and `ref2` references.

Note. Structure copying is not done for boxed structures typed as `System.ValueType` because it is not possible to late bind off of `System.ValueType`.

There is one exception to the rule that boxed value types will be copied on assignment. If a boxed value type reference is stored within another type, the inner reference will not be copied. For example:

```
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
```

The output of the program is:

```
Values: 123, 123
```

This is because the inner boxed value is not copied when the value is copied. Thus, both `val1.Value` and `val2.Value` have a reference to the same boxed value type.

Note. The fact that inner boxed value types are not copied is a limitation of the .NET type system -- to ensure that all inner boxed value types were copied whenever a value of type `Object` was copied would be prohibitively expensive.

As previously described, boxed value types can only be unboxed to their original type. Boxed primitive types, however, are treated specially when typed as `Object`. They can be converted to any other primitive type that they have a conversion to. For example:

```
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
```

Normally, the boxed `Integer` value `5` could not be unboxed into a `Byte` variable. However, because `Integer` and `Byte` are primitive types and have a conversion, the conversion is allowed.

It is important to note that converting a value type to an interface is different than a generic argument constrained to an interface. When accessing interface members on a constrained type parameter (or calling methods on `Object`), boxing does not occur as it does when a value type is converted to an interface and an interface member is accessed. For example, suppose an interface `ICounter` contains a method `Increment` which can be used to modify a value. If `ICounter` is used as a constraint, the implementation of the `Increment` method is called with a reference to the variable that `Increment` was called on, not a boxed copy:

```
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
```

The first call to `Increment` modifies the value in the variable `x`. This is not equivalent to the second call to `Increment`, which modifies the value in a boxed copy of `x`. Thus, the output of the program is:

```
0
1
1
```

8.6.1 Nullable Value Type Conversions

A value type `T` can convert to and from the nullable version of the type, `T?`. The conversion from `T?` to `T` throws a `System.InvalidOperationException` exception if the value being converted is `Nothing`. Also, `T?` has a conversion to a type `S` if `T` has an intrinsic conversion to `S`. And if `S` is a value type, then the following intrinsic conversions exist between `T?` and `S?`:

- A conversion of the same classification (narrowing or widening) from `T?` to `S?`.
- A conversion of the same classification (narrowing or widening) from `T` to `S?`.
- A narrowing conversion from `S?` to `T`.

For example, an intrinsic widening conversion exists from `Integer?` to `Long?` because an intrinsic widening conversion exists from `Integer` to `Long`:

```
Dim i As Integer? = 10
Dim l As Long? = i
```

When converting from `T?` to `S?`, if the value of `T?` is `Nothing`, then the value of `S?` will be `Nothing`. When converting from `S?` to `T` or `T?` to `S`, if the value of `T?` or `S?` is `Nothing`, a `System.InvalidCastException` exception will be thrown.

Because of the behavior of the underlying type `System.Nullable(Of T)`, when a nullable value type `T?` is boxed, the result is a boxed value of type `T`, not a boxed value of type `T?`. And, conversely, when unboxing to a nullable value

type `T?`, the value will be wrapped by `System.Nullable(Of T)`, and `Nothing` will be unboxed to a null value of type `T?`. For example:

```
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
```

A side effect of this behavior is that a nullable value type `T?` appears to implement all of the interfaces of `T`, because converting a value type to an interface requires the type to be boxed. As a result, `T?` is convertible to all the interfaces that `T` is convertible to. It is important to note, however, that a nullable value type `T?` does not actually implement the interfaces of `T` for the purposes of generic constraint checking or reflection. For example:

```
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
```

8.7 String Conversions

Converting `Char` into `String` results in a string whose first character is the character value. Converting `String` into `Char` results in a character whose value is the first character of the string. Converting an array of `Char` into `String` results in a string whose characters are the elements of the array. Converting `String` into an array of `Char` results in an array of characters whose elements are the characters of the string.

The exact conversions between `String` and `Boolean`, `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single`, `Double`, `Date`, and vice versa, are beyond the scope of this specification and are implementation dependent with the exception of one detail. String conversions always consider the current culture of the run-time environment. As such, they must be performed at run time.

8.8 Widening Conversions

Widening conversions never overflow but may entail a loss of precision. The following conversions are widening conversions:

Identity/Default conversions

- From a type to itself.
- From an anonymous delegate type generated for a lambda method reclassification to any delegate type with an identical signature.

- From the literal `Nothing` to a type.

Numeric conversions

- From `Byte` to `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single`, or `Double`.
- From `SByte` to `Short`, `Integer`, `Long`, `Decimal`, `Single`, or `Double`.
- From `UShort` to `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single`, or `Double`.
- From `Short` to `Integer`, `Long`, `Decimal`, `Single` or `Double`.
- From `UInteger` to `ULong`, `Long`, `Decimal`, `Single`, or `Double`.
- From `Integer` to `Long`, `Decimal`, `Single` or `Double`.
- From `ULong` to `Decimal`, `Single`, or `Double`.
- From `Long` to `Decimal`, `Single` or `Double`.
- From `Decimal` to `Single` or `Double`.
- From `Single` to `Double`.
- From the literal `0` to an enumerated type. (**Note.** The conversion from `0` to any enumerated type is widening to simplify testing flags. For example, if `Values` is an enumerated type with a value `One`, you could test a variable `v` of type `Values` by saying `(v And Values.One) = 0`.)
- From an enumerated type to its underlying numeric type, or to a numeric type that its underlying numeric type has a widening conversion to.
- From a constant expression of type `ULong`, `Long`, `UInteger`, `Integer`, `UShort`, `Short`, `Byte`, or `SByte` to a narrower type, provided the value of the constant expression is within the range of the destination type. (**Note.** Conversions from `UInteger` or `Integer` to `Single`, `ULong` or `Long` to `Single` or `Double`, or `Decimal` to `Single` or `Double` may cause a loss of precision, but will never cause a loss of magnitude. The other widening numeric conversions never lose any information.)

Reference conversions

- From a reference type to a base type.
- From a reference type to an interface type, provided that the type implements the interface or a variant compatible interface.
- From an interface type to `Object`.
- From an interface type to a variant compatible interface type.
- From a delegate type to a variant compatible delegate type. (**Note.** Many other reference conversions are implied by these rules. For example, anonymous delegates are reference types that inherit from `System.MulticastDelegate`; array types are reference types that inherit from `System.Array`; anonymous types are reference types that inherit from `System.Object`.)

Anonymous Delegate conversions

- From an anonymous delegate type generated for a lambda method reclassification to any wider delegate type.

Array conversions

- From an array type `S` with an element type `Se` to an array type `T` with an element type `Te`, provided all of the following are true:
  - `S` and `T` differ only in element type.
  - Both `Se` and `Te` are reference types or are type parameters known to be a reference type.

    - A widening reference, array, or type parameter conversion exists from `Se` to `Te`.
- From an array type `S` with an enumerated element type `Se` to an array type `T` with an element type `Te`, provided all of the following are true:
    - `S` and `T` differ only in element type.
    - `Te` is the underlying type of `Se`.
- From an array type `S` of rank 1 with an enumerated element type `Se`, to `System.Collections.Generic.IList(Of Te)`, `IReadOnlyList(Of Te)`, `ICollection(Of Te)`, `IReadOnlyCollection(Of Te)`, and `IEnumerable(Of Te)`, provided one of the following is true:
    - Both `Se` and `Te` are reference types or are type parameters known to be a reference type, and a widening reference, array, or type parameter conversion exists from `Se` to `Te`; or
    - `Te` is the underlying type of `Se`; or
    - `Te` is identical to `Se`

Value Type conversions

- From a value type to a base type.
- From a value type to an interface type that the type implements.

Nullable Value Type conversions

- From a type `T` to the type `T?`.
- From a type `T?` to a type `S?`, where there is a widening conversion from the type `T` to the type `S`.
- From a type `T` to a type `S?`, where there is a widening conversion from the type `T` to the type `S`.
- From a type `T?` to an interface type that the type `T` implements.

String conversions

- From `Char` to `String`.
- From `Char()` to `String`.

Type Parameter conversions

- From a type parameter to `Object`.
- From a type parameter to an interface type constraint or any interface variant compatible with an interface type constraint.
- From a type parameter to an interface implemented by a class constraint.
- From a type parameter to an interface variant compatible with an interface implemented by a class constraint.
- From a type parameter to a class constraint, or a base type of the class constraint.
- From a type parameter `T` to a type parameter constraint `Tx`, or anything `Tx` has a widening conversion to.

8.9 Narrowing Conversions

Narrowing conversions are conversions that cannot be proved to always succeed, conversions that are known to possibly lose information, and conversions across domains of types sufficiently different to merit narrowing notation. The following conversions are classified as narrowing conversions:

Boolean conversions

- From `Boolean` to `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single`, or `Double`.
- From `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single`, or `Double` to `Boolean`.

Numeric conversions

- From `Byte` to `SByte`.
- From `SByte` to `Byte`, `UShort`, `UInteger`, or `ULong`.
- From `UShort` to `Byte`, `SByte`, or `Short`.
- From `Short` to `Byte`, `SByte`, `UShort`, `UInteger`, or `ULong`.
- From `UInteger` to `Byte`, `SByte`, `UShort`, `Short`, or `Integer`.
- From `Integer` to `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, or `ULong`.
- From `ULong` to `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, or `Long`.
- From `Long` to `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, or `ULong`.
- From `Decimal` to `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, or `Long`.
- From `Single` to `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, or `Decimal`.
- From `Double` to `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, or `Single`.
- From a numeric type to an enumerated type.
- From an enumerated type to a numeric type its underlying numeric type has a narrowing conversion to.
- From an enumerated type to another enumerated type.

Reference conversions

- From a reference type to a more derived type.
- From a class type to an interface type, provided the class type does not implement the interface type or an interface type variant compatible with it.
- From an interface type to a class type.
- From an interface type to another interface type, provided there is no inheritance relationship between the two types and provided they are not variant compatible.

Anonymous Delegate conversions

- From an anonymous delegate type generated for a lambda method reclassification to any narrower delegate type.

Array conversions

- From an array type `S` with an element type `Se`, to an array type `T` with an element type `Te`, provided that all of the following are true:
  - `S` and `T` differ only in element type.
  - Both `Se` and `Te` are reference types or are type parameters not known to be value types.
  - A narrowing reference, array, or type parameter conversion exists from `Se` to `Te`.
- From an array type `S` with an element type `Se` to an array type `T` with an enumerated element type `Te`, provided all of the following are true:
  - `S` and `T` differ only in element type.

    - `Se` is the underlying type of `Te` , or they are both different enumerated types that share the same underlying type.
- From an array type `S` of rank 1 with an enumerated element type `Se`, to `IList(Of Te)`, `IReadOnlyList(Of Te)`, `ICollection(Of Te)`, `IReadOnlyCollection(Of Te)` and `IEnumerable(Of Te)`, provided one of the following is true:
    - Both `Se` and `Te` are reference types or are type parameters known to be a reference type, and a narrowing reference, array, or type parameter conversion exists from `Se` to `Te`; or
    - `Se` is the underlying type of `Te`, or they are both different enumerated types that share the same underlying type.

Value type conversions

- From a reference type to a more derived value type.
- From an interface type to a value type, provided the value type implements the interface type.

Nullable Value Type conversions

- From a type `T?` to a type `T`.
- From a type `T?` to a type `S?`, where there is a narrowing conversion from the type `T` to the type `S`.
- From a type `T` to a type `S?`, where there is a narrowing conversion from the type `T` to the type `S`.
- From a type `S?` to a type `T`, where there is a conversion from the type `S` to the type `T`.

String conversions

- From `String` to `Char`.
- From `String` to `Char()`.
- From `String` to `Boolean` and from `Boolean` to `String`.
- Conversions between `String` and `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single`, or `Double`.
- From `String` to `Date` and from `Date` to `String`.

Type Parameter conversions

- From `Object` to a type parameter.
- From a type parameter to an interface type, provided the type parameter is not constrained to that interface or constrained to a class that implements that interface.
- From an interface type to a type parameter.
- From a type parameter to a derived type of a class constraint.
- From a type parameter `T` to anything a type parameter constraint `Tx` has a narrowing conversion to.

8.10 Type Parameter Conversions

Type parameters' conversions are determined by the constraints, if any, put on them. A type parameter `T` can always be converted to itself, to and from `Object`, and to and from any interface type. Note that if the type `T` is a value type at run-time, converting from `T` to `Object` or an interface type will be a boxing conversion and converting from `Object` or an interface type to `T` will be an unboxing conversion. A type parameter with a class constraint `C` defines additional conversions from the type parameter to `C` and its base classes, and vice versa. A type parameter `T` with a type parameter constraint `Tx` defines a conversion to `Tx` and anything `Tx` converts to.

An array whose element type is a type parameter with an interface constraint `I` has the same covariant array conversions as an array whose element type is `I`, provided that the type parameter also has a `Class` or class constraint (since only reference array element types can be covariant). An array whose element type is a type parameter with a class constraint `C` has the same covariant array conversions as an array whose element type is `C`.

The above conversions rules do not permit conversions from unconstrained type parameters to non-interface types, which may be surprising. The reason for this is to prevent confusion about the semantics of such conversions. For example, consider the following declaration:

```
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
```

If the conversion of `T` to `Integer` were permitted, one might easily expect that `X(Of Integer).F(7)` would return `7L`. However, it would not, because numeric conversions are only considered when the types are known to be numeric at compile time. In order to make the semantics clear, the above example must instead be written:

```
Class X(Of T)
    Public Shared Function F(t As T) As Long
        Return CLng(CObj(t))    ' OK, conversions permitted
    End Function
End Class
```

8.11 User-Defined Conversions

Intrinsic conversions are conversions defined by the language (i.e. listed in this specification), while *user-defined conversions* are defined by overloading the `CType` operator. When converting between types, if no intrinsic conversions are applicable then user-defined conversions will be considered. If there is a user-defined conversion that is *most specific* for the source and target types, then the user-defined conversion will be used. Otherwise, a compile-time error results. The most specific conversion is the one whose operand is "closest" to the source type and whose result type is "closest" to the target type. When determining what user-defined conversion to use, the most specific widening conversion will be used; if no widening conversion is most specific, the most specific narrowing conversion will be used. If there is no most specific narrowing conversion, then the conversion is undefined and a compile-time error occurs.

The following sections cover how the most specific conversions are determined. They use the following terms:

If an intrinsic widening conversion exists from a type `A` to a type `B`, and if neither `A` nor `B` are interfaces, then `A` is *encompassed* by `B`, and `B` *encompasses* `A`.

The *most encompassing* type in a set of types is the one type that encompasses all other types in the set. If no single type encompasses all other types, then the set has no most encompassing type. In intuitive terms, the most encompassing type is the "largest" type in the set -- the one type to which each of the other types can be converted through a widening conversion.

The *most encompassed* type in a set of types is the one type that is encompassed by all other types in the set. If no single type is encompassed by all other types, then the set has no most encompassed type. In intuitive terms, the most encompassed type is the "smallest" type in the set -- the one type that can be converted to each of the other types through a narrowing conversion.

When collecting the candidate user-defined conversions for a type `T?`, the user-defined conversion operators defined by `T` are used instead. If the type being converted to is also a nullable value type, then any of `T`'s user-defined conversions operators that involve only non-nullable value types are lifted. A conversion operator from `T` to `S` is lifted to be a conversion from `T?` to `S?` and is evaluated by converting `T?` to `T`, if necessary, then evaluating the user-defined conversion operator from `T` to `S` and then converting `S` to `S?`, if necessary. If the value being converted is `Nothing`, however, a lifted conversion operator converts directly into a value of `Nothing` typed as `S?`. For example:

```
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
```

When resolving conversions, user-defined conversions operators are always preferred over lifted conversion operators. For example:

```
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
```

At run-time, evaluating a user-defined conversion can involve up to three steps:

1. First, the value is converted from the source type to the operand type using an intrinsic conversion, if necessary.
2. Then, the user-defined conversion is invoked.
3. Finally, the result of the user-defined conversion is converted to the target type using an intrinsic conversion, if necessary.

It is important to note that evaluation of a user-defined conversion will never involve more than one user-defined conversion operator.

8.11.1 Most Specific Widening Conversion

Determining the most specific user-defined widening conversion operator between two types is accomplished using the following steps:

1. First, all of the candidate conversion operators are collected. The candidate conversion operators are all of the user-defined widening conversion operators in the source type and all of the user-defined widening conversion operators in the target type.
2. Then, all non-applicable conversion operators are removed from the set. A conversion operator is applicable to a source type and target type if there is an intrinsic widening conversion operator from the source type to the operand type and there is an intrinsic widening conversion operator from the result of the operator to the target type. If there are no applicable conversion operators, then there is no most specific widening conversion.
3. Then, the most specific source type of the applicable conversion operators is determined:
    - If any of the conversion operators convert directly from the source type, then the source type is the most specific source type.
    - Otherwise, the most specific source type is the most encompassed type in the combined set of source types of the conversion operators. If no most encompassed type can be found, then there is no most specific widening conversion.
4. Then, the most specific target type of the applicable conversion operators is determined:
    - If any of the conversion operators convert directly to the target type, then the target type is the most specific target type.
    - Otherwise, the most specific target type is the most encompassing type in the combined set of target types of the conversion operators. If no most encompassing type can be found, then there is no most specific widening conversion.
5. Then, if exactly one conversion operator converts from the most specific source type to the most specific target type, then this is the most specific conversion operator. If more than one such operator exists, then there is no most specific widening conversion.

8.11.2 Most Specific Narrowing Conversion

Determining the most specific user-defined narrowing conversion operator between two types is accomplished using the following steps:

1. First, all of the candidate conversion operators are collected. The candidate conversion operators are all of the user-defined conversion operators in the source type and all of the user-defined conversion operators in the target type.
2. Then, all non-applicable conversion operators are removed from the set. A conversion operator is applicable to a source type and target type if there is an intrinsic conversion operator from the source type to the operand type and there is an intrinsic conversion operator from the result of the operator to the target type. If there are no applicable conversion operators, then there is no most specific narrowing conversion.
3. Then, the most specific source type of the applicable conversion operators is determined:
    - If any of the conversion operators convert directly from the source type, then the source type is the most specific source type.
    - Otherwise, if any of the conversion operators convert from types that encompass the source type, then the most specific source type is the most encompassed type in the combined set of source types of those conversion operators. If no most encompassed type can be found, then there is no most specific narrowing conversion.

- Otherwise, the most specific source type is the most encompassing type in the combined set of source types of the conversion operators. If no most encompassing type can be found, then there is no most specific narrowing conversion.

4. Then, the most specific target type of the applicable conversion operators is determined:
   - If any of the conversion operators convert directly to the target type, then the target type is the most specific target type.
   - Otherwise, if any of the conversion operators convert to types that are encompassed by the target type, then the most specific target type is the most encompassing type in the combined set of source types of those conversion operators. If no most encompassing type can be found, then there is no most specific narrowing conversion.
   - Otherwise, the most specific target type is the most encompassed type in the combined set of target types of the conversion operators. If no most encompassed type can be found, then there is no most specific narrowing conversion.
5. Then, if exactly one conversion operator converts from the most specific source type to the most specific target type, then this is the most specific conversion operator. If more than one such operator exists, then there is no most specific narrowing conversion.

8.12 Native Conversions

Several of the conversions are classified as *native conversions* because they are supported natively by the .NET Framework. These conversions are ones that can be optimized through the use of the `DirectCast` and `TryCast` conversion operators, as well as other special behaviors. The conversions classified as native conversions are: identity conversions, default conversions, reference conversions, array conversions, value type conversions, and type parameter conversions.

8.13 Dominant Type

Given a set of types, it is often necessary in situations such as type inference to determine the *dominant type* of the set. The dominant type of a set of types is determined by first removing any types that one or more other types do not have an implicit conversion to. If there are no types left at this point, there is no dominant type. The dominant type is then the most encompassed of the remaining types. If there is more than one type that is most encompassed, then there is no dominant type.

9. Type Members

Type members define storage locations and executable code. They can be methods, constructors, events, constants, variables, and properties.

9.1 Interface Method Implementation

Methods, events, and properties can implement interface members. To implement an interface member, a member declaration specifies the `Implements` keyword and lists one or more interface members.

```
ImplementsClause:
    | ( 'Implements' ImplementsList  )?
    ;

ImplementsList:
    | InterfaceMemberSpecifier ( Comma InterfaceMemberSpecifier  )*
    ;

InterfaceMemberSpecifier:
    | NonArrayTypeName Period IdentifierOrKeyword
    ;
```

Methods and properties that implement interface members are implicitly `NotOverridable` unless declared to be `MustOverride`, `Overridable`, or overriding another member. It is an error for a member implementing an interface member to be `Shared`. A member's accessibility has no effect on its ability to implement interface members.

For an interface implementation to be valid, the implements list of the containing type must name an interface that contains a compatible member. A compatible member is one whose signature matches the signature of the implementing member. If a generic interface is being implemented, then the type argument supplied in the Implements clause is substituted into the signature when checking compatibility. For example:

```
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
```

If an event declared using a delegate type is implementing an interface event, then a compatible event is one whose underlying delegate type is the same type. Otherwise, the event uses the delegate type from the interface event it is implementing. If such an event implements multiple interface events, all the interface events must have the same underlying delegate type. For example:

```
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
```

An interface member in the implements list is specified using a type name, a period, and an identifier. The type name must be an interface in the implements list or a base interface of an interface in the implements list, and the identifier must be a member of the specified interface. A single member can implement more than one matching interface member.

```
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
```

If the interface member being implemented is unavailable in all explicitly implemented interfaces because of multiple interface inheritance, the implementing member must explicitly reference a base interface on which the member is available. For example, if `I1` and `I2` contain a member `M`, and `I3` inherits from `I1` and `I2`, a type implementing `I3` will implement `I1.M` and `I2.M`. If an interface shadows multiply inherited members, an implementing type will have to implement the inherited members and the member(s) shadowing them.

```
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
```

If the containing interface of the interface member be implemented is generic, the same type arguments as the interface being implements must be supplied. For example:

```
    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
```

9.2 Methods

Methods contain the executable statements of a program.

```
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
    ;
```

Methods, which have an optional list of parameters and an optional return value, are either shared or not shared. Shared methods are accessed through the class or instances of the class. Non-shared methods, also called instance methods, are accessed through instances of the class. The following example shows a class `Stack` that has several shared methods (`Clone` and `Flip`), and several instance methods (`Push`, `Pop`, and `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
```

Methods can be overloaded, which means that multiple methods may have the same name so long as they have unique signatures. The signature of a method consists of the number and types of its parameters. The signature of a method specifically does not include the return type or parameter modifiers such as Optional, ByRef or ParamArray. The following example shows a class with a number of overloads:

```
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
```

The output of the program is:

```
F()
F(Integer)
F(Object)
F(Integer, Integer)
F(Integer())
G(String)
G(String, Optional String)
```

Overloads that differ only in optional parameters can be used for "versioning" of libraries. For instance, v1 of a library might include a function with optional parameters:

```
Sub fopen(fileName As String, Optional accessMode as Integer = 0)
```

Then v2 of the library wants to add another optional parameter "password", and it wants to do so without breaking source compatibility (so applications that used to target v1 can be recompiled), and without breaking binary compatibility (so applications that used to reference v1 can now reference v2 without recompilation). This is how v2 will look:

```
Sub fopen(file As String, mode as Integer)
Sub fopen(file As String, Optional mode as Integer = 0, Optional pword As String = "")
```

Note that optional parameters in a public API are not CLS-compliant. However, they can be consumed at least by Visual Basic and C#4 and F#.

9.2.1 Regular, Async and Iterator Method Declarations

There are two types of methods: *subroutines*, which do not return values, and *functions*, which do. The body and `End` construct of a method may only be omitted if the method is defined in an interface or has the `MustOverride` modifier. If no return type is specified on a function and strict semantics are being used, a compile-time error occurs; otherwise the type is implicitly `Object` or the type of the method's type character. The accessibility domain of the return type and parameter types of a method must be the same as or a superset of the accessibility domain of the method itself.

A **regular method** is one with neither `Async` nor `Iterator` modifiers. It may be a subroutine or a function. Section §10.1.1 details what happens when a regular method is invoked.

An **iterator method** is one with the `Iterator` modifier and no `Async` modifier. It must be a function, and its return type must be `IEnumerator`, `IEnumerable`, or `IEnumerator(Of T)` or `IEnumerable(Of T)` for some `T`, and it must have no `ByRef` parameters. Section §10.1.2 details what happens when an iterator method is invoked.

An **async method** is one with the `Async` modifier and no `Iterator` modifier. It must be either a subroutine, or a function with return type `Task` or `Task(Of T)` for some `T`, and must have no `ByRef` parameters. Section §10.1.3 details what happens when an async method is invoked.

It is a compile-time error if a method is not one of these three kinds of method.

Subroutine and function declarations are special in that their beginning and end statements must each start at the beginning of a logical line. Additionally, the body of a non-`MustOverride` subroutine or function declaration must start at the beginning of a logical line. For example:

```
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
```

9.2.2 External Method Declarations

An external method declaration introduces a new method whose implementation is provided external to the program.

```
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
    ;
```

Because an external method declaration provides no actual implementation, it has no method body or `End` construct. External methods are implicitly shared, may not have type parameters, and may not handle events or implement interface members. If no return type is specified on a function and strict semantics are being used, a compile-time error occurs. Otherwise the type is implicitly `Object` or the type of the method's type character. The accessibility domain of the return type and parameter types of an external method must be the same as or a superset of the accessibility domain of the external method itself.

The library clause of an external method declaration specifies the name of the external file that implements the method. The optional alias clause is a string that specifies the numeric ordinal (prefixed by a `#` character) or name of the method in the external file. A single-character set modifier may also be specified, which governs the character set used to marshal strings during a call to the external method. The `Unicode` modifier marshals all strings to Unicode values, the `Ansi` modifier marshals all strings to ANSI values, and the `Auto` modifier marshals the strings according to .NET Framework rules based on the name of the method, or the alias name if specified. If no modifier is specified, the default is `Ansi`.

If `Ansi` or `Unicode` is specified, then the method name is looked up in the external file with no modification. If `Auto` is specified, then method name lookup depends on the platform. If the platform is considered to be ANSI (for example, Windows 95, Windows 98, Windows ME), then the method name is looked up with no modification. If the lookup fails, an `A` is appended and the lookup tried again. If the platform is considered to be Unicode (for example, Windows NT, Windows 2000, Windows XP), then a `W` is appended and the name is looked up. If the lookup fails, the lookup is tried again without the `W`. For example:

```
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
```

Data types being passed to external methods are marshaled according to the .NET Framework data marshalling conventions with one exception. String variables that are passed by value (that is, `ByVal x As String`) are marshaled to the OLE Automation BSTR type, and changes made to the BSTR in the external method are reflected back in the string argument. This is because the type `String` in external methods is mutable, and this special marshalling mimics that behavior. String parameters that are passed by reference (i.e. `ByRef x As String`) are marshaled as a pointer to the OLE Automation BSTR type. It is possible to override these special behaviors by specifying the `System.Runtime.InteropServices.MarshalAsAttribute` attribute on the parameter.

The example demonstrates use of external methods:

```
    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
```

9.2.3 Overridable Methods

The `Overridable` modifier indicates that a method is overridable. The `Overrides` modifier indicates that a method overrides a base-type overridable method that has the same signature. The `NotOverridable` modifier indicates that an overridable method cannot be further overridden. The `MustOverride` modifier indicates that a method must be overridden in derived classes.

Certain combinations of these modifiers are not valid:

- `Overridable` and `NotOverridable` are mutually exclusive and cannot be combined.
- `MustOverride` implies `Overridable` (and so cannot specify it) and cannot be combined with `NotOverridable`.
- `NotOverridable` cannot be combined with `Overridable` or `MustOverride` and must be combined with `Overrides`.
- `Overrides` implies `Overridable` (and so cannot specify it) and cannot be combined with `MustOverride`.

There are also additional restrictions on overridable methods:

- A `MustOverride` method may not include a method body or an `End` construct, may not override another method, and may only appear in `MustInherit` classes.
- If a method specifies `Overrides` and there is no matching base method to override, a compile-time error occurs. An overriding method may not specify `Shadows`.
- A method may not override another method if the overriding method's accessibility domain is not equal to the accessibility domain of the method being overridden. The one exception is that a method overriding a `Protected Friend` method in another assembly that does not have `Friend` access must specify `Protected` (not `Protected Friend`).
- `Private` methods may not be `Overridable`, `NotOverridable`, or `MustOverride`, nor may they override other methods.
- Methods in `NotInheritable` classes may not be declared `Overridable` or `MustOverride`.

The following example illustrates the differences between overridable and nonoverridable methods:

```
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
```

In the example, class `Base` introduces a method `F` and an `Overridable` method `G`. The class `Derived` introduces a new method `F`, thus shadowing the inherited `F`, and also overrides the inherited method `G`. The example produces the following output:

```
Base.F
Derived.F
Derived.G
Derived.G
```

Notice that the statement `b.G()` invokes `Derived.G`, not `Base.G`. This is because the run-time type of the instance (which is `Derived`) rather than the compile-time type of the instance (which is `Base`) determines the actual method implementation to invoke.

9.2.4 Shared Methods

The `Shared` modifier indicates a method is a *shared method*. A shared method does not operate on a specific instance of a type and may be invoked directly from a type rather than through a particular instance of a type. It is valid, however, to use an instance to qualify a shared method. It is invalid to refer to `Me`, `MyClass`, or `MyBase` in a shared method. Shared methods may not be `Overridable`, `NotOverridable`, or `MustOverride`, and they may not override methods. Methods defined in standard modules and interfaces may not specify `Shared`, because they are implicitly `Shared` already.

A method declared in a structure or class without a `Shared` modifier is an *instance method*. An instance method operates on a given instance of a type. Instance methods can only be invoked through an instance of a type and may refer to the instance through the `Me` expression.

The following example illustrates the rules for accessing shared and instance members:

```
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
```

Method `F` shows that in an instance function member, an identifier can be used to access both instance members and shared members. Method `G` shows that in a shared function member, it is an error to access an instance member through an identifier. Method `Main` shows that in a member access expression, instance members must be accessed through instances, but shared members can be accessed through types or instances.

9.2.5 Method Parameters

A *parameter* is a variable that can be used to pass information into and out of a method. Parameters of a method are declared by the method's parameter list, which consists of one or more parameters separated by commas.

```
ParameterList:
    | Parameter ( Comma Parameter  )*
    ;

Parameter:
    | Attributes? ParameterModifier* ParameterIdentifier ( 'As' TypeName  )?
      ( Equals ConstantExpression  )?
    ;

ParameterModifier:
    | 'ByVal' | 'ByRef' | 'Optional' | 'ParamArray'
    ;

ParameterIdentifier:
    | Identifier IdentifierModifiers
    ;
```

If no type is specified for a parameter and strict semantics are used, a compile-time error occurs. Otherwise the default type is `Object` or the type of the parameter's type character. Even under permissive semantics, if one parameter includes an `As` clause, all parameters must specify types.

Parameters are specified as value, reference, optional, or paramarray parameters by the modifiers `ByVal`, `ByRef`, `Optional`, and `ParamArray`, respectively. A parameter that does not specify `ByRef` or `ByVal` defaults to `ByVal`.

Parameter names are scoped to the entire body of the method and are always publicly accessible. A method invocation creates a copy, specific to that invocation, of the parameters, and the argument list of the invocation assigns values or variable references to the newly created parameters. Because external method declarations and delegate declarations have no body, duplicate parameter names are allowed in parameter lists, but discouraged.

The identifier may be followed by the nullable name modifier `?` to indicate that it is nullable, and also by array name modifiers to indicate that it is an array. They may be combined, e.g. "`ByVal x?() As Integer`". It is not allowed to use explicit array bounds; also, if the nullable name modifier is present then an `As` clause must be present.

9.2.5.1 Value Parameters

A *value parameter* is declared with an explicit `ByVal` modifier. If the `ByVal` modifier is used, the `ByRef` modifier may not be specified. A value parameter comes into existence with the invocation of the member the parameter belongs to, and is initialized with the value of the argument given in the invocation. A value parameter ceases to exist upon return of the member.

A method is permitted to assign new values to a value parameter. Such assignments only affect the local storage location represented by the value parameter; they have no effect on the actual argument given in the method invocation.

A value parameter is used when the value of an argument is passed into a method, and modifications of the parameter do not impact the original argument. A value parameter refers to its own variable, one that is distinct from the variable of the corresponding argument. This variable is initialized by copying the value of the corresponding argument. The following example shows a method `F` that has a value parameter named `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
```

The example produces the following output, even though the value parameter `p` is modified:

```
pre: a = 1
p = 1
post: a = 1
```

9.2.5.2 Reference Parameters

A reference parameter is a parameter declared with a `ByRef` modifier. If the `ByRef` modifier is specified, the `ByVal` modifier may not be used. A reference parameter does not create a new storage location. Instead, a reference parameter represents the variable given as the argument in the method or constructor invocation. Conceptually, the value of a reference parameter is always the same as the underlying variable.

Reference parameters act in two modes, either as *aliases* or through *copy-in copy-back.*

Aliases. A reference parameter is used when the parameter acts as an alias for a caller-provided argument. A reference parameter does not itself define a variable, but rather refers to the variable of the corresponding

argument. Modifications of a reference parameter directly and immediately impact the corresponding argument. The following example shows a method `Swap` that has two reference parameters:

```
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
```

The output of the program is:

```
pre: x = 1, y = 2
post: x = 2, y = 1
```

For the invocation of method `Swap` in class `Main`, `a` represents `x`, and `b` represents `y`. Thus, the invocation has the effect of swapping the values of `x` and `y`.

In a method that takes reference parameters, it is possible for multiple names to represent the same storage location:

```
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
```

In the example the invocation of method `F` in `G` passes a reference to `s` for both `a` and `b`. Thus, for that invocation, the names `s`, `a`, and `b` all refer to the same storage location, and the three assignments all modify the instance variable `s`.

Copy-in copy-back. If the type of the variable being passed to a reference parameter is not compatible with the reference parameter's type, or if a non-variable (e.g. a property) is passed as an argument to a reference parameter, or if the invocation is late-bound, then a temporary variable is allocated and passed to the reference parameter. The value being passed in will be copied into this temporary variable before the method is invoked and will be copied back to the original variable (if there is one and if it's writable) when the method returns. Thus, a reference parameter may not necessarily contain a reference to the exact storage of the variable being passed in, and any changes to the reference parameter may not be reflected in the variable until the method exits. For example:

```
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
```

In the case of the first invocation of `F`, a temporary variable is created and the value of the property `G` is assigned to it and passed into `F`. Upon return from `F`, the value in the temporary variable is assigned back to the property of `G`. In the second case, another temporary variable is created and the value of `d` is assigned to it and passed into `F`. When returning from `F`, the value in the temporary variable is cast back to the type of the variable, `Derived`, and assigned to `d`. Since the value being passed back cannot be cast to `Derived`, an exception is thrown at run time.

9.2.5.3 Optional Parameters

An optional parameter is declared with the `Optional` modifier. Parameters that follow an optional parameter in the formal parameter list must be optional as well; failure to specify the `Optional` modifier on the following parameters will trigger a compile-time error. An optional parameter of some type nullable type `T?` or non-nullable type `T` must specify a constant expression `e` to be used as a default value if no argument is specified. If `e` evaluates to `Nothing` of type Object, then the default value of the *parameter type* will be used as the default for the parameter. Otherwise, `CType(e, T)` must be a constant expression and it is taken as the default for the parameter.

Optional parameters are the only situation in which an initializer on a parameter is valid. The initialization is always done as a part of the invocation expression, not within the method body itself.

```
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
```

The output of the program is:

```
x = 10, y = 20
x = 30, y = 40
```

Optional parameters may not be specified in delegate or event declarations, nor in lambda expressions.

9.2.5.4 ParamArray Parameters

ParamArray parameters are declared with the ParamArray modifier. If the ParamArray modifier is present, the ByVal modifier must be specified, and no other parameter may use the ParamArray modifier. The ParamArray parameter's type must be a one-dimensional array, and it must be the last parameter in the parameter list.

A ParamArray parameter represents an indeterminate number of parameters of the type of the ParamArray. Within the method itself, a ParamArray parameter is treated as its declared type and has no special semantics. A ParamArray parameter is implicitly optional, with a default value of an empty one-dimensional array of the type of the ParamArray.

A ParamArray permits arguments to be specified in one of two ways in a method invocation:

- The argument given for a ParamArray can be a single expression of a type that widens to the ParamArray type. In this case, the ParamArray acts precisely like a value parameter.
- Alternatively, the invocation can specify zero or more arguments for the ParamArray, where each argument is an expression of a type that is implicitly convertible to the element type of the ParamArray. In this case, the invocation creates an instance of the ParamArray type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

Except for allowing a variable number of arguments in an invocation, a ParamArray is precisely equivalent to a value parameter of the same type, as the following example illustrates.

```
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
```

The example produces the output

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

The first invocation of F simply passes the array a as a value parameter. The second invocation of F automatically creates a four-element array with the given element values and passes that array instance as a value parameter. Likewise, the third invocation of F creates a zero-element array and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing:

```
F(New Integer() {10, 20, 30, 40})
F(New Integer() {})
```

ParamArray parameters may not be specified in delegate or event declarations.

9.2.6 Event Handling

Methods can declaratively handle events raised by objects in instance or shared variables. To handle events, a method declaration specifies the `Handles` keyword and lists one or more events.

```
HandlesClause:
    | ( 'Handles' EventHandlesList  )?
    ;

EventHandlesList:
    | EventMemberSpecifier ( Comma EventMemberSpecifier  )*
    ;

EventMemberSpecifier:
    | Identifier Period IdentifierOrKeyword
    | 'MyBase' Period IdentifierOrKeyword
    | 'MyClass' Period IdentifierOrKeyword
    | 'Me' Period IdentifierOrKeyword
    ;
```

An event in the `Handles` list is specified by two identifiers separated by a period:

- The first identifier must be an instance or shared variable in the containing type that specifies the `WithEvents` modifier or the `MyBase` or `MyClass` or `Me` keyword; otherwise, a compile-time error occurs. This variable contains the object that will raise the events handled by this method.
- The second identifier must specify a member of the type of the first identifier. The member must be an event, and may be shared. If a shared variable is specified for the first identifier, then the event must be shared, or an error results.

A handler method `M` is considered a valid event handler for an event `E` if the statement `AddHandler E, AddressOf M` would also be valid. Unlike an `AddHandler` statement, however, explicit event handlers allow handling an event with a method with no arguments regardless of whether strict semantics are being used or not:

```
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
```

A single member can handle multiple matching events, and multiple methods may handle a single event. A method's accessibility has no effect on its ability to handle events. The following example shows how a method can handle events:

```
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
```

This will print out:

```
Raised
Raised
```

A type inherits all event handlers provided by its base type. A derived type cannot in any way alter the event mappings it inherits from its base types, but may add additional handlers to the event.

9.2.7 Extension Methods

Methods can be added to types from outside of the type declaration using *extension methods*. Extension methods are methods with the `System.Runtime.CompilerServices.ExtensionAttribute` attribute applied to them. They can only be declared in standard modules and must have at least one parameter, which specifies the type the method extends. For example, the following extension method extends the type `String`:

```
Imports System.Runtime.CompilerServices

Module StringExtensions
    <Extension> _
    Sub Print(s As String)
        Console.WriteLine(s)
    End Sub
End Module
```

Note. Although Visual Basic requires extension methods to be declared in a standard module, other languages such as C# may allow them to be declared in other kinds of types. As long as the methods follow the other conventions outlined here and the containing type is not an open generic type and cannot be instantiated, Visual Basic will recognize the extension methods.

When an extension method is invoked, the instance it is being invoked on is passed to the first parameter. The first parameter cannot be declared `Optional` or `ParamArray`. Any type, including a type parameter, can appear as the first parameter of an extension method. For example, the following methods extend the types `Integer()`, any type that implements `System.Collections.Generic.IEnumerable(Of T)`, and any type at all:

```
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
```

As the previous example shows, interfaces can be extended. Interface extension methods supply the implementation of the method, so types that implement an interface that has extension methods defined on it still only implement the members originally declared by the interface. For example:

```
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
```

Extension methods can also have type constraints on their type parameters and, just as with non-extension generic methods, type argument can be inferred:

```
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
```

Extension methods can also be accessed through implicit instance expressions within the type being extended:

```
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
```

For the purposes of accessibility, extension methods are also treated as members of the standard module they are declared in -- they have no extra access to the members of the type they are extending beyond the access they have by virtue of their declaration context.

Extensions methods are only available when the standard module method is in scope. Otherwise, the original type will not appear to have been extended. For example:

```
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
```

Referring to a type when only an extension method on the type is available will still produce a compile-time error.

It is important to note that extension methods are considered to be members of the type in all contexts where members are bound, such as the strongly-typed `For Each` pattern. For example:

```
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
```

Delegates can also be created that refer to extension methods. Thus, the code:

```
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
```

is roughly equivalent to:

```
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
```

Note. Visual Basic normally inserts a check on an instance method call that causes a `System.NullReferenceException` to occur if the instance the method is being invoked on is `Nothing`. In the case of extension methods, there is no efficient way to insert this check, so extension methods will need to explicitly check for `Nothing`.

Note. A value type will be boxed when being passed as a `ByVal` argument to a parameter typed as an interface. This implies that side effects of the extension method will operate on a copy of the structure instead of the original. While the language puts no restrictions on the first argument of an extension method, it is recommended that extension methods are not used to extend value types or that when extending value types, the first parameter is passed `ByRef` to ensure that side effects operate on the original value.

9.2.8 Partial Methods

A *partial method* is a method that specifies a signature but not the body of the method. The body of the method can be supplied by another method declaration with the same name and signature, most likely in another partial declaration of the type. For example:

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
```

In this example, a partial declaration of the class `MyForm` declares a partial method `ValidateControls` with no implementation. The constructor in the partial declaration calls the partial method, even though there is no body supplied in the file. The other partial declaration of `MyForm` then supplies the implementation of the method.

Partial methods can be called regardless of whether a body has been supplied; if no method body is supplied, the call is ignored. For example:

```
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
```

Any expressions that are passed in as arguments to a partial method call that is ignored are ignored also and not evaluated. (**Note.** This means that partial methods are a very efficient way of providing behavior that is defined across two partial types, since the partial methods have no cost if they are not used.)

The partial method declaration must be declared as `Private` and must always be a subroutine with no statements in its body. Partial methods cannot themselves implement interface methods, although the method that supplies their body can.

Only one method can supply a body to a partial method. A method supplying a body to a partial method must have the same signature as the partial method, the same constraints on any type parameters, the same declaration modifiers, and the same parameter and type parameter names. Attributes on the partial method and the method that supplies its body are merged, as are any attributes on the methods' parameters. Similarly, the list of events that the methods handle is merged. For example:

```
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
```

9.3 Constructors

Constructors are special methods that allow control over initialization. They are run after the program begins or when an instance of a type is created. Unlike other members, constructors are not inherited and do not introduce a name into a type's declaration space. Constructors may only be invoked by object-creation expressions or by the .NET Framework; they may never be directly invoked.

Note. Constructors have the same restriction on line placement that subroutines have. The beginning statement, end statement and block must all appear at the beginning of a logical line.

```
ConstructorMemberDeclaration:
    | Attributes? ConstructorModifier* 'Sub' 'New'
      ( OpenParenthesis ParameterList? CloseParenthesis  )? LineTerminator
      Block?
      'End' 'Sub' StatementTerminator
    ;

ConstructorModifier:
    | AccessModifier
    | 'Shared'
    ;
```

9.3.1 Instance Constructors

Instance constructors initialize instances of a type and are run by the .NET Framework when an instance is created. The parameter list of a constructor is subject to the same rules as the parameter list of a method. Instance constructors may be overloaded.

All constructors in reference types must invoke another constructor. If the invocation is explicit, it must be the first statement in the constructor method body. The statement can either invoke another of the type's instance constructors -- for example, `Me.New(...)` or `MyClass.New(...)` -- or if it is not a structure it can invoke an instance constructor of the type's base type -- for example, `MyBase.New(...)`. It is invalid for a constructor to invoke itself. If a constructor omits a call to another constructor, `MyBase.New()` is implicit. If there is no parameterless base type constructor, a compile-time error occurs. Because `Me` is not considered to be constructed until after the call to a base class constructor, the parameters to a constructor invocation statement cannot reference `Me`, `MyClass`, or `MyBase` implicitly or explicitly.

When a constructor's first statement is of the form `MyBase.New(...)`, the constructor implicitly performs the initializations specified by the variable initializers of the instance variables declared in the type. This corresponds to a sequence of assignments that are executed immediately after invoking the direct base type constructor. Such ordering ensures that all base instance variables are initialized by their variable initializers before any statements that have access to the instance are executed. For example:

```
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
```

When `New B()` is used to create an instance of `B`, the following output is produced:

```
x = 1, y = 1
```

The value of `y` is `1` because the variable initializer is executed after the base class constructor is invoked. Variable initializers are executed in the textual order they appear in the type declaration.

When a type declares only `Private` constructors, it is not possible in general for other types to derive from the type or create instances of the type; the only exception is types nested within the type. `Private` constructors are commonly used in types that contain only `Shared` members.

If a type contains no instance constructor declarations, a default constructor is automatically provided. The default constructor simply invokes the parameterless constructor of the direct base type. If the direct base type does not have an accessible parameterless constructor, a compile-time error occurs. The declared access type for the default constructor is `Public` unless the type is `MustInherit`, in which case the default constructor is `Protected`.

Note. The default access for a `MustInherit` type's default constructor is `Protected` because `MustInherit` classes cannot be created directly. So there is no point in making the default constructor `Public`.

In the following example a default constructor is provided because the class contains no constructor declarations:

```
Class Message
    Dim sender As Object
    Dim text As String
End Class
```

Thus, the example is precisely equivalent to the following:

```
Class Message
    Dim sender As Object
    Dim text As String

    Sub New()
    End Sub
End Class
```

Default constructors that are emitted into a designer generated class marked with the attribute `Microsoft.VisualBasic.CompilerServices.DesignerGeneratedAttribute` will call the method `Sub InitializeComponent()`, if it exists, after the call to the base constructor. (**Note.** This allows designer generated files, such as those created by the WinForms designer, to omit the constructor in the designer file. This enables the programmer to specify it themselves, if they so choose.)

9.3.2 Shared Constructors

Shared constructors initialize a type's shared variables; they are run after the program begins executing, but before any references to a member of the type. A shared constructor specifies the `Shared` modifier, unless it is in a standard module in which case the `Shared` modifier is implied.

Unlike instance constructors, shared constructors have implicit public access, have no parameters, and may not call other constructors. Before the first statement in a shared constructor, the shared constructor implicitly performs the

initializations specified by the variable initializers of the shared variables declared in the type. This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor. The variable initializers are executed in the textual order they appear in the type declaration.

The following example shows an `Employee` class with a shared constructor that initializes a shared variable:

```
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
```

A separate shared constructor exists for each closed generic type. Because the shared constructor is executed exactly once for each closed type, it is a convenient place to enforce run-time checks on the type parameter that cannot be checked at compile-time via constraints. For example, the following type uses a shared constructor to enforce that the type parameter is `Integer` or `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
```

Exactly when shared constructors are run is mostly implementation dependent, though several guarantees are provided if a shared constructor is explicitly defined:

- Shared constructors are run before the first access to any static field of the type.
- Shared constructors are run before the first invocation of any static method of the type.
- Shared constructors are run before the first invocation of any constructor for the type.

The above guarantees do not apply in the situation where a shared constructor is implicitly created for shared initializers. The output from the following example is uncertain, because the exact ordering of loading and therefore of shared constructor execution is not defined:

```
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
```

The output could be either of the following:

```
Init A
A.F
Init B
B.F
```

or

```
Init B
Init A
A.F
B.F
```

By contrast, the following example produces predictable output. Note that the `Shared` constructor for the class `A` never executes, even though class `B` derives from it:

```
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
```

The output is:

```
Init B
B.G
```

It is also possible to construct circular dependencies that allow `Shared` variables with variable initializers to be observed in their default value state, as in the following example:

```
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
```

This produces the output:

```
X = 1, Y = 2
```

To execute the `Main` method, the system first loads class `B`. The `Shared` constructor of class `B` proceeds to compute the initial value of `Y`, which recursively causes class `A` to be loaded because the value of `A.X` is referenced. The `Shared` constructor of class `A` in turn proceeds to compute the initial value of `X`, and in doing so fetches the *default* value of `Y`, which is zero. `A.X` is thus initialized to `1`. The process of loading `A` then completes, returning to the calculation of the initial value of `Y`, the result of which becomes `2`.

Had the `Main` method instead been located in class `A`, the example would have produced the following output:

```
X = 2, Y = 1
```

Avoid circular references in `Shared` variable initializers since it is generally impossible to determine the order in which classes containing such references are loaded.

9.4 Events

Events are used to notify code of a particular occurrence. An event declaration consists of an identifier, either a delegate type or a parameter list, and an optional `Implements` clause.

```
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'
    ;
```

If a delegate type is specified, the delegate type may not have a return type. If a parameter list is specified, it may not contain `Optional` or `ParamArray` parameters. The accessibility domain of the parameter types and/or delegate type must be the same as, or a superset of, the accessibility domain of the event itself. Events may be shared by specifying the `Shared` modifier.

In addition to the member name added to the type's declaration space, an event declaration implicitly declares several other members. Given an event named `X`, the following members are added to the declaration space:

- If the form of the declaration is a method declaration, a nested delegate class named `XEventHandler` is introduced. The nested delegate class matches the method declaration and has the same accessibility as the event. The attributes in the parameter list apply to the parameters of the delegate class.
- A `Private` instance variable typed as the delegate, named `XEvent`.
- Two methods named `add_X` and `remove_X` which cannot be invoked, overridden or overloaded.

If a type attempts to declare a name that matches one of the above names, a compile-time error will result, and the implicit `add_X` and `remove_X` declarations are ignored for the purposes of name binding. It is not possible to override or overload any of the introduced members, although it is possible to shadow them in derived types. For example, the class declaration

```
Class Raiser
    Public Event Constructed(i As Integer)
End Class
```

is equivalent to the following declaration

```
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
```

Declaring an event without specifying a delegate type is the simplest and most compact syntax, but has the disadvantage of declaring a new delegate type for each event. For example, in the following example, three hidden delegate types are created, even though all three events have the same parameter list:

```
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
```

In the following example, the events simply use the same delegate, `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
```

Events can be handled in one of two ways: statically or dynamically. Statically handling events is simpler and only requires a `WithEvents` variable and a `Handles` clause. In the following example, class `Form1` statically handles the event `Click` of object `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
```

Dynamically handling events is more complex because the event must be explicitly connected and disconnected to in code. The statement `AddHandler` adds a handler for an event, and the statement `RemoveHandler` removes a handler for an event. The next example shows a class `Form1` that adds `Button1_Click` as an event handler for `Button1`'s `Click` event:

```
    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
```

In method `Disconnect`, the event handler is removed.

9.4.1 Custom Events

As discussed in the previous section, event declarations implicitly define a field, an `add_` method, and a `remove_` method that are used to keep track of event handlers. In some situations, however, it may be desirable to provide custom code for tracking event handlers. For example, if a class defines forty events of which only a few will ever be handled, using a hash table instead of forty fields to track the handlers for each event may be more efficient. *Custom events* allow the `add_X` and `remove_X` methods to be defined explicitly, which enables custom storage for event handlers.

Custom events are declared in the same way that events that specify a delegate type are declared, with the exception that the keyword `Custom` must precede the `Event` keyword. A custom event declaration contains three declarations: an `AddHandler` declaration, a `RemoveHandler` declaration and a `RaiseEvent` declaration. None of the declarations can have any modifiers, although they can have attributes.

```
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
    ;
```

For example:

```
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
```

The `AddHandler` and `RemoveHandler` declaration take one `ByVal` parameter, which must be of the delegate type of the event. When an `AddHandler` or `RemoveHandler` statement is executed (or a `Handles` clause automatically handles an event), the corresponding declaration will be called. The `RaiseEvent` declaration takes the same parameters as the event delegate and will be called when a `RaiseEvent` statement is executed. All of the declarations must be provided and are considered to be subroutines.

Note that `AddHandler`, `RemoveHandler` and `RaiseEvent` declarations have the same restriction on line placement that subroutines have. The beginning statement, end statement and block must all appear at the beginning of a logical line.

In addition to the member name added to the type's declaration space, a custom event declaration implicitly declares several other members. Given an event named `X`, the following members are added to the declaration space:

- A method named `add_X`, corresponding to the `AddHandler` declaration.
- A method named `remove_X`, corresponding to the `RemoveHandler` declaration.
- A method named `fire_X`, corresponding to the `RaiseEvent` declaration.

If a type attempts to declare a name that matches one of the above names, a compile-time error will result, and the implicit declarations are all ignored for the purposes of name binding. It is not possible to override or overload any of the introduced members, although it is possible to shadow them in derived types.

Note. `Custom` is not a reserved word.

9.4.1.1 Custom events in WinRT assemblies

As of Microsoft Visual Basic 11.0, events declared in a file compiled with `/target:winmdobj`, or declared in an interface in such a file and then implemented elsewhere, are treated a little differently.

- External tools used to build the winmd will typically allow only certain delegate types such as `System.EventHandler(Of T)` or `System.TypedEventHandle(Of T, U)`, and will disallow others.
- The `XEvent` field has type `System.Runtime.InteropServices.WindowsRuntime.EventRegistrationTokenTable(Of T)` where `T` is the delegate type.
- The AddHandler accessor returns a `System.Runtime.InteropServices.WindowsRuntime.EventRegistrationToken`, and the RemoveHandler accessor takes a single parameter of the same type.

Here is an example of such a custom event.

```
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 Class
```

9.5 Constants

A *constant* is a constant value that is a member of a type.

```
ConstantMemberDeclaration:
    | Attributes? ConstantModifier* 'Const' ConstantDeclarators StatementTerminator
    ;

ConstantModifier:
    | AccessModifier
    | 'Shadows'
    ;

ConstantDeclarators:
    | ConstantDeclarator ( Comma ConstantDeclarator  )*
    ;

ConstantDeclarator:
    | Identifier ( 'As' TypeName  )? Equals ConstantExpression StatementTerminator
    ;
```

Constants are implicitly shared. If the declaration contains an `As` clause, the clause specifies the type of the member introduced by the declaration. If the type is omitted then the type of the constant is inferred. The type of a constant may only be a primitive type or `Object`. If a constant is typed as `Object` and there is no type character, the real type of the constant will be the type of the constant expression. Otherwise, the type of the constant is the type of the constant's type character.

The following example shows a class named `Constants` that has two public constants:

```
Class Constants
    Public Const A As Integer = 1
    Public Const B As Integer = A + 1
End Class
```

Constants can be accessed through the class, as in the following example, which prints out the values of `Constants.A` and `Constants.B`.

```
Module Test
    Sub Main()
        Console.WriteLine(Constants.A & ", " & Constants.B)
    End Sub
End Module
```

A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants. The following example declares three constants in one declaration statement.

```
Class A
    Protected Const x As Integer = 1, y As Long = 2, z As Short = 3
End Class
```

This declaration is equivalent to the following:

```
Class A
    Protected Const x As Integer = 1
    Protected Const y As Long = 2
    Protected Const z As Short = 3
End Class
```

The accessibility domain of the type of the constant must be the same as or a superset of the accessibility domain of the constant itself. The constant expression must yield a value of the constant's type or of a type that is implicitly convertible to the constant's type. The constant expression may not be circular; that is, a constant may not be defined in terms of itself.

The compiler automatically evaluates the constant declarations in the appropriate order. In the following example, the compiler first evaluates `Y`, then `Z`, and finally `X`, producing the values 10, 11, and 12, respectively.

```
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
```

When a symbolic name for a constant value is desired, but the type of the value is not permitted in a constant declaration or when the value cannot be computed at compile time by a constant expression, a read-only variable may be used instead.

9.6 Instance and Shared Variables

An instance or shared variable is a member of a type that can store information.

```
VariableMemberDeclaration:
    | Attributes? VariableModifier+ VariableDeclarators StatementTerminator
    ;

VariableModifier:
    | AccessModifier
    | 'Shadows'
    | 'Shared'
    | 'ReadOnly'
    | 'WithEvents'
    | 'Dim'
    ;

VariableDeclarators:
    | VariableDeclarator ( Comma VariableDeclarator  )*
    ;

VariableDeclarator:
    | VariableIdentifiers 'As' ObjectCreationExpression
    | VariableIdentifiers ( 'As' TypeName  )? ( Equals Expression  )?
    ;

VariableIdentifiers:
    | VariableIdentifier ( Comma VariableIdentifier  )*
    ;

VariableIdentifier:
    | Identifier IdentifierModifiers
    ;
```

The `Dim` modifier must be specified if no modifiers are specified, but may be omitted otherwise. A single variable declaration may include multiple variable declarators; each variable declarator introduces a new instance or shared member.

If an initializer is specified, only one instance or shared variable may be declared by the variable declarator:

```
Class Test
    Dim a, b, c, d As Integer = 10  ' Invalid: multiple initialization
End Class
```

This restriction does not apply to object initializers:

```
Class Test
    Dim a, b, c, d As New Collection() ' OK
End Class
```

A variable declared with the `Shared` modifier is a *shared variable*. A shared variable identifies exactly one storage location regardless of the number of instances of the type that are created. A shared variable comes into existence when a program begins executing, and ceases to exist when the program terminates.

A shared variable is shared only among instances of a particular closed generic type. For example, the program:

```
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
```

Prints out:

```
1
1
2
```

A variable declared without the `Shared` modifier is called an *instance variable*. Every instance of a class contains a separate copy of all instance variables of the class. An instance variable of a reference type comes into existence when a new instance of that type is created, and ceases to exist when there are no references to that instance and the `Finalize` method has executed. An instance variable of a value type has exactly the same lifetime as the variable to which it belongs. In other words, when a variable of a value type comes into existence or ceases to exist, so does the instance variable of the value type.

If the declarator contains an `As` clause, the clause specifies the type of the members introduced by the declaration. If the type is omitted and strict semantics are being used, a compile-time error occurs. Otherwise the type of the members is implicitly `Object` or the type of the members' type character.

Note. There is no ambiguity in the syntax: if a declarator omits a type, it will always use the type of a following declarator.

The accessibility domain of an instance or shared variable's type or array element type must be the same as or a superset of the accessibility domain of the instance or shared variable itself.

The following example shows a `Color` class that has internal instance variables named `redPart`, `greenPart`, and `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
```

9.6.1 Read-Only Variables

When an instance or shared variable declaration includes a `ReadOnly` modifier, assignments to the variables introduced by the declaration may only occur as part of the declaration or in a constructor in the same class. Specifically, assignments to a read-only instance or shared variable are permitted only in the following situations:

- In the variable declaration that introduces the instance or shared variable (by including a variable initializer in the declaration).
- For an instance variable, in the instance constructors of the class that contains the variable declaration. The instance variable can only be accessed in an unqualified manner or through `Me` or `MyClass`.
- For a shared variable, in the shared constructor of the class that contains the shared variable declaration.

A shared read-only variable is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a constant declaration, or when the value cannot be computed at compile time by a constant expression.

An example of the first such application follows, in which color shared variables are declared `ReadOnly` to prevent them from being changed by other programs:

```
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
```

Constants and read-only shared variables have different semantics. When an expression references a constant, the value of the constant is obtained at compile time, but when an expression references a read-only shared variable, the value of the shared variable is not obtained until run time. Consider the following application, which consists of two separate programs.

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
```

The namespaces `Program1` and `Program2` denote two programs that are compiled separately. Because variable `Program1.Utils.X` is declared as `Shared ReadOnly`, the value output by the `Console.WriteLine` statement is not known at compile time, but rather is obtained at run time. Thus, if the value of `X` is changed and `Program1` is recompiled, the `Console.WriteLine` statement will output the new value even if `Program2` is not recompiled. However, if `X` had been a constant, the value of `X` would have been obtained at the time `Program2` was compiled, and would have remained unaffected by changes in `Program1` until `Program2` was recompiled.

9.6.2 WithEvents Variables

A type can declare that it handles some set of events raised by one of its instance or shared variables by declaring the instance or shared variable that raises the events with the `WithEvents` modifier. For example:

```
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
```

In this example, the method `E1Handler` handles the event `E1` that is raised by the instance of the type `Raiser` stored in the instance variable `x`.

The `WithEvents` modifier causes the variable to be renamed with a leading underscore and replaced with a property of the same name that does the event hookup. For example, if the variable's name is `F`, it is renamed to `_F` and a property `F` is implicitly declared. If there is a collision between the variable's new name and another declaration, a compile-time error will be reported. Any attributes applied to the variable are carried over to the renamed variable.

The implicit property created by a `WithEvents` declaration takes care of hooking and unhooking the relevant event handlers. When a value is assigned to the variable, the property first calls the `remove` method for the event on the instance currently in the variable (unhooking the existing event handler, if any). Next the assignment is made, and

the property calls the `add` method for the event on the new instance in the variable (hooking up the new event handler). The following code is equivalent to the code above for the standard module `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
```

It is not valid to declare an instance or shared variable as `WithEvents` if the variable is typed as a structure. In addition, `WithEvents` may not be specified in a structure, and `WithEvents` and `ReadOnly` cannot be combined.

9.6.3 Variable Initializers

Instance and shared variable declarations in classes and instance variable declarations (but not shared variable declarations) in structures may include variable initializers. For `Shared` variables, variable initializers correspond to assignment statements that are executed after the program begins, but before the `Shared` variable is first referenced. For instance variables, variable initializers correspond to assignment statements that are executed when an instance of the class is created. Structures cannot have instance variable initializers because their parameterless constructors cannot be modified.

Consider the following example:

```
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
```

The example produces the following output:

```
x = 1.4142135623731, i = 100, s = Hello
```

An assignment to `x` occurs when the class is loaded, and assignments to `i` and `s` occur when a new instance of the class is created.

It is useful to think of variable initializers as assignment statements that are automatically inserted in the block of the type's constructor. The following example contains several instance variable initializers.

```
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
```

The example corresponds to the code shown below, where each comment indicates an automatically inserted statement.

```
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
```

All variables are initialized to the default value of their type before any variable initializers are executed. For example:

```
    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
```

Because `b` is automatically initialized to its default value when the class is loaded and `i` is automatically initialized to its default value when an instance of the class is created, the preceding code produces the following output:

```
    b = False, i = 0
```

Each variable initializer must yield a value of the variable's type or of a type that is implicitly convertible to the variable's type. A variable initializer may be circular or refer to a variable that will be initialized after it, in which case the value of the referenced variable is its default value for the purposes of the initializer. Such an initializer is of dubious value.

There are three forms of variable initializers: regular initializers, array-size initializers, and object initializers. The first two forms appear after an equal sign that follows the type name, the latter two are part of the declaration itself. Only one form of initializer may be used on any particular declaration.

9.6.3.1 Regular Initializers

A *regular initializer* is an expression that is implicitly convertible to the type of the variable. It appears after an equal sign that follows the type name and must be classified as a value. For example:

```
    Module Test
        Dim x As Integer = 10
        Dim y As Integer = 20
```

```
        Sub Main()
            Console.WriteLine("x = " & x & ", y = " & y)
        End Sub
    End Module
```

This program produces the output:

```
    x = 10, y = 20
```

If a variable declaration has a regular initializer, then only a single variable can be declared at a time. For example:

```
    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
```

9.6.3.2 Object Initializers

An *object initializer* is specified using an object creation expression in the place of the type name. An object initializer is equivalent to a regular initializer assigning the result of the object creation expression to the variable. So

```
    Module TestModule
        Sub Main()
            Dim x As New Test(10)
        End Sub
    End Module
```

is equivalent to

```
    Module TestModule
        Sub Main()
            Dim x As Test = New Test(10)
        End Sub
    End Module
```

The parenthesis in an object initializer is always interpreted as the argument list for the constructor and never as array type modifiers. A variable name with an object initializer cannot have an array type modifier or a nullable type modifier.

9.6.3.3 Array-Size Initializers

An *array-size initializer* is a modifier on the name of the variable that gives a set of dimension upper bounds denoted by expressions.

```
ArraySizeInitializationModifier:
    | OpenParenthesis BoundList CloseParenthesis ArrayTypeModifiers?
    ;

BoundList:
    | Bound ( Comma Bound  )*
    ;

Bound:
    | Expression
    | '0' 'To' Expression
    ;
```

The upper bound expressions must be classified as values and must be implicitly convertible to `Integer`. The set of upper bounds is equivalent to a variable initializer of an array-creation expression with the given upper bounds. The number of dimensions of the array type is inferred from the array size initializer. So

```
Module Test
    Sub Main()
        Dim x(5, 10) As Integer
    End Sub
End Module
```

is equivalent to

```
Module Test
    Sub Main()
        Dim x As Integer(,) = New Integer(5, 10) {}
    End Sub
End Module
```

All upper bounds must be equal to or greater than -1, and all dimensions must have an upper bound specified. If the element type of the array being initialized is itself an array type, the array-type modifiers go to the right of the array-size initializer. For example

```
Module Test
    Sub Main()
        Dim x(5,10)(,,) As Integer
    End Sub
End Module
```

declares a local variable `x` whose type is a two-dimensional array of three-dimensional arrays of `Integer`, initialized to an array with bounds of `0..5` in the first dimension and `0..10` in the second dimension. It is not possible to use an array size initializer to initialize the elements of a variable whose type is an array of arrays.

A variable declaration with an array-size initializer cannot have an array type modifier on its type or a regular initializer.

9.6.4 System.MarshalByRefObject Classes

Classes that derive from the class `System.MarshalByRefObject` are marshaled across context boundaries using proxies (that is, by reference) rather than through copying (that is, by value). This means that an instance of such a class may not be a true instance but instead may just be a stub that marshals variable accesses and method calls across a context boundary.

As a result, it is not possible to create a reference to the storage location of variables defined on such classes. This means that variables typed as classes derived from `System.MarshalByRefObject` cannot be passed to reference parameters, and methods and variables of variables typed as value types may not be accessed. Instead, Visual Basic treats variables defined on such classes as if they were properties (since the restrictions are the same on properties).

There is one exception to this rule: a member implicitly or explicitly qualified with `Me` is exempt from the above restrictions, because `Me` is always guaranteed to be an actual object, not a proxy.

9.7 Properties

Properties are a natural extension of variables; both are named members with associated types, and the syntax for accessing variables and properties is the same. Unlike variables, however, properties do not denote storage locations. Instead, properties have *accessors*, which specify the statements to execute in order to read or write their values.

Properties are defined with property declarations. The first part of a property declaration resembles a field declaration. The second part includes a `Get` accessor and/or a `Set` accessor.

```
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
    ;
```

In the example below, the `Button` class defines a `Caption` property.

```
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
```

Based on the `Button` class above, the following is an example of use of the `Caption` property:

```
Dim okButton As Button = New Button()

okButton.Caption = "OK" ' Invokes Set accessor.
Dim s As String = okButton.Caption ' Invokes Get accessor.
```

Here, the `Set` accessor is invoked by assigning a value to the property, and the `Get` accessor is invoked by referencing the property in an expression.

If no type is specified for a property and strict semantics are being used, a compile-time error occurs; otherwise the type of the property is implicitly `Object` or the type of the property's type character. A property declaration may contain either a `Get` accessor, which retrieves the value of the property, a `Set` accessor, which stores the value of the property, or both. Because a property implicitly declares methods, a property may be declared with the same modifiers as a method. If the property is defined in an interface or defined with the `MustOverride` modifier, the property body and the `End` construct must be omitted; otherwise, a compile-time error occurs.

The index parameter list makes up the signature of the property, so properties may be overloaded on index parameters but not on the type of the property. The index parameter list is the same as for a regular method. However, none of the parameters may be modified with the `ByRef` modifier and none of them may be named `Value` (which is reserved for the implicit value parameter in the `Set` accessor).

A property may be declared as follows:

- If the property specifies no property type modifier, the property must have both a `Get` accessor and a `Set` accessor. The property is said to be a read-write property.
- If the property specifies the `ReadOnly` modifier, the property must have a `Get` accessor and may not have a `Set` accessor. The property is said to be read-only property. It is a compile-time error for a read-only property to be the target of an assignment.
- If the property specifies the `WriteOnly` modifier, the property must have a `Set` accessor and may not have a `Get` accessor. The property is said to be write-only property. It is a compile-time error to reference a write-only property in an expression except as the target of an assignment or as an argument to a method.

The `Get` and `Set` accessors of a property are not distinct members, and it is not possible to declare the accessors of a property separately. The following example does not declare a single read-write property. Rather, it declares two properties with the same name, one read-only and one write-only:

```
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
```

Since two members declared in the same class cannot have the same name, the example causes a compile-time error.

By default, the accessibility of a property's `Get` and `Set` accessors is the same as the accessibility of the property itself. However, the `Get` and `Set` accessors can also specify accessibility separately from the property. In that case, the accessibility of an accessor must be more restrictive than the accessibility of the property and only one accessor can have a different accessibility level from the property. Access types are considered more or less restrictive as follows:

- `Private` is more restrictive than `Public`, `Protected Friend`, `Protected`, or `Friend`.
- `Friend` is more restrictive than `Protected Friend` or `Public`.
- `Protected` is more restrictive than `Protected Friend` or `Public`.
- `Protected Friend` is more restrictive than `Public`.

When one of a property's accessors is accessible but the other one is not, the property is treated as if it was read-only or write-only. For example:

```
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
```

When a derived type shadows a property, the derived property hides the shadowed property with respect to both reading and writing. In the following example, the `P` property in `B` hides the `P` property in `A` with respect to both reading and writing:

```
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
```

The accessibility domain of the return type or parameter types must be the same as or a superset of the accessibility domain of the property itself. A property may only have one `Set` accessor and one `Get` accessor.

Except for differences in declaration and invocation syntax, `Overridable`, `NotOverridable`, `Overrides`, `MustOverride`, and `MustInherit` properties behave exactly like `Overridable`, `NotOverridable`, `Overrides`, `MustOverride`, and `MustInherit` methods. When a property is overridden, the overriding property must be of the same type (read-write, read-only, write-only). An `Overridable` property cannot contain a `Private` accessor.

In the following example `X` is an `Overridable` read-only property, `Y` is an `Overridable` read-write property, and `Z` is a `MustOverride` read-write property.

```
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
```

Because `Z` is `MustOverride`, the containing class `A` must be declared `MustInherit`.

By contrast, a class that derives from class `A` is shown below:

```
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
```

Here, the declarations of properties `X`,`Y`, and `Z` override the base properties. Each property declaration exactly matches the accessibility modifiers, type, and name of the corresponding inherited property. The `Get` accessor of property `X` and the `Set` accessor of property `Y` use the `MyBase` keyword to access the inherited properties. The declaration of property `Z` overrides the `MustOverride` property -- thus, there are no outstanding `MustOverride` members in class `B`, and `B` is permitted to be a regular class.

Properties can be used to delay initialization of a resource until the moment it is first referenced. For example:

```
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
```

The `ConsoleStreams` class contains three properties, `In`, `Out`, and `Error`, that represent the standard input, output, and error devices, respectively. By exposing these members as properties, the `ConsoleStreams` class can delay their initialization until they are actually used. For example, upon first referencing the `Out` property, as in `ConsoleStreams.Out.WriteLine("hello, world")`, the underlying `TextWriter` for the output device is initialized. But if the application makes no reference to the `In` and `Error` properties, then no objects are created for those devices.

9.7.1 Get Accessor Declarations

A `Get` accessor (getter) is declared by using a property `Get` declaration. A property `Get` declaration consists of the keyword `Get` followed by a statement block. Given a property named `P`, a `Get` accessor declaration implicitly declares a method with the name `get_P` with the same modifiers, type, and parameter list as the property. If the type contains a declaration with that name, a compile-time error results, but the implicit declaration is ignored for the purposes of name binding.

A special local variable, which is implicitly declared in the `Get` accessor body's declaration space with the same name as the property, represents the return value of the property. The local variable has special name resolution semantics when used in expressions. If the local variable is used in a context that expects an expression that is classified as a method group, such as an invocation expression, then the name resolves to the function rather than to the local variable. For example:

```
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
```

The use of parentheses can cause ambiguous situations (such as `F(1)` where `F` is a property whose type is a one-dimensional array). In all ambiguous situations, the name resolves to the property rather than the local variable. For example:

```
    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
```

When control flow leaves the `Get` accessor body, the value of the local variable is passed back to the invocation expression. Because invoking a `Get` accessor is conceptually equivalent to reading the value of a variable, it is considered bad programming style for `Get` accessors to have observable side effects, as illustrated in the following example:

```
    Class Counter
        Private Value As Integer

        Public ReadOnly Property NextValue() As Integer
            Get
                Value += 1
                Return Value
            End Get
        End Property
    End Class
```

The value of the `NextValue` property depends on the number of times the property has previously been accessed. Thus, accessing the property produces an observable side effect, and the property should instead be implemented as a method.

The "no side effects" convention for `Get` accessors does not mean that `Get` accessors should always be written to simply return values stored in variables. Indeed, `Get` accessors often compute the value of a property by accessing multiple variables or invoking methods. However, a properly designed `Get` accessor performs no actions that cause observable changes in the state of the object.

Note. `Get` accessors have the same restriction on line placement that subroutines have. The beginning statement, end statement and block must all appear at the beginning of a logical line.

```
PropertyGetDeclaration:
    | Attributes? AccessModifier? 'Get' LineTerminator
      Block?
      'End' 'Get' StatementTerminator
    ;
```

9.7.2 Set Accessor Declarations

A `Set` accessor (setter) is declared by using a property set declaration. A property set declaration consists of the keyword `Set`, an optional parameter list, and a statement block. Given a property named `P`, a setter declaration implicitly declares a method with the name `set_P` with the same modifiers and parameter list as the property. If the type contains a declaration with that name, a compile-time error results, but the implicit declaration is ignored for the purposes of name binding.

If a parameter list is specified, it must have one member, that member must have no modifiers except `ByVal`, and its type must be the same as the type of the property. The parameter represents the property value being set. If the parameter is omitted, a parameter named `Value` is implicitly declared.

Note. `Set` accessors have the same restriction on line placement that subroutines have. The beginning statement, end statement and block must all appear at the beginning of a logical line.

```
PropertySetDeclaration:
    | Attributes? AccessModifier? 'Set'
      ( OpenParenthesis ParameterList? CloseParenthesis  )? LineTerminator
      Block?
      'End' 'Set' StatementTerminator
    ;
```

9.7.3 Default Properties

A property that specifies the modifier `Default` is called a *default property*. Any type that allows properties may have a default property, including interfaces. The default property may be referenced without having to qualify the instance with the name of the property. Thus, given a class

```
Class Test
    Public Default ReadOnly Property Item(i As Integer) As Integer
        Get
            Return i
        End Get
    End Property
End Class
```

the code

```
Module TestModule
    Sub Main()
        Dim x As Test = New Test()
        Dim y As Integer

        y = x(10)
    End Sub
End Module
```

is equivalent to

```
Module TestModule
    Sub Main()
        Dim x As Test = New Test()
        Dim y As Integer

        y = x.Item(10)
    End Sub
End Module
```

Once a property is declared `Default`, all of the properties overloaded on that name in the inheritance hierarchy become the default property, whether they have been declared `Default` or not. Declaring a property `Default` in a derived class when the base class declared a default property by another name does not require any other modifiers such as `Shadows` or `Overrides`. This is because the default property has no identity or signature and so cannot be shadowed or overloaded. For example:

```
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
```

This program will produce the output:

```
MoreDerived = 10
Derived = 10
Base = 10
```

All default properties declared within a type must have the same name and, for clarity, must specify the `Default` modifier. Because a default property with no index parameters would cause an ambiguous situation when assigning instances of the containing class, default properties must have index parameters. Furthermore, if one property overloaded on a particular name includes the `Default` modifier, all properties overloaded on that name must specify it. Default properties may not be `Shared`, and at least one accessor of the property must not be `Private`.

9.7.4 Automatically Implemented Properties

If a property omits declaration of any accessors, an implementation of the property will be automatically supplied unless the property is declared in an interface or is declared `MustOverride`. Only read/write properties with no arguments can be automatically implemented; otherwise, a compile-time error occurs.

An automatically implemented property `x`, even one overriding another property, introduces a private local variable `_x` with the same type as the property. If there is a collision between the local variable's name and another declaration, a compile-time error will be reported. The automatically implemented property's `Get` accessor returns the value of the local and the property's `Set` accessor that sets the value of the local. For example, the declaration:

```
Public Property x() As Integer
```

is roughly equivalent to:

```
Private _x As Integer
Public Property x() As Integer
    Get
        Return _x
    End Get
    Set (value As Integer)
        _x = value
    End Set
End Property
```

As with variable declarations, an automatically implemented property can include an initializer. For example:

```
Public Property x() As Integer = 10
Public Shared Property y() As New Customer() With { .Name = "Bob" }
```

Note. When an automatically implemented property is initialized, it is initialized through the property, not the underlying field. This is so overriding properties can intercept the initialization if they need to.

Array initializers are allowed on automatically implemented properties, except that there is no way to specify the array bounds explicitly. For example:

```
' 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
```

9.7.5 Iterator Properties

An *iterator property* is a property with the `Iterator` modifier. It is used for the same reason an iterator method (Section §10.1.2) is used -- as a convenient way to generate a sequence, one which can be consumed by the `For Each` statement. The `Get` accessor of an iterator property is interpreted in the same way as an iterator method.

An iterator property must have an explicit `Get` accessor, and its type must be `IEnumerator`, or `IEnumerable`, or `IEnumerator(Of T)` or `IEnumerable(Of T)` for some `T`.

Here is an example of an iterator property:

```
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
```

9.8 Operators

Operators are methods that define the meaning of an existing Visual Basic operator for the containing class. When the operator is applied to the class in an expression, the operator is compiled into a call to the operator method defined in the class. Defining an operator for a class is also known as *overloading* the operator.

```
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'
    ;
```

It is not possible to overload an operator that already exists; in practice, this primarily applies to conversion operators. For example, it is not possible to overload the conversion from a derived class to a base class:

```
        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
```

Operators can also be overloaded in the common sense of the word:

```
        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
```

Operator declarations do not explicitly add names to the containing type's declaration space; however they do implicitly declare a corresponding method starting with the characters "op_". The following sections list the corresponding method names with each operator.

There are three classes of operators that can be defined: unary operators, binary operators and conversion operators. All operator declarations share certain restrictions:

- Operator declarations must always be `Public` and `Shared`. The `Public` modifier can be omitted in contexts where the modifier will be assumed.
- The parameters of an operator cannot be declared `ByRef`, `Optional` or `ParamArray`.
- The type of at least one of the operands or the return value must be the type that contains the operator.
- There is no function return variable defined for operators. Therefore, the `Return` statement must be used to return values from an operator body.

The only exception to these restrictions applies to nullable value types. Since nullable value types do not have an actual type definition, a value type can declare user-defined operators for the nullable version of the type. When determining whether a type can declare a particular user-defined operator, the `?` modifiers are first dropped off of all of the types involved in the declaration for the purposes of validity checking. This relaxation does not apply to the return type of the `IsTrue` and `IsFalse` operators; they must still return `Boolean`, not `Boolean?`.

The precedence and associativity of an operator cannot be modified by an operator declaration.

Note. Operators have the same restriction on line placement that subroutines have. The beginning statement, end statement and block must all appear at the beginning of a logical line.

9.8.1 Unary Operators

The following unary operators can be overloaded:

- The unary plus operator `+` (corresponding method: `op_UnaryPlus`)
- The unary minus operator `-` (corresponding method: `op_UnaryNegation`)
- The logical `Not` operator (corresponding method: `op_OnesComplement`)
- The `IsTrue` and `IsFalse` operators (corresponding methods: `op_True`, `op_False`)

All overloaded unary operators must take a single parameter of the containing type and may return any type, except for `IsTrue` and `IsFalse`, which must return `Boolean`. If the containing type is a generic type, the type parameters must match the containing type's type parameters. For example,

```
Structure Complex
    ...

    Public Shared Operator +(v As Complex) As Complex
        Return v
    End Operator
End Structure
```

If a type overloads one of `IsTrue` or `IsFalse`, then it must overload the other as well. If only one is overloaded, a compile-time error results.

Note. `IsTrue` and `IsFalse` are not reserved words.

9.8.2 Binary Operators

The following binary operators can be overloaded:

- The addition `+`, subtraction `-`, multiplication `*`, division `/`, integral division `\`, modulo `Mod` and exponentiation `^` operators (corresponding method: `op_Addition`, `op_Subtraction`, `op_Multiply`, `op_Division`, `op_IntegerDivision`, `op_Modulus`, `op_Exponent`)
- The relational operators `=`, `<>`, `<`, `>`, `<=`, `>=` (corresponding methods: `op_Equality`, `op_Inequality`, `op_LessThan`, `op_GreaterThan`, `op_LessThanOrEqual`, `op_GreaterThanOrEqual`). **Note.** While the equality operator can be overloaded, the assignment operator (used only in assignment statements) cannot be overloaded.
- The `Like` operator (corresponding method: `op_Like`)

- The concatenation operator `&` (corresponding method: `op_Concatenate`)
- The logical `And`, `Or` and `Xor` operators (corresponding methods: `op_BitwiseAnd`, `op_BitwiseOr`, `op_ExclusiveOr`)
- The shift operators `<<` and `>>` (corresponding methods: `op_LeftShift`, `op_RightShift`)

All overloaded binary operators must take the containing type as one of the parameters. If the containing type is a generic type, the type parameters must match the containing type's type parameters. The shift operators further restrict this rule to require the first parameter to be of the containing type; the second parameter must always be of type `Integer`.

The following binary operators must be declared in pairs:

- Operator `=` and operator `<>`
- Operator `>` and operator `<`
- Operator `>=` and operator `<=`

If one of the pair is declared, then the other must also be declared with matching parameter and return types, or a compile-time error will result. (**Note.** The purpose of requiring paired declarations of relational operators is to try and ensure at least a minimum level of logical consistency in overloaded operators.)

In contrast to the relational operators, overloading both the division and integral division operators is strongly discouraged, although not an error. (**Note.** In general, the two types of division should be entirely distinct: a type that supports division is either integral (in which case it should support `\`) or not (in which case it should support `/`). We considered making it an error to define both operators, but because their languages do not generally distinguish between two types of division the way Visual Basic does, we felt it was safest to allow the practice but strongly discourage it.)

Compound assignment operators cannot be overloaded directly. Instead, when the corresponding binary operator is overloaded, the compound assignment operator will use the overloaded operator. For example:

```
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
```

9.8.3 Conversion Operators

Conversion operators define new conversions between types. These new conversions are called *user-defined conversions*. A conversion operator converts from a source type, indicated by the parameter type of the conversion operator, to a target type, indicated by the return type of the conversion operator. Conversions must be classified as either widening or narrowing. A conversion operator declaration that includes the `Widening` keyword introduces a user-defined widening conversion (corresponding method: `op_Implicit`). A conversion operator declaration that includes the `Narrowing` keyword introduces a user-defined narrowing conversion (corresponding method: `op_Explicit`).

In general, user-defined widening conversions should be designed to never throw exceptions and never lose information. If a user-defined conversion can cause exceptions (for example, because the source argument is out of range) or loss of information (such as discarding high-order bits), then that conversion should be defined as a narrowing conversion. In the example:

```
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
```

the conversion from `Digit` to `Byte` is a widening conversion because it never throws exceptions or loses information, but the conversion from `Byte` to `Digit` is a narrowing conversion since `Digit` can only represent a subset of the possible values of a `Byte`.

Unlike all other type members that can be overloaded, the signature of a conversion operator includes the target type of the conversion. This is the only type member for which the return type participates in the signature. The widening or narrowing classification of a conversion operator, however, is not part of the operator's signature. Thus, a class or structure cannot declare both a widening conversion operator and a narrowing conversion operator with the same source and target types.

A user-defined conversion operator must convert either to or from the containing type -- for example, it is possible for a class `C` to define a conversion from `C` to `Integer` and from `Integer` to `C`, but not from `Integer` to `Boolean`. If the containing type is a generic type, the type parameters must match the containing type's type parameters. Also, it is not possible to redefine an intrinsic (i.e. non-user-defined) conversion. As a result, a type cannot declare a conversion where:

- The source type and the destination type are the same.
- Both the source type and the destination type are not the type that defines the conversion operator.
- The source type or the destination type is an interface type.
- The source type and destination types are related by inheritance (including `Object`).

The only exception to these rules applies to nullable value types. Since nullable value types do not have an actual type definition, a value type can declare user-defined conversions for the nullable version of the type. When determining whether a type can declare a particular user-defined conversion, the `?` modifiers are first dropped off of all of the types involved in the declaration for the purposes of validity checking. Thus, the following declaration is valid because `S` can define a conversion from `S` to `T`:

```
Structure T
    ...
End Structure

Structure S
    Public Shared Widening Operator CType(ByVal v As S?) As T
    ...
    End Operator
End Structure
```

The following declaration is not valid, however, because structure `S` cannot define a conversion from `S` to `S`:

```
Structure S
    Public Shared Widening Operator CType(ByVal v As S) As S?
        ...
    End Operator
End Structure
```

9.8.4 Operator Mapping

Because the set of operators that Visual Basic supports may not exactly match the set of operators that other languages on the .NET Framework, some operators are mapped specially onto other operators when being defined or used. Specifically:

- Defining an integral division operator will automatically define a regular division operator (usable only from other languages) that will call the integral division operator.
- Overloading the `Not`, `And`, and `Or` operators will overload only the bitwise operator from the perspective of other languages that distinguish between logical and bitwise operators.
- A class that overloads only the logical operators in a language that distinguishes between logical and bitwise operators (i.e. a languages that uses `op_LogicalNot`, `op_LogicalAnd`, and `op_LogicalOr` for `Not`, `And`, and `Or`, respectively) will have their logical operators mapped onto the Visual Basic logical operators. If both the logical and bitwise operators are overloaded, only the bitwise operators will be used.
- Overloading the `<<` and `>>` operators will overload only the signed operators from the perspective of other languages that distinguish between signed and unsigned shift operators.
- A class that overloads only an unsigned shift operator will have the unsigned shift operator mapped onto the corresponding Visual Basic shift operator. If both an unsigned and signed shift operator is overloaded, only the signed shift operator will be used.

10. Statements

Statements represent executable code.

```
Statement:
    | LabelDeclarationStatement
    | LocalDeclarationStatement
    | WithStatement
    | SyncLockStatement
    | EventStatement
    | AssignmentStatement
    | InvocationStatement
    | ConditionalStatement
    | LoopStatement
    | ErrorHandlingStatement
    | BranchStatement
    | ArrayHandlingStatement
    | UsingStatement
    | AwaitStatement
    | YieldStatement
    ;
```

Note. The Microsoft Visual Basic Compiler only allows statements which start with a keyword or an identifier. Thus, for instance, the invocation statement "`Call (Console).WriteLine`" is allowed, but the invocation statement "`(Console).WriteLine`" is not.

10.1 Control Flow

Control flow is the sequence in which statements and expressions are executed. The order of execution depends on the particular statement or expression.

For example, when evaluating an addition operator (Section §11.13.3), first the left operand is evaluated, then the right operand, and then the operator itself. Blocks are executed (Section §10.1.4) by first executing their first substatement, and then proceeding one by one through the statements of the block.

Implicit in this ordering is the concept of a *control point*, which is the next operation to be executed. When a method is invoked (or "called"), we say it creates an *instance* of the method. A method instance consists of its own copy of the method's parameters and local variables, and its own control point.

10.1.1 Regular Methods

Here is an example of a regular method

```
Function Test() As Integer
    Console.WriteLine("hello")
    Return 1
End Sub

Dim x = Test()    ' invokes the function, prints "hello", assigns 1 to x
```

When a regular method is invoked,

1. First an instance of the method is created specific to that invocation. This instance includes a copy of all parameters and local variables of the method.

2. Then all of its parameters are initialized to the supplied values, and all of its local variables to the default values of their types.
3. In the case of a `Function`, an implicit local variable is also initialized called the *function return variable* whose name is the function's name, whose type is the return type of the function and whose initial value is the default of its type.
4. The method instance's control point is then set at the first statement of the method body, and the method body immediately starts to execute from there (Section §10.1.4).

When control flow exits the method body normally - through reaching the `End Function` or `End Sub` that mark its end, or through an explicit `Return` or `Exit` statement - control flow returns to the caller of the method instance. If there is a function return variable, the result of the invocation is the final value of this variable.

When control flow exits the method body through an unhandled exception, that exception is propagated to the caller.

After control flow has exited, there are no longer any live references to the method instance. If the method instance held the only references to its copy of local variables or parameters, then they may be garbage collected.

10.1.2 Iterator Methods

Iterator methods are used as a convenient way to generate a sequence, one which can be consumed by the `For Each` statement. Iterator methods use the `Yield` statement (Section §10.15) to provide elements of the sequence. (An iterator method with no `Yield` statements will produce an empty sequence). Here is an example of an iterator method:

```
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
```

When an iterator method is invoked whose return type is `IEnumerator(Of T)`,

1. First an instance of the iterator method is created specific to that invocation. This instance includes a copy of all parameters and local variables of the method.
2. Then all of its parameters are initialized to the supplied values, and all of its local variables to the default values of their types.
3. An implicit local variable is also initialized called the *iterator current variable*, whose type is `T` and whose initial value is the default of its type.
4. The method instance's control point is then set at the first statement of the method body.
5. An *iterator object* is then created, associated with this method instance. The iterator object implements the declared return type and has behavior as described below.
6. Control flow is then resumed *immediately* in the caller, and the result of the invocation is the iterator object. Note that this transfer is done without exiting the iterator method instance, and does not cause finally handlers to execute. The method instance is still referenced by the iterator object, and will not be garbage collected so long as there exists a live reference to the iterator object.

When the iterator object's `Current` property is accessed, the *current variable* of the invocation is returned.

When the iterator object's `MoveNext` method is invoked, the invocation does not create a new method instance. Instead the existing method instance is used (and its control point and local variables and parameters) - the instance that was created when the iterator method was first invoked. Control flow resumes execution at the control point of that method instance, and proceeds through the body of the iterator method as normal.

When the iterator object's `Dispose` method is invoked, again the existing method instance is used. Control flow resumes at the control point of that method instance, but then immediately behaves as if an `Exit Function` statement were the next operation.

The above descriptions of behavior for invocation of `MoveNext` or `Dispose` on an iterator object only apply if all previous invocations of `MoveNext` or `Dispose` on that iterator object have already returned to their callers. If they haven't, then the behavior is undefined.

When control flow exits the iterator method body normally -- through reaching the `End Function` that mark its end, or through an explicit `Return` or `Exit` statement -- it must have done so in the context of an invocation of `MoveNext` or `Dispose` function on an iterator object to resume the iterator method instance, and it will have been using the method instance that was created when the iterator method was first invoked. The control point of that instance is left at the `End Function` statement, and control flow resumes in the caller; and if it had been resumed by a call to `MoveNext` then the value `False` is returned to the caller.

When control flow exits the iterator method body through an unhandled exception, then the exception is propagated to the caller, which again will be either an invocation of `MoveNext` or of `Dispose`.

As for the other possible return types of an iterator function,

- When an iterator method is invoked whose return type is `IEnumerable(Of T)` for some `T`, an instance is first created -- specific to that invocation of the iterator method -- of all parameters in the method, and they are initialized with the supplied values. The result of the invocation is an an object which implements the return type. Should this object's `GetEnumerator` method be called, it creates an instance -- specific to that invocation of `GetEnumerator` -- of all parameters and local variables in the method. It initializes the parameters to the values already saved, and proceeds as for the iterator method above.
- When an iterator method is invoked whose return type is the non-generic interface `IEnumerator`, the behavior is exactly as for `IEnumerator(Of Object)`.
- When an iterator method is invoked whose return type is the non-generic interface `IEnumerable`, the behavior is exactly as for `IEnumerable(Of Object)`.

10.1.3 Async Methods

Async methods are a convenient way to do long-running work without for example blocking the UI of an application. Async is short for *Asynchronous* - it means that the caller of the async method will resume its execution promptly, but the eventual completion of the async method's instance may happen at some later time in the future. By convention async methods are named with the suffix "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"
```

Note. Async methods are *not* run on a background thread. Instead they allow a method to suspend itself through the `Await` operator, and schedule itself to be resumed in response to some event.

When an async method is invoked

1. First an instance of the async method is created specific to that invocation. This instance includes a copy of all parameters and local variables of the method.
2. Then all of its parameters are initialized to the supplied values, and all of its local variables to the default values of their types.
3. In the case of an async method with return type `Task(Of T)` for some `T`, an implicit local variable is also initialized called the *task return variable*, whose type is `T` and whose initial value is the default of `T`.
4. If the async method is a `Function` with return type `Task` or `Task(Of T)` for some `T`, then an object of that type implicitly created, associated with the current invocation. This is called an *async object* and represents the future work that will be done by executing the instance of the async method. When control resumes in the caller of this async method instance, the caller will receive this async object as the result of its invocation.
5. The instance's control point is then set at the first statement of the async method body, and immediately starts to execute the method body from there (Section §10.1.4).

Resumption Delegate and Current Caller

As detailed in Section §11.25, execution of an `Await` expression has the ability to suspend the method instance's control point leaving control flow to go elsewhere. Control flow can later resume at the same instance's control point through invocation of a *resumption delegate*. Note that this suspension is done without exiting the async method, and does not cause finally handlers to execute. The method instance is still referenced by both the resumption delegate and the `Task` or `Task(Of T)` result (if any), and will not be garbage collected so long as there exists a live reference to either delegate or result.

It is helpful to imagine the statement `Dim x = Await WorkAsync()` approximately as syntactic shorthand for the following:

```
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()
```

In the following, the *current caller* of the method instance is defined as either the original caller, or the caller of the resumption delegate, whichever is more recent.

When control flow exits the async method body -- through reaching the `End Sub` or `End Function` that mark its end, or through an explicit `Return` or `Exit` statement, or through an unhandled exception -- the instance's control point is set to the end of the method. Behavior then depends on the return type of the async method.

- In the case of an `Async Function` with return type `Task`:

  a. If control flow exits through an unhandled exception, then the async object's status is set to `TaskStatus.Faulted` and its `Exception.InnerException` property is set to the exception (except: certain implementation-defined exceptions such as `OperationCanceledException` change it to `TaskStatus.Canceled`). Control flow resumes in the current caller.

  b. Otherwise, the async object's status is set to `TaskStatus.Completed`. Control flow resumes in the current caller.

  (**Note.** The whole point of Task, and what makes async methods interesting, is that when a task becomes Completed then any methods that were awaiting it will presently have their resumption delegates executed, i.e. they will become unblocked.)

- In the case of an `Async Function` with return type `Task(Of T)` for some `T`: the behavior is as above, except that in non-exception cases the async object's `Result` property is also set to the final value of the task return variable.

- In the case of an `Async Sub`:
  a. If control flow exits through an unhandled exception, then that exception is propagated to the environment in some implementation-specific manner. Control flow resumes in the current caller.
  b. Otherwise, control flow simply resumes in the current caller.

10.1.3.1 Async Sub

There is some Microsoft-specific behavior of an `Async Sub`.

If `SynchronizationContext.Current` is `Nothing` at the start of the invocation, then any unhandled exceptions from an Async Sub will be posted to the Threadpool.

If `SynchronizationContext.Current` is not `Nothing` at the start of the invocation, then `OperationStarted()` is invoked on that context before the start of the method and `OperationCompleted()` after the end. Additionally, any unhandled exceptions will be posted to be rethrown on the synchronization context.

This means that in UI applications, for an `Async Sub` that is invoked on the UI thread, any exceptions it fails to handle will be reposted the UI thread.

10.1.3.2 Mutable structures in async and iterator methods

Mutable structures in general are considered bad practice, and they are not supported by async or iterator methods. In particular, each invocation of an async or iterator method in a structure will implicitly operate on a *copy* of that structure that is copied at its moment of invocation. Thus, for example,

```
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"
```

10.1.4 Blocks and Labels

A group of executable statements is called a statement block. Execution of a statement block begins with the first statement in the block. Once a statement has been executed, the next statement in lexical order is executed, unless a statement transfers execution elsewhere or an exception occurs.

Within a statement block, the division of statements on logical lines is not significant with the exception of label declaration statements. A label is an identifier that identifies a particular position within the statement block that can be used as the target of a branch statement such as `GoTo`.

```
Block:
    | Statements*
    ;

LabelDeclarationStatement:
    | LabelName ':'
    ;

LabelName:
    | Identifier
    | IntLiteral
    ;

Statements:
```

```
    | Statement? ( ':' Statement?  )*
    ;
```

Label declaration statements must appear at the beginning of a logical line and labels may be either an identifier or an integer literal. Because both label declaration statements and invocation statements can consist of a single identifier, a single identifier at the beginning of a local line is always considered a label declaration statement. Label declaration statements must always be followed by a colon, even if no statements follow on the same logical line.

Labels have their own declaration space and do not interfere with other identifiers. The following example is valid and uses the name variable `x` both as a parameter and as a label.

```
Function F(x As Integer) As Integer
    If x >= 0 Then
        GoTo x
    End If
    x = -x
x:
    Return x
End Function
```

The scope of a label is the body of the method containing it.

For the sake of readability, statement productions that involve multiple substatements are treated as a single production in this specification, even though the substatements may each be by themselves on a labeled line.

10.1.5 Local Variables and Parameters

The preceding sections detail how and when method instances are created, and with them the copies of a method's local variables and parameters. In addition, each time the body of a loop is entered, a new copy is made of each local variable declared inside that loop as described in Section §10.9, and the method instance now contains this copy of its local variable rather than the previous copy.

All locals are initialized to their type's default value. Local variables and parameters are always publicly accessible. It is an error to refer to a local variable in a textual position that precedes its declaration, as the following example illustrates:

```
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
```

In the `F` method above, the first assignment to `i` specifically does not refer to the field declared in the outer scope. Rather, it refers to the local variable, and it is in error because it textually precedes the declaration of the variable. In the `G` method, a subsequent variable declaration refers to a local variable declared in an earlier variable declaration within the same local variable declaration.

Each block in a method creates a declaration space for local variables. Names are introduced into this declaration space through local variable declarations in the method body and through the parameter list of the method, which introduces names into the outermost block's declaration space. Blocks do not allow shadowing of names through nesting: once a name has been declared in a block, the name may not be redeclared in any nested blocks.

Thus, in the following example, the `F` and `G` methods are in error because the name `i` is declared in the outer block and cannot be redeclared in the inner block. However, the `H` and `I` methods are valid because the two `i`'s are declared in separate non-nested blocks.

```
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
```

When the method is a function, a special local variable is implicitly declared in the method body's declaration space with the same name as the method representing the return value of the function. The local variable has special name resolution semantics when used in expressions. If the local variable is used in a context that expects an expression classified as a method group, such as an invocation expression, then the name resolves to the function rather than to the local variable. For example:

```
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
```

The use of parentheses can cause ambiguous situations (such as `F(1)`, where `F` is a function whose return type is a one-dimensional array); in all ambiguous situations, the name resolves to the function rather than the local variable. For example:

```
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
```

When control flow leaves the method body, the value of the local variable is passed back to the invocation expression. If the method is a subroutine, there is no such implicit local variable, and control simply returns to the invocation expression.

10.2 Local Declaration Statements

A local declaration statement declares a new local variable, local constant, or static variable. *Local variables* and *local constants* are equivalent to instance variables and constants scoped to the method and are declared in the same way. *Static variables* are similar to `Shared` variables and are declared using the `Static` modifier.

```
LocalDeclarationStatement:
    | LocalModifier VariableDeclarators StatementTerminator
    ;

LocalModifier:
    | 'Static' | 'Dim' | 'Const'
    ;
```

Static variables are locals that retain their value across invocations of the method. Static variables declared within non-shared methods are per instance: each instance of the type that contains the method has its own copy of the static variable. Static variables declared within `Shared` methods are per type; there is only one copy of the static variable for all instances. While local variables are initialized to their type's default value upon each entry into the method, static variables are only initialized to their type's default value when the type or type instance is initialized. Static variables may not be declared in structures or generic methods.

Local variables, local constants, and static variables always have public accessibility and may not specify accessibility modifiers. If no type is specified on a local declaration statement, then the following steps determine the type of the local declaration:

1. If the declaration has a type character, the type of the type character is the type of the local declaration.
2. If the local declaration is a local constant, or if the local declaration is a local variable with an initializer and local variable type inference is being used, the type of the local declaration is inferred from the type of the initializer. If the initializer refers to the local declaration, a compile-time error occurs. (Local constants are required to have initializers.)
3. If strict semantics are not being used, the type of the local declaration statement is implicitly `Object`.
4. Otherwise, a compile-time error occurs.

If no type is specified on a local declaration statement that has an array size or array type modifier, then the type of the local declaration is an array with the specified rank and the previous steps are used to determine the element type of the array. If local variable type inference is used, the type of the initializer must be an array type with the same array shape (i.e. array type modifiers) as the local declaration statement. Note that it is possible that the inferred element type may still be an array type. For example:

```
    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
```

If no type is specified on a local declaration statement that has a nullable type modifier, then the type of the local declaration is the nullable version of the inferred type or the inferred type itself if it a nullable value type already. If the inferred type is not a value type that can be made nullable, a compile-time error occurs. If both a nullable type modifier and an array size or array type modifier are placed on a local declaration statement with no type, then the nullable type modifier is considered to apply to the element type of the array and the previous steps are used to determine the element type.

Variable initializers on local declaration statements are equivalent to assignment statements placed at the textual location of the declaration. Thus, if execution branches over the local declaration statement, the variable initializer is not executed. If the local declaration statement is executed more than once, the variable initializer is executed an equal number of times. Static variables only execute their initializer the first time. If an exception occurs while initializing a static variable, the static variable is considered initialized with the default value of the static variable's type.

The following example shows the use of initializers:

```
    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
```

This program prints:

```
Static variable x = 5
Static variable x = 6
Static variable x = 7
Local variable y = 8
Local variable y = 8
Local variable y = 8
```

Initializers on static locals are thread-safe and protected against exceptions during initialization. If an exception occurs during a static local initializer, the static local will have its default value and not be initialized. A static local initializer

```
Module Test
    Sub F()
        Static x As Integer = 5
    End Sub
End Module
```

is equivalent to

```
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
```

Local variables, local constants, and static variables are scoped to the statement block in which they are declared. Static variables are special in that their names may only be used once throughout the entire method. For example, it is not valid to specify two static variable declarations with the same name even if they are in different blocks.

10.2.1 Implicit Local Declarations

In addition to local declaration statements, local variables can also be declared implicitly through use. A simple name expression that uses a name that does not resolve to something else declares a local variable by that name. For example:

```
Option Explicit Off
```

```
Module Test
    Sub Main()
        x = 10
        y = 20
        Console.WriteLine(x + y)
    End Sub
End Module
```

Implicit local declaration only occurs in expression contexts that can accept an expression classified as a variable. The exception to this rule is that a local variable may not be implicitly declared when it is the target of a function invocation expression, indexing expression, or a member access expression.

Implicit locals are treated as if they are declared at the beginning of the containing method. Thus, they are always scoped to the entire method body, even if declared inside of a lambda expression. For example, the following code:

```
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
```

will print the value `10`. Implicit locals are typed as `Object` if no type character was attached to the variable name; otherwise the type of the variable is the type of the type character. Local variable type inference is not used for implicit locals.

If explicit local declaration is specified by the compilation environment or by `Option Explicit`, all local variables must be explicitly declared and implicit variable declaration is disallowed.

10.3 With Statement

A `With` statement allows multiple references to an expression's members without specifying the expression multiple times.

```
WithStatement:
    | 'With' Expression StatementTerminator
      Block?
      'End' 'With' StatementTerminator
    ;
```

The expression must be classified as a value and is evaluated once, upon entry into the block. Within the `With` statement block, a member access expression or dictionary access expression starting with a period or an exclamation point is evaluated as if the `With` expression preceded it. For example:

```
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
```

It is invalid to branch into a `With` statement block from outside of the block.

10.4 SyncLock Statement

A `SyncLock` statement allows statements to be synchronized on an expression, which ensures that multiple threads of execution do not execute the same statements at the same time.

```
SyncLockStatement:
    | 'SyncLock' Expression StatementTerminator
      Block?
      'End' 'SyncLock' StatementTerminator
    ;
```

The expression must be classified as a value and is evaluated once, upon entry to the block. When entering the `SyncLock` block, the `Shared` method `System.Threading.Monitor.Enter` is called on the specified expression, which blocks until the thread of execution has an exclusive lock on the object returned by the expression. The type of the expression in a `SyncLock` statement must be a reference type. For example:

```
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
```

The example above synchronizes on the specific instance of the class `Test` to ensure that no more than one thread of execution can add or subtract from the count variable at a time for a particular instance.

The `SyncLock` block is implicitly contained by a `Try` statement whose `Finally` block calls the `Shared` method `System.Threading.Monitor.Exit` on the expression. This ensures the lock is freed even when an exception is thrown. As a result, it is invalid to branch into a `SyncLock` block from outside of the block, and a `SyncLock` block is treated as a single statement for the purposes of `Resume` and `Resume Next`. The above example is equivalent to the following code:

```
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
```

10.5 Event Statements

The `RaiseEvent`, `AddHandler`, and `RemoveHandler` statements raise events and handle events dynamically.

```
EventStatement:
    | RaiseEventStatement
    | AddHandlerStatement
    | RemoveHandlerStatement
    ;
```

10.5.1 RaiseEvent Statement

A `RaiseEvent` statement notifies event handlers that a particular event has occurred.

```
RaiseEventStatement:
    | 'RaiseEvent' IdentifierOrKeyword
      ( OpenParenthesis ArgumentList? CloseParenthesis  )? StatementTerminator
    ;
```

The simple name expression in a `RaiseEvent` statement is interpreted as a member lookup on `Me`. Thus, `RaiseEvent x` is interpreted as if it were `RaiseEvent Me.x`. The result of the expression must be classified as an event access for an event defined in the class itself; events defined on base types cannot be used in a `RaiseEvent` statement.

The `RaiseEvent` statement is processed as a call to the `Invoke` method of the event's delegate, using the supplied parameters, if any. If the delegate's value is `Nothing`, no exception is thrown. If there are no arguments, the parentheses may be omitted. For example:

```
    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
```

The class `Raiser` above is equivalent to:

```
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
```

10.5.2 AddHandler and RemoveHandler Statements

Although most event handlers are automatically hooked up through `WithEvents` variables, it may be necessary to dynamically add and remove event handlers at run time. `AddHandler` and `RemoveHandler` statements do this.

```
AddHandlerStatement:
    | 'AddHandler' Expression Comma Expression StatementTerminator
    ;

RemoveHandlerStatement:
    | 'RemoveHandler' Expression Comma Expression StatementTerminator
    ;
```

Each statement takes two arguments: the first argument must be an expression that is classified as an event access and the second argument must be an expression that is classified as a value. The second argument's type must be the delegate type associated with the event access. For example:

```
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
```

Given an event `E`, the statement calls the relevant `add_E` or `remove_E` method on the instance to add or remove the delegate as a handler for the event. Thus, the above code is equivalent to:

```
    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
```

10.6 Assignment Statements

An assignment statement assigns the value of an expression to a variable. There are several types of assignment.

```
AssignmentStatement:
    | RegularAssignmentStatement
    | CompoundAssignmentStatement
    | MidAssignmentStatement
    ;
```

10.6.1 Regular Assignment Statements

A simple assignment statement stores the result of an expression in a variable.

```
RegularAssignmentStatement:
    | Expression Equals Expression StatementTerminator
    ;
```

The expression on the left side of the assignment operator must be classified as a variable or a property access, while the expression on the right side of the assignment operator must be classified as a value. The type of the expression must be implicitly convertible to the type of the variable or property access.

If the variable being assigned into is an array element of a reference type, a run-time check will be performed to ensure that the expression is compatible with the array-element type. In the following example, the last assignment causes a `System.ArrayTypeMismatchException` to be thrown, because an instance of `ArrayList` cannot be stored in an element of a `String` array.

```
    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.
```

If the expression on the left side of the assignment operator is classified as a variable, then the assignment statement stores the value in the variable. If the expression is classified as a property access, then the assignment statement turns the property access into an invocation of the `Set` accessor of the property with the value substituted for the value parameter. For example:

```
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
```

If the target of the variable or property access is typed as a value type but not classified as a variable, a compile-time error occurs. For example:

```
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
```

Note that the semantics of the assignment depend on the type of the variable or property to which it is being assigned. If the variable to which it is being assigned is a value type, the assignment copies the value of the expression into the variable. If the variable to which it is being assigned is a reference type, the assignment copies the reference, not the value itself, into the variable. If the type of the variable is `Object`, the assignment semantics are determined by whether the value's type is a value type or a reference type at run time.

Note. For intrinsic types such as `Integer` and `Date`, reference and value assignment semantics are the same because the types are immutable. As a result, the language is free to use reference assignment on boxed intrinsic types as an optimization. From a value perspective, the result is the same.

Because the equals character (`=`) is used both for assignment and for equality, there is an ambiguity between a simple assignment and an invocation statement in situations such as `x = y.ToString()`. In all such cases, the assignment statement takes precedence over the equality operator. This means that the example expression is interpreted as `x = (y.ToString())` rather than `(x = y).ToString()`.

10.6.2 Compound Assignment Statements

A *compound assignment statement* takes the form `V op= E` (where `op` is a valid binary operator).

```
CompoundAssignmentStatement:
    | Expression CompoundBinaryOperator LineTerminator? Expression StatementTerminator
    ;

CompoundBinaryOperator:
    | '^' '=' | '*' '=' | '/' '=' | '\\' '=' | '+' '=' | '-' '='
    | '&' '=' | '<' '<' '=' | '>' '>' '='
    ;
```

The expression on the left side of the assignment operator must be classified as a variable or property access, while the expression on the right side of the assignment operator must be classified as a value. The compound assignment statement is equivalent to the statement `V = V op E` with the difference that the variable on the left side of the compound assignment operator is only evaluated once. The following example demonstrates this difference:

```
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
```

The expression `a(GetIndex())` is evaluated twice for simple assignment but only once for compound assignment, so the code prints:

```
Simple assignment
Getting index
Getting index
Compound assignment
Getting index
```

10.6.3 Mid Assignment Statement

A `Mid` assignment statement assigns a string into another string. The left side of the assignment has the same syntax as a call to the function `Microsoft.VisualBasic.Strings.Mid`.

```
MidAssignmentStatement:
    | 'Mid' '$'? OpenParenthesis Expression Comma Expression
      ( Comma Expression  )? CloseParenthesis Equals Expression StatementTerminator
    ;
```

The first argument is the target of the assignment and must be classified as a variable or a property access whose type is implicitly convertible to and from `String`. The second parameter is the 1-based start position that corresponds to where the assignment should begin in the target string and must be classified as a value whose type must be implicitly convertible to `Integer`. The optional third parameter is the number of characters from the right-side value to assign into the target string and must be classified as a value whose type is implicitly convertible to `Integer`. The right side is the source string and must be classified as a value whose type is implicitly convertible to `String`. The right side is truncated to the length parameter, if specified, and replaces the characters in the left-side string, starting at the start position. If the right side string contained fewer characters than the third parameter, only the characters from the right side string will be copied.

The following example displays `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
```

Note. `Mid` is not a reserved word.

10.7 Invocation Statements

An invocation statement invokes a method preceded by the optional keyword `Call`. The invocation statement is processed in the same way as the function invocation expression, with some differences noted below. The invocation expression must be classified as a value or void. Any value resulting from the evaluation of the invocation expression is discarded.

If the `Call` keyword is omitted, then the invocation expression must start with an identifier or keyword, or with `.` inside a `With` block. Thus, for instance, "`Call 1.ToString()`" is a valid statement but "`1.ToString()`" is not. (Note that in an expression context, invocation expressions also need not start with an identifier. For example, "`Dim x = 1.ToString()`" is a valid statement).

There is another difference between the invocation statements and invocation expressions: if an invocation statement includes an argument list, then this is always taken as the argument list of the invocation. The following example illustrates the difference:

```
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
    ;
```

10.8 Conditional Statements

Conditional statements allow conditional execution of statements based on expressions evaluated at run time.

```
ConditionalStatement:
    | IfStatement
    | SelectStatement
    ;
```

10.8.1 If...Then...Else Statements

An `If...Then...Else` statement is the basic conditional statement.

```
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
    ;
```

Each expression in an `If...Then...Else` statement must be a Boolean expression, as per Section §11.19. (Note: this does not require the expression to have Boolean type). If the expression in the `If` statement is true, the statements enclosed by the `If` block are executed. If the expression is false, each of the `ElseIf` expressions is evaluated. If one of the `ElseIf` expressions evaluates to true, the corresponding block is executed. If no expression evaluates to true and there is an `Else` block, the `Else` block is executed. Once a block finishes executing, execution passes to the end of the `If...Then...Else` statement.

The line version of the `If` statement has a single set of statements to be executed if the `If` expression is `True` and an optional set of statements to be executed if the expression is `False`. For example:

```
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
```

The line version of the If statement binds less tightly than ":", and its `Else` binds to the lexically nearest preceding `If` that is allowed by the syntax. For example, the following two versions are equivalent:

```
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
```

All statements other than label declaration statements are allowed inside a line `If` statement, including block statements. However, they may not use LineTerminators as StatementTerminators except inside multi-line lambda expressions. For example:

```
' 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()
```

10.8.2 Select Case Statements

A `Select Case` statement executes statements based on the value of an expression.

```
SelectStatement:
    | 'Select' 'Case'? Expression StatementTerminator
      CaseStatement*
      CaseElseStatement?
      'End' 'Select' StatementTerminator
    ;

CaseStatement:
    | 'Case' CaseClauses StatementTerminator
      Block?
    ;

CaseClauses:
    | CaseClause ( Comma CaseClause  )*
    ;

CaseClause:
    | ( 'Is' LineTerminator?  )? ComparisonOperator LineTerminator? Expression
    | Expression ( 'To' Expression  )?
    ;

ComparisonOperator:
    | '=' | '<' '>' | '<' | '>' | '>' '=' | '<' '='
    ;

CaseElseStatement:
    | 'Case' 'Else' StatementTerminator
      Block?
    ;
```

The expression must be classified as a value. When a `Select Case` statement is executed, the `Select` expression is evaluated first, and the `Case` statements are then evaluated in order of textual declaration. The first `Case` statement that evaluates to `True` has its block executed. If no `Case` statement evaluates to `True` and there is a `Case Else` statement, that block is executed. Once a block has finished executing, execution passes to the end of the `Select` statement.

Execution of a `Case` block is not permitted to "fall through" to the next switch section. This prevents a common class of bugs that occur in other languages when a `Case` terminating statement is accidentally omitted. The following example illustrates this behavior:

```
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
```

The code prints:

```
x = 10
```

Although `Case 10` and `Case 20 - 10` select for the same value, `Case 10` is executed because it precedes `Case 20 - 10` textually. When the next `Case` is reached, execution continues after the `Select` statement.

A `Case` clause may take two forms. One form is an optional `Is` keyword, a comparison operator, and an expression. The expression is converted to the type of the `Select` expression; if the expression is not implicitly convertible to the type of the `Select` expression, a compile-time error occurs. If the `Select` expression is *E*, the comparison operator is *Op*, and the `Case` expression is *E1*, the case is evaluated as *E OP E1*. The operator must be valid for the types of the two expressions; otherwise a compile-time error occurs.

The other form is an expression optionally followed by the keyword `To` and a second expression. Both expressions are converted to the type of the `Select` expression; if either expression is not implicitly convertible to the type of the `Select` expression, a compile-time error occurs. If the `Select` expression is `E`, the first `Case` expression is `E1`, and the second `Case` expression is `E2`, the `Case` is evaluated either as `E = E1` (if no `E2` is specified) or `(E >= E1) And (E <= E2)`. The operators must be valid for the types of the two expressions; otherwise a compile-time error occurs.

10.9 Loop Statements

Loop statements allow repeated execution of the statements in their body.

```
LoopStatement:
    | WhileStatement
    | DoLoopStatement
    | ForStatement
    | ForEachStatement
    ;
```

Each time a loop body is entered, a fresh copy is made of all local variables declared in that body, initialized to the previous values of the variables. Any reference to a variable within the loop body will use the most recently made copy. This code shows an example:

```
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
```

The code produces the output:

```
31    32    33
```

When the loop body is executed, it uses whichever copy of the variable is current. For example, the statement `Dim y = x` refers to the latest copy of `y` and the original copy of `x`. And when a lambda is created, it remembers whichever copy of a variable was current at the time it was created. Therefore, each lambda uses the same shared copy of `x`, but a different copy of `y`. At the end of the program, when it executes the lambdas, that shared copy of `x` that they all refer to is now at its final value 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.

10.9.1 While...End While and Do...Loop Statements

A `While` or `Do` loop statement loops based on a Boolean expression.

```
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'
    ;
```

A `While` loop statement loops as long as the Boolean expression evaluates to true; a `Do` loop statement may contain a more complex condition. An expression may be placed after the `Do` keyword or after the `Loop` keyword, but not after both. The Boolean expression is evaluated as per Section §11.19. (Note: this does not require the expression to have Boolean type). It is also valid to specify no expression at all; in that case, the loop will never exit. If the expression is placed after `Do`, it will be evaluated before the loop block is executed on each iteration. If the expression is placed after `Loop`, it will be evaluated after the loop block has executed on each iteration. Placing the expression after `Loop` will therefore generate one more loop than placement after `Do`. The following example demonstrates this behavior:

```
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
```

The code produces the output:

```
Second Loop
```

In the case of the first loop, the condition is evaluated before the loop executes. In the case of the second loop, the condition is executed after the loop executes. The conditional expression must be prefixed with either a `While` keyword or an `Until` keyword. The former breaks the loop if the condition evaluates to false, the latter when the condition evaluates to true.

Note. `Until` is not a reserved word.

10.9.2 For...Next Statements

A `For...Next` statement loops based on a set of bounds. A `For` statement specifies a loop control variable, a lower bound expression, an upper bound expression, and an optional step value expression. The loop control variable is specified either through an identifier followed by an optional `As` clause or an expression.

```
ForStatement:
    | 'For' LoopControlVariable Equals Expression 'To' Expression
      ( 'Step' Expression  )? StatementTerminator
      Block?
      ( 'Next' NextExpressionList? StatementTerminator  )?
    ;

LoopControlVariable:
    | Identifier ( IdentifierModifiers 'As' TypeName  )?
    | Expression
    ;

NextExpressionList:
    | Expression ( Comma Expression  )*
    ;
```

As per the following rules, the loop control variable refers either to a new local variable specific to this `For...Next` statement, or to a pre-existing variable, or to an expression.

- If the loop control variable is an identifier with an `As` clause, the identifier defines a new local variable of the type specified in the `As` clause, scoped to the entire `For` loop.
- If the loop control variable is an identifier without an `As` clause, then the identifier is first resolved using the simple name resolution rules (see Section §11.4.4), excepting that this occurrence of the identifier would not in and of itself cause an implicit local variable to be created (Section §10.2.1).
    - If this resolution succeeds and the result is classified as a variable, then the loop control variable is that pre-existing variable.
    - If resolution fails, or if resolution succeeds and the result is classified as a type, then:
        - if local variable type inference is being used, the identifier defines a new local variable whose type is inferred from the bound and step expressions, scoped to the entire `For` loop;
        - if local variable type inference is not being used but implicit local declaration is, then an implicit local variable is created whose scope is the entire method (Section §10.2.1), and the loop control variable refers to this pre-existing variable;
        - if neither local variable type inference nor implicit local declarations are used, it is an error.
    - If resolution succeeds with something classified as neither a type nor a variable, it is an error.
- If the loop control variable is an expression, the expression must be classified as a variable.

A loop control variable cannot be used by another enclosing `For...Next` statement. The type of the loop control variable of a `For` statement determines the type of the iteration and must be one of:

- `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single`, `Double`
- An enumerated type
- `Object`
- A type `T` that has the following operators, where `B` is a type that can be used in a Boolean expression:

```
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
```

The bound and step expressions must be implicitly convertible to the type of the loop control variable and must be classified as values. At compile time, the type of the loop control variable is inferred by choosing the widest type among the lower bound, upper bound, and step expression types. If there is no widening conversion between two of the types, then a compile-time error occurs.

At run time, if the type of the loop control variable is `Object`, then the type of the iteration is inferred the same as at compile time, with two exceptions. First, if the bound and step expressions are all of integral types but have no widest type, then the widest type that encompasses all three types will be inferred. And second, if the type of the loop control variable is inferred to be `String`, `Double` will be inferred instead. If, at run time, no loop control type can be determined or if any of the expressions cannot be converted to the loop control type, a `System.InvalidCastException` will occur. Once a loop control type has been chosen at the beginning of the loop, the same type will be used throughout the iteration, regardless of changes made to the value in the loop control variable.

A `For` statement must be closed by a matching `Next` statement. A `Next` statement without a variable matches the innermost open `For` statement, while a `Next` statement with one or more loop control variables will, from left to right, match the `For` loops that match each variable. If a variable matches a `For` loop that is not the most nested loop at that point, a compile-time error results.

At the beginning of the loop, the three expressions are evaluated in textual order and the lower bound expression is assigned to the loop control variable. If the step value is omitted, it is implicitly the literal `1`, converted to the type of the loop control variable. The three expressions are only ever evaluated at the beginning of the loop.

At the beginning of each loop, the control variable is compared to see if it is greater than the end point if the step expression is positive, or less than the end point if the step expression is negative. If it is, the `For` loop terminates; otherwise the loop block executes. If the loop control variable is not a primitive type, the comparison operator is determined by whether the expression `step >= step - step` is true or false. At the `Next` statement, the step value is added to the control variable and execution returns to the top of the loop.

Note that a new copy of the loop control variable is *not* created on each iteration of the loop block. In this respect, the `For` statement differs from `For Each` (Section §10.9.3).

It is not valid to branch into a `For` loop from outside the loop.

10.9.3 For Each...Next Statements

A `For Each...Next` statement loops based on the elements in an expression. A `For Each` statement specifies a loop control variable and an enumerator expression. The loop control variable is specified either through an identifier followed by an optional `As` clause or an expression.

```
ForEachStatement:
    | 'For' 'Each' LoopControlVariable 'In' LineTerminator? Expression StatementTerminator
      Block?
      ( 'Next' NextExpressionList? StatementTerminator  )?
    ;
```

Following the same rules as `For...Next` statements (Section §10.9.2), the loop control variable refers either to a new local variable specific to this For Each...Next statement, or to a pre-existing variable, or to an expression.

The enumerator expression must be classified as a value and its type must be a collection type or `Object`. If the type of the enumerator expression is `Object`, then all processing is deferred until run-time. Otherwise, a conversion must exist from the element type of the collection to the type of the loop control variable.

The loop control variable cannot be used by another enclosing `For Each` statement. A `For Each` statement must be closed by a matching `Next` statement. A `Next` statement without a loop control variable matches the innermost open `For Each`. A `Next` statement with one or more loop control variables will, from left to right, match the `For Each` loops that have the same loop control variable. If a variable matches a `For Each` loop that is not the most nested loop at that point, a compile-time error occurs.

A type `C` is said to be a *collection type* if one of:

- All of the following are true:
  - `C` contains an accessible instance, shared or extension method with the signature `GetEnumerator()` that returns a type `E`.
  - `E` contains an accessible instance, shared or extension method with the signature `MoveNext()` and the return type `Boolean`.
  - `E` contains an accessible instance or shared property named `Current` that has a getter. The type of this property is the element type of the collection type.
- It implements the interface `System.Collections.Generic.IEnumerable(Of T)`, in which case the element type of the collection is considered to be `T`.
- It implements the interface `System.Collections.IEnumerable`, in which case the element type of the collection is considered to be `Object`.

Following is an example of a class that can be enumerated:

```
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
```

Before the loop begins, the enumerator expression is evaluated. If the type of the expression does not satisfy the design pattern, then the expression is cast to `System.Collections.IEnumerable` or `System.Collections.Generic.IEnumerable(Of T)`. If the expression type implements the generic interface, the generic interface is preferred at compile-time but the non-generic interface is preferred at run-time. If the expression type implements the generic interface multiple times, the statement is considered ambiguous and a compile-time error occurs.

Note. The non-generic interface is preferred in the late bound case, because picking the generic interface would mean that all the calls to the interface methods would involve type parameters. Since it is not possible to know the matching type arguments at run-time, all such calls would have to be made using late-bound calls. This would be slower than calling the non-generic interface because the non-generic interface could be called using compile-time calls.

`GetEnumerator` is called on the resulting value and the return value of the function is stored in a temporary. Then at the beginning of each iteration, `MoveNext` is called on the temporary. If it returns `False`, the loop terminates. Otherwise, each iteration of the loop is executed as follows:

1. If the loop control variable identified a new local variable (rather than a pre-existing one), then a fresh copy of this local variable is created. For the current iteration, all references within the loop block will refer to this copy.
2. The `Current` property is retrieved, coerced to the type of the loop control variable (regardless of whether the conversion is implicit or explicit), and assigned to the loop control variable.
3. The loop block executes.

Note. There is a slight change in behavior between version 10.0 and 11.0 of the language. Prior to 11.0, a fresh iteration variable was *not* created for each iteration of the loop. This difference is observable only if the iteration variable is captured by a lambda or a LINQ expression which is then invoked after the loop:

```
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()
```

Up to Visual Basic 10.0, this produced a warning at compile-time and printed "3" three times. That was because there was only a single variable "x" shared by all iterations of the loop, and all three lambdas captured the same "x", and by the time the lambdas were executed it then held the number 3. As of Visual Basic 11.0, it prints "1, 2, 3". That is because each lambda captures a different variable "x".

Note. The current element of the iteration is converted to the type of the loop control variable even if the conversion is explicit because there is no convenient place to introduce a conversion operator in the statement. This became particularly troublesome when working with the now-obsolete type `System.Collections.ArrayList`, because its element type is `Object`. This would have required casts in a great many loops, something we felt was not ideal. Ironically, generics enabled the creation of a strongly-typed collection, `System.Collections.Generic.List(Of T)`, which might have made us rethink this design point, but for compatibility's sake, this cannot be changed now.

When the `Next` statement is reached, execution returns to the top of the loop. If a variable is specified after the `Next` keyword, it must be the same as the first variable after the `For Each`. For example, consider the following code:

```
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
```

It is equivalent to the following code:

```
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
```

If the type `E` of the enumerator implements `System.IDisposable`, then the enumerator is disposed upon exiting the loop by calling the `Dispose` method. This ensures that resources held by the enumerator are released. If the method containing the `For Each` statement does not use unstructured error handling, then the `For Each` statement is wrapped in a `Try` statement with the `Dispose` method called in the `Finally` to ensure cleanup.

Note. The `System.Array` type is a collection type, and since all array types derive from `System.Array`, any array type expression is permitted in a `For Each` statement. For single-dimensional arrays, the `For Each` statement enumerates the array elements in increasing index order, starting with index 0 and ending with index Length - 1. For multidimensional arrays, the indices of the rightmost dimension are increased first.

For example, the following code prints `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
```

It is not valid to branch into a `For Each` statement block from outside the block.

10.10 Exception-Handling Statements

Visual Basic supports structured exception handling and unstructured exception handling. Only one style of exception handling may be used in a method, but the `Error` statement may be used in structured exception handling. If a method uses both styles of exception handling, a compile-time error results.

```
ErrorHandlingStatement:
    | StructuredErrorStatement
    | UnstructuredErrorStatement
    ;
```

10.10.1 Structured Exception-Handling Statements

Structured exception handling is a method of handling errors by declaring explicit blocks within which certain exceptions will be handled. Structured exception handling is done through a `Try` statement.

```
StructuredErrorStatement:
    | ThrowStatement
    | TryStatement
    ;

TryStatement:
    | 'Try' StatementTerminator
      Block?
      CatchStatement*
      FinallyStatement?
      'End' 'Try' StatementTerminator
    ;
```

For example:

```
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
```

A `Try` statement is made up of three kinds of blocks: try blocks, catch blocks, and finally blocks. A *try block* is a statement block that contains the statements to be executed. A *catch block* is a statement block that handles an exception. A *finally block* is a statement block that contains statements to be run when the `Try` statement is exited, regardless of whether an exception has occurred and been handled. A `Try` statement, which can only contain one try block and one finally block, must contain at least one catch block or finally block. It is invalid to explicitly transfer execution into a try block except from within a catch block in the same statement.

10.10.1.1 Finally Blocks

A `Finally` block is always executed when execution leaves any part of the `Try` statement. No explicit action is required to execute the `Finally` block; when execution leaves the `Try` statement, the system will automatically execute the `Finally` block and then transfer execution to its intended destination. The `Finally` block is executed

regardless of how execution leaves the `Try` statement: through the end of the `Try` block, through the end of a `Catch` block, through an `Exit Try` statement, through a `GoTo` statement, or by not handling a thrown exception.

Note that the `Await` expression in an async method, and the `Yield` statement in an iterator method, can cause flow of control to suspend in the async or iterator method instance and resume in some other method instance. However, this is merely a suspension of execution and does not involve exiting the respective async method or iterator method instance, and so does not cause `Finally` blocks to be executed.

It is invalid to explicitly transfer execution into a `Finally` block; it is also invalid to transfer execution out of a `Finally` block except through an exception.

```
FinallyStatement:
    | 'Finally' StatementTerminator
      Block?
    ;
```

10.10.1.2 Catch Blocks

If an exception occurs while processing the `Try` block, each `Catch` statement is examined in textual order to determine if it handles the exception.

```
CatchStatement:
    | 'Catch' ( Identifier ( 'As' NonArrayTypeName  )?  )?
      ( 'When' BooleanExpression  )? StatementTerminator
      Block?
    ;
```

The identifier specified in a `Catch` clause represents the exception that has been thrown. If the identifier contains an `As` clause, then the identifier is considered to be declared within the `Catch` block's local declaration space. Otherwise, the identifier must be a local variable (not a static variable) that was defined in a containing block.

A `Catch` clause with no identifier will catch all exceptions derived from `System.Exception`. A `Catch` clause with an identifier will only catch exceptions whose types are the same as or derived from the type of the identifier. The type must be `System.Exception`, or a type derived from `System.Exception`. When an exception is caught that derives from `System.Exception`, a reference to the exception object is stored in the object returned by the function `Microsoft.VisualBasic.Information.Err`.

A `Catch` clause with a `When` clause will only catch exceptions when the expression evaluates to `True`; the type of the expression must be a Boolean expression as per Section §11.19. A `When` clause is only applied after checking the type of the exception, and the expression may refer to the identifier representing the exception, as this example demonstrates:

```
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
```

This example prints:

```
Third handler
```

If a Catch clause handles the exception, execution transfers to the Catch block. At the end of the Catch block, execution transfers to the first statement following the Try statement. The Try statement will not handle any exceptions thrown in a Catch block. If no Catch clause handles the exception, execution transfers to a location determined by the system.

It is invalid to explicitly transfer execution into a Catch block.

The filters in When clauses are normally evaluated prior to the exception being thrown. For instance, the following code will print "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
```

However, Async and Iterator methods cause all finally blocks inside them to be executed prior to any filters outside. For instance, if the above code had `Async Sub Foo()`, then the output would be "Finally, Filter, Catch".

10.10.1.3 Throw Statement

The Throw statement raises an exception, which is represented by an instance of a type derived from System.Exception.

```
ThrowStatement:
    | 'Throw' Expression? StatementTerminator
    ;
```

If the expression is not classified as a value or is not a type derived from System.Exception, then a compile-time error occurs. If the expression evaluates to a null value at run time, then a System.NullReferenceException exception is raised instead.

A Throw statement may omit the expression within a catch block of a Try statement, as long as there is no intervening finally block. In that case, the statement rethrows the exception currently being handled within the catch block. For example:

```
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
```

10.10.2 Unstructured Exception-Handling Statements

Unstructured exception handling is a method of handling errors by indicating statements to branch to when an exception occurs. Unstructured exception handling is implemented using three statements: the `Error` statement, the `On Error` statement, and the `Resume` statement.

```
UnstructuredErrorStatement:
    | ErrorStatement
    | OnErrorStatement
    | ResumeStatement
    ;
```

For example:

```
    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
```

When a method uses unstructured exception handling, a single structured exception handler is established for the entire method that catches all exceptions. (Note that in constructors this handler does not extend over the call to the call to `New` at the beginning of the constructor.) The method then keeps track of the most recent exception-handler location and the most recent exception that has been thrown. At entry to the method, the exception-handler location and the exception are both set to `Nothing`. When an exception is thrown in a method that uses unstructured exception handling, a reference to the exception object is stored in the object returned by the function `Microsoft.VisualBasic.Information.Err`.

Unstructured error handling statements are not allowed in iterator or async methods.

10.10.2.1 Error Statement

An `Error` statement throws a `System.Exception` exception containing a Visual Basic 6 exception number. The expression must be classified as a value and its type must be implicitly convertible to `Integer`.

```
ErrorStatement:
    | 'Error' Expression StatementTerminator
    ;
```

10.10.2.2 On Error Statement

An `On Error` statement modifies the most recent exception-handling state.

```
OnErrorStatement:
    | 'On' 'Error' ErrorClause StatementTerminator
    ;

ErrorClause:
    | 'GoTo' '-' '1'
    | 'GoTo' '0'
    | GoToStatement
    | 'Resume' 'Next'
    ;
```

It may be used in one of four ways:

- `On Error GoTo -1` resets the most recent exception to `Nothing`.
- `On Error GoTo 0` resets the most recent exception-handler location to `Nothing`.
- `On Error GoTo LabelName` establishes the label as the most recent exception-handler location. This statement cannot be used in a method that contains a lambda or query expression.
- `On Error Resume Next` establishes the `Resume Next` behavior as the most recent exception-handler location.

10.10.2.3 Resume Statement

A `Resume` statement returns execution to the statement that caused the most recent exception.

```
ResumeStatement:
    | 'Resume' ResumeClause? StatementTerminator
    ;

ResumeClause:
    | 'Next'
    | LabelName
    ;
```

If the `Next` modifier is specified, execution returns to the statement that would have been executed after the statement that caused the most recent exception. If a label name is specified, execution returns to the label.

Because the `SyncLock` statement contains an implicit structured error-handling block, `Resume` and `Resume Next` have special behaviors for exceptions that occur in `SyncLock` statements. `Resume` returns execution to the beginning of the `SyncLock` statement, while `Resume Next` returns execution to the next statement following the `SyncLock` statement. For example, consider the following code:

```
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
```

It prints the following result.

```
Before exception
Before exception
After SyncLock
```

The first time through the `SyncLock` statement, `Resume` returns execution to the beginning of the `SyncLock` statement. The second time through the `SyncLock` statement, `Resume Next` returns execution to the end of the `SyncLock` statement. `Resume` and `Resume Next` are not allowed within a `SyncLock` statement.

In all cases, when a `Resume` statement is executed, the most recent exception is set to `Nothing`. If a `Resume` statement is executed with no most recent exception, the statement raises a `System.Exception` exception containing the Visual Basic error number `20` (Resume without error).

10.11 Branch Statements

Branch statements modify the flow of execution in a method. There are six branch statements:

1. A `GoTo` statement causes execution to transfer to the specified label in the method. It is not allowed to `GoTo` into a `Try`, `Using`, `SyncLock`, `With`, `For` or `For Each` block, nor into any loop block if a local variable of that block is captured in a lambda or LINQ expression.
2. An `Exit` statement transfers execution to the next statement after the end of the immediately containing block statement of the specified kind. If the block is the method block, then control flow exits the method as described in Section §10.1. If the `Exit` statement is not contained within the kind of block specified in the statement, a compile-time error occurs.
3. A `Continue` statement transfers execution to the end of the immediately containing block loop statement of the specified kind. If the `Continue` statement is not contained within the kind of block specified in the statement, a compile-time error occurs.
4. A `Stop` statement causes a debugger exception to occur.
5. An `End` statement terminates the program. Finalizers are run before shutdown, but the finally blocks of any currently executing `Try` statements are not executed. This statement may not be used in programs that are not executable (for example, DLLs).
6. A `Return` statement with no expression is equivalent to an `Exit Sub` or `Exit Function` statement. A `Return` statement with an expression is only allowed in a regular method that is a function, or in an async method that is a function with return type `Task(Of T)` for some `T`. The expression must be classified as a value which is implicitly convertible to the *function return variable* (in the case of regular methods) or to the *task return variable* (in the case of async methods). Its behavior is to evaluate its expression, then store it in the return variable, then execute an implicit `Exit Function` statement.

```
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
    ;
```

10.12 Array-Handling Statements

Two statements simplify working with arrays: `ReDim` statements and `Erase` statements.

```
ArrayHandlingStatement:
    | RedimStatement
    | EraseStatement
    ;
```

10.12.1 ReDim Statement

A `ReDim` statement instantiates new arrays.

```
RedimStatement:
    | 'ReDim' 'Preserve'? RedimClauses StatementTerminator
    ;

RedimClauses:
    | RedimClause ( Comma RedimClause  )*
    ;

RedimClause:
    | Expression ArraySizeInitializationModifier
    ;
```

Each clause in the statement must be classified as a variable or a property access whose type is an array type or `Object`, and be followed by a list of array bounds. The number of the bounds must be consistent with the type of the variable; any number of bounds is allowed for `Object`. At run time, an array is instantiated for each expression from left to right with the specified bounds and then assigned to the variable or property. If the variable type is `Object`, the number of dimensions is the number of dimensions specified, and the array element type is `Object`. If the given number of dimensions is incompatible with the variable or property at run time a compile-time error occurs. For example:

```
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
```

If the `Preserve` keyword is specified, then the expressions must also be classifiable as a value, and the new size for each dimension except for the rightmost one must be the same as the size of the existing array. The values in the existing array are copied into the new array: if the new array is smaller, the existing values are discarded; if the new array is bigger, the extra elements will be initialized to the default value of the element type of the array. For example, consider the following code:

```
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
```

It prints the following result:

```
3, 0
```

If the existing array reference is a null value at run time, no error is given. Other than the rightmost dimension, if the size of a dimension changes, a `System.ArrayTypeMismatchException` will be thrown.

Note. `Preserve` is not a reserved word.

10.12.2 Erase Statement

An `Erase` statement sets each of the array variables or properties specified in the statement to `Nothing`. Each expression in the statement must be classified as a variable or property access whose type is an array type or `Object`. For example:

```
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 ( Comma Expression  )*
    ;
```

10.13 Using statement

Instances of types are automatically released by the garbage collector when a collection is run and no live references to the instance are found. If a type holds on a particularly valuable and scarce resource (such as database connections or file handles), it may not be desirable to wait until the next garbage collection to clean up a particular instance of the type that is no longer in use. To provide a lightweight way of releasing resources before a collection, a type may implement the `System.IDisposable` interface. A type that does so exposes a `Dispose` method that can be called to force valuable resources to be released immediately, as such:

```
    Module Test
        Sub Main()
            Dim x As DBConnection = New DBConnection("...")

            ' Do some work
            ...

            x.Dispose()        ' Free the connection
        End Sub
    End Module
```

The `Using` statement automates the process of acquiring a resource, executing a set of statements, and then disposing of the resource. The statement can take two forms: in one, the resource is a local variable declared as a part of the statement and treated as a regular local variable declaration statement; in the other, the resource is the result of an expression.

```
UsingStatement:
    | 'Using' UsingResources StatementTerminator
      Block?
      'End' 'Using' StatementTerminator
    ;

UsingResources:
    | VariableDeclarators
    | Expression
    ;
```

If the resource is a local variable declaration statement then the type of the local variable declaration must be a type that can be implicitly converted to `System.IDisposable`. The declared local variables are read-only, scoped to the `Using` statement block and must include an initializer. If the resource is the result of an expression then the expression must be classified as a value and must be of a type that can be implicitly converted to `System.IDisposable`. The expression is only evaluated once, at the beginning of the statement.

The `Using` block is implicitly contained by a `Try` statement whose finally block calls the method `IDisposable.Dispose` on the resource. This ensures the resource is disposed even when an exception is thrown. As a result, it is invalid to branch into a `Using` block from outside of the block, and a `Using` block is treated as a single statement for the purposes of `Resume` and `Resume Next`. If the resource is `Nothing`, then no call to `Dispose` is made. Thus, the example:

```
Using f As C = New C()
    ...
End Using
```

is equivalent to:

```
Dim f As C = New C()
Try
    ...
Finally
    If f IsNot Nothing Then
        f.Dispose()
    End If
End Try
```

A `Using` statement that has a local variable declaration statement may acquire multiple resources at a time, which is equivalent to nested `Using` statements. For example, a `Using` statement of the form:

```
Using r1 As R = New R(), r2 As R = New R()
    r1.F()
    r2.F()
End Using
```

is equivalent to:

```
Using r1 As R = New R()
    Using r2 As R = New R()
        r1.F()
        r2.F()
    End Using
End Using
```

10.14 Await Statement

An await statement has the same syntax as an await operator expression (Section §11.25), is allowed only in methods that also allow await expressions, and has the same behavior as an await operator expression.

However, it may be classified as either a value or void. Any value resulting from evaluation of the await operator expression is discarded.

```
AwaitStatement:
    | AwaitOperatorExpression StatementTerminator
    ;
```

10.15 Yield Statement

Yield statements are related to iterator methods, which are described in Section §10.1.2.

```
YieldStatement:
    | 'Yield' Expression StatementTerminator
    ;
```

`Yield` is a reserved word if the immediately enclosing method or lambda expression in which it appears has an `Iterator` modifier, and if the `Yield` appears after that `Iterator` modifier; it is unreserved elsewhere. It is also

unreserved in preprocessor directives. The yield statement is only allowed in the body of a method or lambda expression where it is a reserved word. Within the immediately enclosing method or lambda, the yield statement may not occur inside the body of a `Catch` or `Finally` block, nor inside the body of a `SyncLock` statement.

The yield statement takes a single expression which must be classified as a value and whose type is implicitly convertible to the type of the *iterator current variable* (Section §10.1.2) of its enclosing iterator method.

Control flow only ever reaches a `Yield` statement when the `MoveNext` method is invoked on an iterator object. (This is because an iterator method instance only ever executes its statements due to the `MoveNext` or `Dispose` methods being called on an iterator object; and the `Dispose` method will only ever execute code in `Finally` blocks, where `Yield` is not allowed).

When a `Yield` statement is executed, its expression is evaluated and stored in the *iterator current variable* of the iterator method instance associated with that iterator object. The value `True` is returned to the invoker of `MoveNext`, and the control point of this instance stops advancing until the next invocation of `MoveNext` on the iterator object.

11. Expressions

An expression is a sequence of operators and operands that specifies a computation of a value, or that designates a variable or constant. This chapter defines the syntax, order of evaluation of operands and operators, and meaning of expressions.

```
Expression:
    | SimpleExpression
    | TypeExpression
    | MemberAccessExpression
    | DictionaryAccessExpression
    | InvocationExpression
    | IndexExpression
    | NewExpression
    | CastExpression
    | OperatorExpression
    | ConditionalExpression
    | LambdaExpression
    | QueryExpression
    | XMLLiteralExpression
    | XMLMemberAccessExpression
    ;
```

11.1 Expression Classifications

Every expression is classified as one of the following:

- *A value.* Every value has an associated type.
- *A variable.* Every variable has an associated type, namely the declared type of the variable.
- *A namespace.* An expression with this classification can only appear as the left side of a member access. In any other context, an expression classified as a namespace causes a compile-time error.
- *A type.* An expression with this classification can only appear as the left side of a member access. In any other context, an expression classified as a type causes a compile-time error.
- *A method group,* which is a set of methods overloaded on the same name. A method group may have an associated target expression and an associated type argument list.
- *A method pointer,* which represents the location of a method. A method pointer may have an associated target expression and an associated type argument list.
- *A lambda method,* which is an anonymous method.
- *A property group,* which is a set of properties overloaded on the same name. A property group may have an associated target expression.
- *A property access.* Every property access has an associated type, namely the type of the property. A property access may have an associated target expression.
- *A late-bound access,* which represents a method or property access deferred until run-time. A late-bound access may have an associated target expression and an associated type argument list. The type of a late-bound access is always `Object`.

- *An event access.* Every event access has an associated type, namely the type of the event. An event access may have an associated target expression. An event access may appear as the first argument of the `RaiseEvent`, `AddHandler`, and `RemoveHandler` statements. In any other context, an expression classified as an event access causes a compile-time error.
- *An array literal,* which represents the initial values of an array whose type has not yet been determined.
- *Void.* This occurs when the expression is an invocation of a subroutine, or an await operator expression with no result. An expression classified as void is only valid in the context of an invocation statement or an await statement.
- *A default value.* Only the literal `Nothing` produces this classification.

The final result of an expression is usually a value or a variable, with the other categories of expressions functioning as intermediate values that are only permitted in certain contexts.

Note that expressions whose type is a type parameter can be used in statements and expressions that require the type of an expression to have certain characteristics (such as being a reference type, value type, deriving from some type, etc.) if the constraints imposed on the type parameter satisfy those characteristics.

11.1.1 Expression Reclassification

Normally, when an expression is used in a context that requires a classification different from that of the expression, a compile-time error occurs -- for example, attempting to assign a value to a literal. However, in many cases it is possible to change an expression's classification through the process of *reclassification*.

If reclassification succeeds, then the reclassification is judged as widening or narrowing. Unless otherwise noted, all the reclassifications in this list are widening.

The following types of expressions can be reclassified:

- A variable can be reclassified as a value. The value stored in the variable is fetched.
- A method group can be reclassified as a value. The method group expression is interpreted as an invocation expression with the associated target expression and type parameter list, and empty parentheses (that is, `f` is interpreted as `f()` and `f(Of Integer)` is interpreted as `f(Of Integer)()`). This reclassification may result in the expression being further reclassified as void.
- A method pointer can be reclassified as a value. This reclassification can only occur in the context of a conversion where the target type is known. The method pointer expression is interpreted as the argument to a delegate instantiation expression of the appropriate type with the associated type argument list. For example:

```
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
```

- A lambda method can be reclassified as a value. If the reclassification occurs in the context of a conversion where the target type is known, then one of two reclassifications can occur:

a. If the target type is a delegate type, the lambda method is interpreted as the argument to a delegate-construction expression of the appropriate type.

b. If the target type is `System.Linq.Expressions.Expression(Of T)`, and `T` is a delegate type, then the lambda method is interpreted as if it was being used in delegate-construction expression for `T` and then converted to an expression tree.

   An async or iterator lambda method may only be interpreted as the argument to a delegate-construction expression, if the delegate has no ByRef parameters.

   If conversion from any of the delegate's parameter types to the corresponding lambda parameter types is a narrowing conversion, then the reclassification is judged as narrowing; otherwise it is widening.

   Note. The exact translation between lambda methods and expression trees may not be fixed between versions of the compiler and is beyond the scope of this specification. For Microsoft Visual Basic 11.0, all lambda expressions may be converted to expression trees subject to the following restrictions: (1) 1. Only single-line lambda expressions without ByRef parameters may be converted to expression trees. Of the single-line `Sub` lambdas, only invocation statements may be converted to expression trees. (2) Anonymous type expressions cannot be converted to expression trees if an earlier field initializer is used to initialize a subsequent field initializer, e.g. `New With {.a=1, .b=.a}`. (3) Object initializer expressions cannot be converted to expression trees if a member of the current object being initialized is used in one of the field initializers, e.g. `New C1 With {.a=1, .b=.Method1()}`. (4) Multi-dimensional array creation expressions can only be converted to expression trees if they declare their element type explicitly. (5) Late-binding expressions cannot be converted to expression trees. (6) When a variable or field is passed ByRef to an invocation expression but does not have exactly the same type as the ByRef parameter, or when a property is passed ByRef, normal VB semantics are that a copy of the argument is passed ByRef and its final value is then copied back into the variable or field or property. In expression trees, the copy-back does not happen. (7) All these restrictions apply to nested lambda expressions as well.

   If the target type is not known, then the lambda method is interpreted as the argument to a delegate instantiation expression of an anonymous delegate type with the same signature of the lambda method. If strict semantics are being used and the type of any of the parameters are omitted, a compile-time error occurs; otherwise, `Object` is substituted for any missing parameter type. For example:

```
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
```

- A property group can be reclassified as a property access. The property group expression is interpreted as an index expression with empty parentheses (that is, `f` is interpreted as `f()`).
- A property access can be reclassified as a value. The property access expression is interpreted as an invocation expression of the `Get` accessor of the property. If the property has no getter, then a compile-time error occurs.
- A late-bound access can be reclassified as a late-bound method or late-bound property access. In a situation where a late-bound access can be reclassified both as a method access and as a property access, reclassification to a property access is preferred.
- A late-bound access can be reclassified as a value.
- An array literal can be reclassified as a value. The type of the value is determined as follows:

a. If the reclassification occurs in the context of a conversion where the target type is known and the target type is an array type, then the array literal is reclassified as a value of type T(). If the target type is `System.Collections.Generic.IList(Of T)`, `IReadOnlyList(Of T)`, `ICollection(Of T)`, `IReadOnlyCollection(Of T)`, or `IEnumerable(Of T)`, and the array literal has one level of nesting, then the array literal is reclassified as a value of type `T()`.

b. Otherwise, the array literal is reclassified to a value whose type is an array of rank equal to the level of nesting is used, with element type determined by the dominant type of the elements in the initializer; if no dominant type can be determined, `Object` is used. For example:

```
' 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()
```

Note. There is a slight change in behavior between version 9.0 and version 10.0 of the language. Prior to 10.0, array element initializers did not affect local variable type inference and now they do. So `Dim a() = { 1, 2, 3 }` would have inferred `Object()` as the type of `a` in version 9.0 of the language and `Integer()` in version 10.0.

The reclassification then reinterprets the array literal as an array-creation expression. So the examples:

```
Dim x As Double = { 1, 2, 3, 4 }
Dim y = { "a", "b" }
```

are equivalent to:

```
Dim x As Double = New Double() { 1, 2, 3, 4 }
Dim y = New String() { "a", "b" }
```

The reclassification is judged as narrowing if any conversion from an element expression to the array element type is narrowing; otherwise it is judged as widening.

- The default value `Nothing` can be reclassified as a value. In a context where the target type is known, the result is the default value of the target type. In a context where the target type is not known, the result is a null value of type `Object`.

A namespace expression, type expression, event access expression, or void expression cannot be reclassified. Multiple reclassifications can be done at the same time. For example:

```
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
```

In this case, the property group expression `P` is first reclassified from a property group to a property access and then reclassified from a property access to a value. The fewest number of reclassifications are performed to reach a valid classification in the context.

11.2 Constant Expressions

A *constant expression* is an expression whose value can be fully evaluated at compile time.

```
ConstantExpression:
    | Expression
    ;
```

The type of a constant expression can be `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Char`, `Single`, `Double`, `Decimal`, `Date`, `Boolean`, `String`, `Object`, or any enumeration type. The following constructs are permitted in constant expressions:

- Literals (including `Nothing`).
- References to constant type members or constant locals.
- References to members of enumeration types.
- Parenthesized subexpressions.
- Coercion expressions, provided the target type is one of the types listed above. Coercions to and from `String` are an exception to this rule and are only allowed on null values because `String` conversions are always done in the current culture of the execution environment at run time. Note that constant coercion expressions can only ever use intrinsic conversions.
- The `+`, `-` and `Not` unary operators, provided the operand and result is of a type listed above.
- The `+`, `-`, `*`, `^`, `Mod`, `/`, `\`, `<<`, `>>`, `&`, `And`, `Or`, `Xor`, `AndAlso`, `OrElse`, `=`, `<`, `>`, `<>`, `<=`, and `=>` binary operators, provided each operand and result is of a type listed above.
- The conditional operator If, provided each operand and result is of a type listed above.
- The following run-time functions: `Microsoft.VisualBasic.Strings.ChrW`; `Microsoft.VisualBasic.Strings.Chr` if the constant value is between 0 and 128; `Microsoft.VisualBasic.Strings.AscW` if the constant string is not empty; `Microsoft.VisualBasic.Strings.Asc` if the constant string is not empty.

The following constructs are *not* permitted in constant expressions:

- Implicit binding through a `With` context.

Constant expressions of an integral type (`ULong`, `Long`, `UInteger`, `Integer`, `UShort`, `Short`, `SByte`, or `Byte`) can be implicitly converted to a narrower integral type, and constant expressions of type `Double` can be implicitly converted to `Single`, provided the value of the constant expression is within the range of the destination type. These narrowing conversions are allowed regardless of whether permissive or strict semantics are being used.

11.3 Late-Bound Expressions

When the target of a member access expression or index expression is of type `Object`, the processing of the expression may be deferred until run time. Deferring processing this way is called *late binding*. Late binding allows `Object` variables to be used in a *typeless* way, where all resolution of members is based on the actual run-time type of the value in the variable. If strict semantics are specified by the compilation environment or by `Option Strict`, late binding causes a compile-time error. Non-public members are ignored when doing late-binding, including for the purposes of overload resolution. Note that, unlike the early-bound case, invoking or accessing a `Shared` member

late-bound will cause the invocation target to be evaluated at run time. If the expression is an invocation expression for a member defined on `System.Object`, late binding will not take place.

In general, late-bound accesses are resolved at run time by looking up the identifier on the actual run-time type of the expression. If late-bound member lookup fails at run time, a `System.MissingMemberException` exception is thrown. Because late-bound member lookup is done solely off the run-time type of the associated target expression, an object's run-time type is never an interface. Therefore, it is impossible to access interface members in a late-bound member access expression.

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
```

This code displays:

```
Early-bound: xy
Late-bound: yx
```

Because late-bound overload resolution is done on the run-time type of the arguments, it is possible that an expression might produce different results based on whether it is evaluated at compile time or run time. The following example illustrates this difference:

```
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
```

This code displays:

```
    F(Base)
    F(Derived)
```

11.4 Simple Expressions

Simple expressions are literals, parenthesized expressions, instance expressions, or simple name expressions.

```
SimpleExpression:
    | LiteralExpression
    | ParenthesizedExpression
    | InstanceExpression
    | SimpleNameExpression
    | AddressOfExpression
    ;
```

11.4.1 Literal Expressions

Literal expressions evaluate to the value represented by the literal. A literal expression is classified as a value, except for the literal `Nothing`, which is classified as a default value.

```
LiteralExpression:
    | Literal
    ;
```

11.4.2 Parenthesized Expressions

A parenthesized expression consists of an expression enclosed in parentheses. A parenthesized expression is classified as a value, and the enclosed expression must be classified as a value. A parenthesized expression evaluates to the value of the expression within the parentheses.

```
ParenthesizedExpression:
    | OpenParenthesis Expression CloseParenthesis
    ;
```

11.4.3 Instance Expressions

An *instance expression* is the keyword `Me`. It may only be used within the body of a non-shared method, constructor, or property accessor. It is classified as a value. The keyword `Me` represents the instance of the type containing the method or property accessor being executed. If a constructor explicitly invokes another constructor (Section §9.3), `Me` cannot be used until after that constructor call, because the instance has not yet been constructed.

```
InstanceExpression:
    | 'Me'
    ;
```

11.4.4 Simple Name Expressions

A *simple name expression* consists of a single identifier followed by an optional type argument list.

```
SimpleNameExpression:
    | Identifier ( OpenParenthesis 'Of' TypeArgumentList CloseParenthesis  )?
    ;
```

The name is resolved and classified by the following "simple name resolution rules":

1. Starting with the immediately enclosing block and continuing with each enclosing outer block (if any), if the identifier matches the name of a local variable, static variable, constant local, method type parameter, or parameter, then the identifier refers to the matching entity.

   If the identifier matches a local variable, static variable, or constant local and a type argument list was provided, a compile-time error occurs. If the identifier matches a method type parameter and a type argument list was provided, no match occurs and resolution continues. If the identifier matches a local variable, the local variable matched is the implicit function or `Get` accessor return local variable, and the expression is part of an invocation expression, invocation statement, or an `AddressOf` expression, then no match occurs and resolution continues.

   The expression is classified as a variable if it is a local variable, static variable, or parameter. The expression is classified as a type if it is a method type parameter. The expression is classified as a value if it is a constant local.

2. For each nested type containing the expression, starting from the innermost and going to the outermost, if a lookup of the identifier in the type produces a match with an accessible member:

   a. If the matching type member is a type parameter, then the result is classified as a type and is the matching type parameter. If a type argument list was provided, no match occurs and resolution continues.

   b. Otherwise, if the type is the immediately enclosing type and the lookup identifies a non-shared type member, then the result is the same as a member access of the form `Me.E(Of A)`, where `E` is the identifier and `A` is the type argument list, if any.

   c. Otherwise, the result is exactly the same as a member access of the form `T.E(Of A)`, where `T` is the type containing the matching member, `E` is the identifier, and `A` is the type argument list, if any. In this case, it is an error for the identifier to refer to a non-shared member.

3. For each nested namespace, starting from the innermost and going to the outermost namespace, do the following:

   a. If the namespace contains an accessible type with the given name and has the same number of type parameters as was supplied in the type argument list, if any, then the identifier refers to that type and is classified as a type.

   b. Otherwise, if no type argument list was supplied and the namespace contains a namespace member with the given name, then the identifier refers to that namespace and is classified as a namespace.

   c. Otherwise, if the namespace contains one or more accessible standard modules, and a member name lookup of the identifier produces an accessible match in exactly one standard module, then the result is exactly the same as a member access of the form `M.E(Of A)`, where `M` is the standard module containing the matching member, `E` is the identifier, and `A` is the type argument list, if any. If the identifier matches accessible type members in more than one standard module, a compile-time error occurs.

4. If the source file has one or more import aliases, and the identifier matches the name of one of them, then the identifier refers to that namespace or type. If a type argument list is supplied, a compile-time error occurs.

5. If the source file containing the name reference has one or more imports:

   a. If the identifier matches in exactly one import the name of an accessible type with the same number of type parameters as was supplied in the type argument list, if any, or a type member, then the identifier refers to that type or type member. If the identifier matches in more than one import the name of an accessible type with the same number of type parameters as was supplied in the type argument list, if any, or an accessible type member, a compile-time error occurs.

   b. Otherwise, if no type argument list was supplied and the identifier matches in exactly one import the name of a namespace with accessible types, then the identifier refers to that namespace. If no type argument list was supplied and the identifier matches in more than one import the name of a namespace with accessible types, a compile-time error occurs.

   c. Otherwise, if the imports contain one or more accessible standard modules, and a member name lookup of the identifier produces an accessible match in exactly one standard module, then the result is exactly the same as a member access of the form `M.E(Of A)`, where `M` is the standard module containing the matching member, `E` is the identifier, and `A` is the type argument list, if any. If the identifier matches accessible type members in more than one standard module, a compile-time error occurs.

6. If the compilation environment defines one or more import aliases, and the identifier matches the name of one of them, then the identifier refers to that namespace or type. If a type argument list is supplied, a compile-time error occurs.

7. If the compilation environment defines one or more imports:

   a. If the identifier matches in exactly one import the name of an accessible type with the same number of type parameters as was supplied in the type argument list, if any, or a type member, then the identifier refers to that type or type member. If the identifier matches in more than one import the name of an accessible type with the same number of type parameters as was supplied in the type argument list, if any, or a type member, a compile-time error occurs.

   b. Otherwise, if no type argument list was supplied and the identifier matches in exactly one import the name of a namespace with accessible types, then the identifier refers to that namespace. If no type argument list was supplied and the identifier matches in more than one import the name of a namespace with accessible types, a compile-time error occurs.

   c. Otherwise, if the imports contain one or more accessible standard modules, and a member name lookup of the identifier produces an accessible match in exactly one standard module, then the result is exactly the same as a member access of the form `M.E(Of A)`, where `M` is the standard module containing the matching member, `E` is the identifier, and `A` is the type argument list, if any. If the identifier matches accessible type members in more than one standard module, a compile-time error occurs.

8. Otherwise, the name given by the identifier is undefined.

A simple name expression that is undefined is a compile-time error.

Normally, a name can only occur once in a particular namespace. However, because namespaces can be declared across multiple .NET assemblies, it is possible to have a situation where two assemblies define a type with the same fully qualified name. In that case, a type declared in the current set of source files is preferred over a type declared in an external .NET assembly. Otherwise, the name is ambiguous and there is no way to disambiguate the name.

11.4.5 AddressOf Expressions

An `AddressOf` expression is used to produce a method pointer. The expression consists of the `AddressOf` keyword and an expression that must be classified as a method group or a late-bound access. The method group cannot refer to constructors.

The result is classified as a method pointer, with the same associated target expression and type argument list (if any) as the method group.

```
AddressOfExpression:
    | 'AddressOf' Expression
    ;
```

11.5 Type Expressions

A *type expression* is a `GetType` expression, a `TypeOf...Is` expression, an `Is` expression, or a `GetXmlNamespace` expression.

```
TypeExpression:
    | GetTypeExpression
    | TypeOfIsExpression
    | IsExpression
    | GetXmlNamespaceExpression
    ;
```

11.5.1 GetType Expressions

A `GetType` expression consists of the keyword `GetType` and the name of a type.

```
GetTypeExpression:
    | 'GetType' OpenParenthesis GetTypeTypeName CloseParenthesis
    ;

GetTypeTypeName:
    | TypeName
    | QualifiedOpenTypeName
    ;

QualifiedOpenTypeName:
    | Identifier TypeArityList? ( Period IdentifierOrKeyword TypeArityList? )*
    | 'Global' Period IdentifierOrKeyword TypeArityList?
      ( Period IdentifierOrKeyword TypeArityList? )*
    ;

TypeArityList:
    | OpenParenthesis 'Of' CommaList? CloseParenthesis
    ;

CommaList:
    | Comma Comma*
    ;
```

A `GetType` expression is classified as a value, and its value is the reflection (`System.Type`) class that represents its GetTypeTypeName. If the GetTypeTypeName is a type parameter, the expression will return the `System.Type` object that corresponds to the type argument supplied for the type parameter at run-time.

The GetTypeTypeName is special in two ways:

- It is allowed to be `System.Void`, the only place in the language where this type name may be referenced.
- It may be a constructed generic type with the type arguments omitted. This allows the `GetType` expression to return the `System.Type` object that corresponds to the generic type itself.

The following example demonstrates the `GetType` expression:

```
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
```

The resulting output is:

```
    Int32
    Int32
    String
    Double[]
```

11.5.2 TypeOf...Is Expressions

A `TypeOf...Is` expression is used to check whether the run-time type of a value is compatible with a given type. The first operand must be classified as a value, cannot be a reclassified lambda method, and must be of a reference type or an unconstrained type parameter type. The second operand must be a type name. The result of the expression is classified as a value and is a `Boolean` value. The expression evaluates to `True` if the run-time type of the operand has an identity, default, reference, array, value type, or type parameter conversion to the type, `False` otherwise. A compile-time error occurs if no conversion exists between the type of the expression and the specific type.

```
TypeOfIsExpression:
    | 'TypeOf' Expression 'Is' LineTerminator? TypeName
    ;
```

11.5.3 Is Expressions

An `Is` or `IsNot` expression is used to do a reference equality comparison.

```
IsExpression:
    | Expression 'Is' LineTerminator? Expression
    | Expression 'IsNot' LineTerminator? Expression
    ;
```

Each expression must be classified as a value and the type of each expression must be a reference type, an unconstrained type parameter type, or a nullable value type. If the type of one expression is an unconstrained type parameter type or nullable value type, however, the other expression must be the literal `Nothing`.

The result is classified as a value and is typed as `Boolean`. An `Is` operation evaluates to `True` if both values refer to the same instance or both values are `Nothing`, or `False` otherwise. An `IsNot` operation evaluates to `False` if both values refer to the same instance or both values are `Nothing`, or `True` otherwise.

11.5.4 GetXmlNamespace Expressions

A `GetXmlNamespace` expression consists of the keyword `GetXmlNamespace` and the name of an XML namespace declared by the source file or compilation environment.

```
GetXmlNamespaceExpression:
    | 'GetXmlNamespace' OpenParenthesis XMLNamespaceName? CloseParenthesis
    ;
```

An `GetXmlNamespace` expression is classified as a value, and its value is an instance of `System.Xml.Linq.XNamespace` that represents the XMLNamespaceName. If that type is not available, then a compile-time error will occur.

For example:

```
    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
```

Everything between the parentheses is considered part of the namespace name, so XML rules around things such as whitespace apply. For example:

```
    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
```

The XML namespace expression can also be omitted, in which case the expression returns the object that represents the default XML namespace.

11.6 Member Access Expressions

A member access expression is used to access a member of an entity.

```
MemberAccessExpression:
    | MemberAccessBase? Period IdentifierOrKeyword
      ( OpenParenthesis 'Of' TypeArgumentList CloseParenthesis  )?
    ;

MemberAccessBase:
    | Expression
    | NonArrayTypeName
    | 'Global'
    | 'MyClass'
    | 'MyBase'
    ;
```

A member access of the form `E.I(Of A)`, where `E` is an expression, a non-array type name, the keyword `Global`, or omitted and `I` is an identifier with an optional type argument list `A`, is evaluated and classified as follows:

1. If `E` is omitted, then the expression from the immediately containing `With` statement is substituted for `E` and the member access is performed. If there is no containing `With` statement, a compile-time error occurs.
2. If `E` is classified as a namespace or `E` is the keyword `Global`, then the member lookup is done in the context of the specified namespace. If `I` is the name of an accessible member of that namespace with the same number of type parameters as was supplied in the type argument list, if any, then the result is that member. The result is classified as a namespace or a type depending on the member. Otherwise, a compile-time error occurs.
3. If `E` is a type or an expression classified as a type, then the member lookup is done in the context of the specified type. If `I` is the name of an accessible member of `E`, then `E.I` is evaluated and classified as follows:
    a. If `I` is the keyword `New` and `E` is not an enumeration then a compile-time error occurs.

b. If `I` identifies a type with the same number of type parameters as was supplied in the type argument list, if any, then the result is that type.

c. If `I` identifies one or more methods, then the result is a method group with the associated type argument list and no associated target expression.

d. If `I` identifies one or more properties and no type argument list was supplied, then the result is a property group with no associated target expression.

e. If `I` identifies a shared variable and no type argument list was supplied, then the result is either a variable or a value. If the variable is read-only, and the reference occurs outside the shared constructor of the type in which the variable is declared, then the result is the value of the shared variable `I` in `E`. Otherwise, the result is the shared variable `I` in `E`.

f. If `I` identifies a shared event and no type argument list was supplied, the result is an event access with no associated target expression.

g. If `I` identifies a constant and no type argument list was supplied, then the result is the value of that constant.

h. If `I` identifies an enumeration member and no type argument list was supplied, then the result is the value of that enumeration member.

i. Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.

4. If `E` is classified as a variable or value, the type of which is `T`, then the member lookup is done in the context of `T`. If `I` is the name of an accessible member of `T`, then `E.I` is evaluated and classified as follows:

a. If `I` is the keyword `New`, `E` is `Me`, `MyBase`, or `MyClass`, and no type arguments were supplied, then the result is a method group representing the instance constructors of the type of `E` with an associated target expression of `E` and no type argument list. Otherwise, a compile-time error occurs.

b. If `I` identifies one or more methods, including extension methods if `T` is not `Object`, then the result is a method group with the associated type argument list and an associated target expression of `E`.

c. If `I` identifies one or more properties and no type arguments were supplied, then the result is a property group with an associated target expression of `E`.

d. If `I` identifies a shared variable or an instance variable and no type arguments were supplied, then the result is either a variable or a value. If the variable is read-only, and the reference occurs outside a constructor of the class in which the variable is declared appropriate for the kind of variable (shared or instance), then the result is the value of the variable `I` in the object referenced by `E`. If `T` is a reference type, then the result is the variable `I` in the object referenced by `E`. Otherwise, if `T` is a value type and the expression `E` is classified as a variable, the result is a variable; otherwise the result is a value.

e. If `I` identifies an event and no type arguments were supplied, the result is an event access with an associated target expression of `E`.

f. If `I` identifies a constant and no type arguments were supplied, then the result is the value of that constant.

g. If `I` identifies an enumeration member and no type arguments were supplied, then the result is the value of that enumeration member.

h. If `T` is `Object`, then the result is a late-bound member lookup classified as a late-bound access with the associated type argument list and an associated target expression of `E`.

5. Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.

A member access of the form `MyClass.I(Of A)` is equivalent to `Me.I(Of A)`, but all members accessed on it are treated as if the members are non-overridable. Thus, the member accessed will not be affected by the run-time type of the value on which the member is being accessed.

A member access of the form `MyBase.I(Of A)` is equivalent to `CType(Me, T).I(Of A)` where `T` is the direct base type of the type containing the member access expression. All method invocations on it are treated as if the method being invoked is non-overridable. This form of member access is also called a *base access*.

The following example demonstrates how `Me`, `MyBase` and `MyClass` relate:

```
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
```

This code prints out:

```
MoreDerived.F
Derived.F
Derived.F
```

When a member access expression begins with the keyword `Global`, the keyword represents the outermost unnamed namespace, which is useful in situations where a declaration shadows an enclosing namespace. The `Global` keyword allows "escaping" out to the outermost namespace in that situation. For example:

```
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
```

In the above example, the first method call is invalid because the identifier `System` binds to the class `System`, not the namespace `System`. The only way to access the `System` namespace is to use `Global` to escape out to the outermost namespace.

If the member being accessed is shared, any expression on the left side of the period is superfluous and is not evaluated unless the member access is done late-bound. For example, consider the following code:

```
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
```

It prints `The value of F is: 10` because the function `ReturnC` does not need to be called to provide an instance of `C` to access the shared member `F`.

11.6.1 Identical Type and Member Names

It is not uncommon to name members using the same name as their type. In that situation, however, inconvenient name hiding can occur:

```
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
```

In the previous example, the simple name `Color` in `DefaultColor` binds to the instance property instead of the type. Because an instance member cannot be referenced in a shared member, this would normally be an error.

However, a special rule allows access to the type in this case. If the base expression of a member access expression is a simple name and binds to a constant, field, property, local variable or parameter whose type has the same name, then the base expression can refer either to the member or the type. This can never result in ambiguity because the members that can be accessed off of either one are the same.

11.6.2 Default Instances

In some situations, classes derived from a common base class usually or always have only a single instance. For example, most windows shown in a user interface only ever have one instance showing on the screen at any time. To simplify working with these types of classes, Visual Basic can automatically generate *default instances* of the classes that provide a single, easily referenced instance for each class.

Default instances are always created for a *family* of types rather than for one particular type. So instead of creating a default instance for a class Form1 that derives from Form, default instances are created for all classes derived from Form. This means that each individual class that derives from the base class does not have to be specially marked to have a default instance.

The default instance of a class is represented by a compiler-generated property that returns the default instance of that class. The property generated as a member of a class called the *group class* that manages allocating and destroying default instances for all classes derived from the particular base class. For example, all of the default instance properties of classes derived from `Form` may be collected in the `MyForms` class. If an instance of the group class is returned by the expression `My.Forms`, then the following code accesses the default instances of derived classes `Form1` and `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
```

Default instances will not be created until the first reference to them; fetching the property representing the default instance causes the default instance to be created if it has not already been created or has been set to `Nothing`. To allow testing for the existence of a default instance, when a default instance is the target of an `Is` or `IsNot` operator, the default instance will not be created. Thus, it is possible to test whether a default instance is `Nothing` or some other reference without causing the default instance to be created.

Default instances are intended to make it easy to refer to the default instance from outside of the class that has the default instance. Using a default instance from within a class that defines it might cause confusion as to which instance is being referred to, i.e. the default instance or the current instance. For example, the following code modifies only the value `x` in the default instance, even though it is being called from another instance. Thus the code would print the value `5` instead of `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
```

To prevent this kind of confusion, it is not valid to refer to a default instance from within an instance method of the default instance's type.

11.6.2.1 Default Instances and Type Names

A default instance may also be accessible directly through its type's name. In this case, in any expression context where the type name is not allowed the expression `E`, where `E` represents the fully qualified name of the class with a default instance, is changed to `E'`, where `E'` represents an expression that fetches the default instance property. For example, if default instances for classes derived from `Form` allow accessing the default instance through the type name, then the following code is equivalent to the code in the previous example:

```
Module Main
    Sub Main()
        Form1.x = 10
        Console.WriteLine(Form2.y)
    End Sub
End Module
```

This also means that a default instance that is accessible through its type's name is also assignable through the type name. For example, the following code sets the default instance of `Form1` to `Nothing`:

```
Module Main
    Sub Main()
        Form1 = Nothing
    End Sub
End Module
```

Note that the meaning of `E.I` were `E` represents a class and `I` represents a shared member does not change. Such an expression still accesses the shared member directly off of the class instance and does not reference the default instance.

11.6.2.2 Group Classes

The `Microsoft.VisualBasic.MyGroupCollectionAttribute` attribute indicates the group class for a family of default instances. The attribute has four parameters:

- The parameter `TypeToCollect` specifies the base class for the group. All instantiable classes without open type parameters that derive from a type with this name (regardless of type parameters) will automatically have a default instance.
- The parameter `CreateInstanceMethodName` specifies the method to call in the group class to create a new instance in a default instance property.
- The parameter `DisposeInstanceMethodName` specifies the method to call in the group class to dispose of a default instance property if the default instance property is assigned the value `Nothing`.
- The parameter `DefaultInstanceAlias` specifics the expression `E'` to substitute for the class name if the default instances are accessible directly through their type name. If this parameter is `Nothing` or an empty string, default instances on this group type are not accessible directly through their type's name. (**Note.** In all

current implementations of the Visual Basic language, the `DefaultInstanceAlias` parameter is ignored, except in compiler-provided code.)

Multiple types can be collected into the same group by separating the names of the types and methods in the first three parameters using commas. There must be the same number of items in each parameter, and the list elements are matched in order. For example, the following attribute declaration collects types that derive from `C1`, `C2` or `C3` into a single group:

```
<Microsoft.VisualBasic.MyGroupCollection("C1, C2, C3", _
    "CreateC1, CreateC2, CreateC3", _
    "DisposeC1, DisposeC2, DisposeC3", "My.Cs")>
Public NotInheritable Class MyCs
    ...
End Class
```

The signature of the create method must be of the form `Shared Function <Name>(Of T As {New, <Type>})(Instance Of T) As T`. The dispose method must be of the form `Shared Sub <Name>(Of T As <Type>)(ByRef Instance Of T)`. Thus, the group class for the example in the preceding section could be declared as follows:

```
<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
```

If a source file declared a derived class `Form1`, the generated group class would be equivalent to:

```
<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
```

11.6.3 Extension Method Collection

Extension methods for the member access expression `E.I` are collected by gathering all of the extension methods with the name `I` that are available in the current context:

1. First, each nested type containing the expression is checked, starting from the innermost and going to the outermost.
2. Then, each nested namespace is checked, starting from the innermost and going to the outermost namespace.
3. Then, the imports in the source file are checked.
4. Then, the imports defined by the compilation environment are checked.

An extension method is collected only if there is a widening native conversion from the target expression type to the type of the first parameter of the extension method. And unlike regular simple name expression binding, the search collects *all* extension methods; the collection does not stop when an extension method is found. For example:

```
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
```

In this example, even though `N2C1Extensions.M1` is found before `N1C1Extensions.M1`, they both are considered as extension methods. Once all of the extension methods have been collected, they are then *curried*. Currying takes the target of the extension method call and applies it to the extension method call, resulting in a new method signature with the first parameter removed (because it has been specified). For example:

```
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
```

In the above example, the curried result of applying `v` to `Ext1.M` is the method signature `Sub M(y As Integer)`.

In addition to removing the first parameter of the extension method, currying also removes any method type parameters that are a part of the type of the first parameter. When currying an extension method with method type parameter, type inference is applied to the first parameter and the result is fixed for any type parameters that are inferred. If type inference fails, the method is ignored. For example:

```
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
```

In the above example, the curried result of applying `v` to `Ext1.M` is the method signature `Sub M(Of U)(y As U)`, because the type parameter `T` is inferred as a result of the currying and is now fixed. Because the type parameter `U` was not inferred as a part of the currying, it remains an open parameter. Similarly, because the type parameter `T` is inferred as a result of applying `v` to `Ext2.M`, the type of parameter `y` becomes fixed as `Integer`. It will not be inferred to be any other type. When currying the signature, all constraints except for `New` constraints are also applied. If the constraints are not satisfied, or depend on a type that was not inferred as a part of currying, the extension method is ignored. For example:

```
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
```

Note. One of the main reasons for doing currying of extension methods is that it allows query expressions to infer the type of the iteration before evaluating the arguments to a query pattern method. Since most query pattern methods take lambda expressions, which require type inference themselves, this greatly simplifies the process of evaluating a query expression.

Unlike normal interface inheritance, extension methods that extend two interfaces that do not relate to one another are available, as long as they do not have the same curried signature:

```
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
```

Finally, it is important to remember that extension methods are not considered when doing late binding:

```
    Module Test
        Sub Main()
            Dim o As Object = ...

            ' Ignores extension methods
            o.M1()
        End Sub
    End Module
```

11.7 Dictionary Member Access Expressions

A *dictionary member access expression* is used to look up a member of a collection. A dictionary member access takes the form of `E!I`, where `E` is an expression that is classified as a value and `I` is an identifier.

```
DictionaryAccessExpression:
    | Expression? '!' IdentifierOrKeyword
    ;
```

The type of the expression must have a default property indexed by a single `String` parameter. The dictionary member access expression `E!I` is transformed into the expression `E.D("I")`, where `D` is the default property of `E`. For example:

```
    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
```

If an exclamation point is specified with no expression, the expression from the immediately containing `With` statement is assumed. If there is no containing `With` statement, a compile-time error occurs.

11.8 Invocation Expressions

An invocation expression consists of an invocation target and an optional argument list.

```
InvocationExpression:
    | Expression ( OpenParenthesis ArgumentList? CloseParenthesis  )?
    ;

ArgumentList:
    | PositionalArgumentList
    | PositionalArgumentList Comma NamedArgumentList
    | NamedArgumentList
    ;

PositionalArgumentList:
    | Expression? ( Comma Expression?  )*
    ;

NamedArgumentList:
    | IdentifierOrKeyword ColonEquals Expression
      ( Comma IdentifierOrKeyword ColonEquals Expression  )*
    ;
```

The target expression must be classified as a method group or a value whose type is a delegate type. If the target expression is a value whose type is a delegate type, then the target of the invocation expression becomes the method group for the `Invoke` member of the delegate type and the target expression becomes the associated target expression of the method group.

An argument list has two sections: positional arguments and named arguments. *Positional arguments* are expressions and must precede any named arguments. *Named arguments* start with an identifier that can match keywords, followed by `:=` and an expression.

If the method group only contains one accessible method, including both instance and extension methods, and that method takes no arguments and is a function, then the method group is interpreted as an invocation expression with an empty argument list and the result is used as the target of an invocation expression with the provided argument list(s). For example:

```
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
```

Otherwise, overload resolution is applied to the methods to pick the most applicable method for the given argument list(s). If the most applicable method is a function, then the result of the invocation expression is classified as a value typed as the return type of the function. If the most applicable method is a subroutine, then the result is classified as void. If the most applicable method is a partial method that has no body, then the invocation expression is ignored and the result is classified as void.

For an early-bound invocation expression, the arguments are evaluated in the order in which the corresponding parameters are declared in the target method. For a late-bound member access expression, they are evaluated in the order in which they appear in the member access expression: see Section §11.3.

11.8.1 Overloaded Method Resolution

In practice, the rules for determining overload resolution are intended to find the overload that is "closest" to the actual arguments supplied. If there is a method whose parameter types match the argument types, then that method is obviously the closest. Barring that, one method is closer than another if all of its parameter types are narrower than (or the same as) the parameter types of the other method. If neither method's parameters are narrower than the other, then there is no way for to determine which method is closer to the arguments.

Note. Overload resolution does not take into account the expected return type of the method.

Also note that because of the named parameter syntax, the ordering of the actual and formal parameters may not be the same.

Given a method group, the most applicable method in the group for an argument list is determined using the following steps. If, after applying a particular step, no members remain in the set, then a compile-time error occurs. If only one member remains in the set, then that member is the most applicable member. The steps are:

1. First, if no type arguments have been supplied, apply type inference to any methods which have type parameters. If type inference succeeds for a method, then the inferred type arguments are used for that particular method. If type inference fails for a method, then that method is eliminated from the set.
2. Next, eliminate all members from the set that are inaccessible or not applicable (Section §11.8.2) to the argument list
3. Next, if one or more arguments are `AddressOf` or lambda expressions, then calculate the *delegate relaxation levels* for each such argument as below. If the worst (lowest) delegate relaxation level in `N` is worse than the lowest delegate relaxation level in `M`, then eliminate `N` from the set. The delegate relaxation levels are as follows:
    a. *Error delegate relaxation level* -- if the `AddressOf` or lambda cannot be converted to the delegate type.
    b. *Narrowing delegate relaxation of return type or parameters* -- if the argument is `AddressOf` or a lambda with a declared type and the conversion from its return type to the delegate return type is narrowing; or if the argument is a regular lambda and the conversion from any of its return expressions to the delegate return type is narrowing, or if the argument is an async lambda and the delegate return type is `Task(Of T)` and the conversion from any of its return expressions to `T` is narrowing; or if the argument is an iterator lambda and the delegate return type `IEnumerator(Of T)` or `IEnumerable(Of T)` and the conversion from any of its yield operands to `T` is narrowing.
    c. *Widening delegate relaxation to delegate without signature* -- if delegate type is `System.Delegate` or `System.MultiCastDelegate` or `System.Object`.
    d. *Drop return or arguments delegate relaxation* -- if the argument is `AddressOf` or a lambda with a declared return type and the delegate type lacks a return type; or if the argument is a lambda with one or more

return expressions and the delegate type lacks a return type; or if the argument is `AddressOf` or lambda with no parameters and the delegate type has parameters.

e. *Widening delegate relaxation of return type* -- if the argument is `AddressOf` or a lambda with a declared return type, and there is a widening conversion from its return type to that of the delegate; or if the argument is a regular lambda where the conversion from all return expressions to the delegate return type is widening or identity with at least one widening; or if the argument is an async lambda and the delegate is `Task(Of T)` or `Task` and the conversion from all return expressions to `T`/`Object` respectively is widening or identity with at least one widening; or if the argument is an iterator lambda and the delegate is `IEnumerator(Of T)` or `IEnumerable(Of T)` or `IEnumerator` or `IEnumerable` and the conversion from all return expressions to `T`/`Object` is widening or identity with at least one widening.

f. *Identity delegate relaxation* -- if the argument is an `AddressOf` or a lambda which matches the delegate exactly, with no widening or narrowing or dropping of parameters or returns or yields.Next, if some members of the set do not requiring narrowing conversions to be applicable to any of the arguments, then eliminate all members that do. For example:

```
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
```

4. Next, elimination is done based on narrowing as follows. (Note that, if Option Strict is On, then all members that require narrowing have already been judged inapplicable (Section §11.8.2) and removed by Step 2.)

   a. If some instance members of the set only require narrowing conversions where the argument expression type is `Object`, then eliminate all other members.

   b. If the set contains more than one member which requires narrowing only from `Object`, then the invocation target expression is reclassified as a late-bound method access (and an error is given if the type containing the method group is an interface, or if any of the applicable members were extension members).

   c. If there are any candidates that only require narrowing from numeric literals, then chose the most specific among all remaining candidates by the steps below. If the winner requires only narrowing from numeric literals, then it is picked as the result of overload resolution; otherwise it is an error.

      Note. The justification for this rule is that if a program is loosely-typed (that is, most or all variables are declared as `Object`), overload resolution can be difficult when many conversions from `Object` are narrowing. Rather than have the overload resolution fail in many situations (requiring strong typing of the arguments to the method call), resolution the appropriate overloaded method to call is deferred until run time. This allows the loosely-typed call to succeed without additional casts. An unfortunate side-effect of this, however, is that performing the late-bound call requires casting the call target to `Object`. In the case of a structure value, this means that the value must be boxed to a temporary. If the method eventually called tries to change a field of the structure, this change will be lost once the method returns. Interfaces are excluded from this special rule because late binding always resolves against the members of the runtime class or structure type, which may have different names than the members of the interfaces they implement.

5. Next, if any instance methods remain in the set which do not require narrowing, then eliminate all extension methods from the set. For example:

```
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
```

Note. Extension methods are ignored if there are applicable instance methods to guarantee that adding an import (that might bring new extension methods into scope) will not cause a call on an existing instance method to rebind to an extension method. Given the broad scope of some extension methods (i.e. those defined on interfaces and/or type parameters), this is a safer approach to binding to extension methods.

6. Next, if, given any two members of the set `M` and `N`, `M` is more *specific* (Section §11.8.1.1) than `N` given the argument list, eliminate `N` from the set. If more than one member remains in the set and the remaining members are not equally specific given the argument list, a compile-time error results.

7. Otherwise, given any two members of the set, `M` and `N`, apply the following tie-breaking rules, in order:

   a. 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
```

The above example produces the following output:

```
F(Object, Object())
F(Object, Object, Object())
F(Object, Object, Object())
G(Object)
```

Note. When a class declares a method with a paramarray parameter, it is not uncommon to also include some of the expanded forms as regular methods. By doing so it is possible to avoid the allocation of an array instance that occurs when an expanded form of a method with a paramarray parameter is invoked.

b. If `M` is defined in a more derived type than `N`, eliminate `N` from the set. For example:

```
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
```

This rule also applies to the types that extension methods are defined on. For example:

```
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
```

c. If `M` and `N` are extension methods and the target type of `M` is a class or structure and the target type of `N` is an interface, eliminate `N` from the set. For example:

```
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
```

d. If `M` and `N` are extension methods, and the target type of `M` and `N` are identical after type parameter substitution, and the target type of `M` before type parameter substitution does not contain type parameters but the target type of `N` does, then has fewer type parameters than the target type of `N`, eliminate `N` from the set. For example:

```
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
```

e. Before type arguments have been substituted, if `M` is *less generic* (Section §11.8.1.2) than `N`, eliminate `N` from the set.

f. If `M` is not an extension method and `N` is, eliminate `N` from the set.

g. If `M` and `N` are extension methods and `M` was found before `N` (Section §11.6.3), eliminate `N` from the set. For example:

```
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
```

If the extension methods were found in the same step, then those extension methods are ambiguous. The call may always be disambiguated using the name of the standard module containing the extension method and calling the extension method as if it was a regular member. For example:

```
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
```

h. If `M` and `N` both required type inference to produce type arguments, and `M` did not require determining the dominant type for any of its type arguments (i.e. each the type arguments inferred to a single type), but `N` did, eliminate `N` from the set.

   Note. This rule ensures that overload resolution that succeeded in previous versions (where inferring multiple types for a type argument would cause an error), continue to produce the same results.

i. If overload resolution is being done to resolve the target of a delegate-creation expression from an `AddressOf` expression, and both the delegate and `M` are functions while `N` is a subroutine, eliminate `N` from the set. Likewise, if both the delegate and `M` are subroutines, while `N` is a function, eliminate `N` from the set.

j. If `M` did not use any optional parameter defaults in place of explicit arguments, but `N` did, then eliminate `N` from the set.

k. Before type arguments have been substituted, if `M` has *greater depth of genericity* (Section §11.8.1.2) than `N`, then eliminate `N` from the set.

8. Otherwise, the call is ambiguous and a compile-time error occurs.

11.8.1.1 Specificity of members/types given an argument list

A member `M` is considered *equally specific* as `N`, given an argument-list `A`, if their signatures are the same or if each parameter type in `M` is the same as the corresponding parameter type in `N`.

Note. Two members can end up in a method group with the same signature due to extension methods. Two members can also be equally specific but not have the same signature due to type parameters or paramarray expansion.

A member M is considered *more specific* than N if their signatures are different and at least one parameter type in M is more specific than a parameter type in N, and no parameter type in N is more specific than a parameter type in M. Given a pair of parameters Mj and Nj that matches an argument Aj, the type of Mj is considered *more specific* than the type of Nj if one of the following conditions is true:

- There exists a widening conversion from the type of Mj to the type Nj. (**Note.** Because parameters types are being compared without regard to the actual argument in this case, the widening conversion from constant expressions to a numeric type the value fits into is not considered in this case.)
- Aj is the literal 0, Mj is a numeric type and Nj is an enumerated type. (**Note.** This rule is necessary because the literal 0 widens to any enumerated type. Since an enumerated type widens to its underlying type, this means that overload resolution on 0 will, by default, prefer enumerated types over numeric types. We received a lot of feedback that this behavior was counterintuitive.)
- Mj and Nj are both numeric types, and Mj comes earlier than Nj in the list Byte, SByte, Short, UShort, Integer, UInteger, Long, ULong, Decimal, Single, Double. (**Note.** The rule about the numeric types is useful because the signed and unsigned numeric types of a particular size only have narrowing conversions between them. The above rule breaks the tie between the two types in favor of the more "natural" numeric type. This is particularly important when doing overload resolution on a type that widens to both the signed and unsigned numeric types of a particular size, for example, a numeric literal that fits into both.)
- Mj and Nj are delegate function types and the return type of Mj is more specific than the return type of Nj If Aj is classified as a lambda method, and Mj or Nj is System.Linq.Expressions.Expression(Of T), then the type argument of the type (assuming it is a delegate type) is substituted for the type being compared.
- Mj is identical to the type of Aj, and Nj is not. (**Note.** It is interesting to note that the previous rule differs slightly from C#, in that C# requires that the delegate function types have identical parameter lists before comparing return types, while Visual Basic does not.)

11.8.1.2 Genericity

A member M is determined to be *less generic* than a member N as follows:

1. If, for each pair of matching parameters Mj and Nj, Mj is less or equally generic than Nj with respect to type parameters on the method, and at least one Mj is less generic with respect to type parameters on the method.
2. Otherwise, if for each pair of matching parameters Mj and Nj, Mj is less or equally generic than Nj with respect to type parameters on the type, and at least one Mj is less generic with respect to type parameters on the type, then M is less generic than N.

A parameter M is considered to be equally generic to a parameter N if their types Mt and Nt both refer to type parameters or both don't refer to type parameters. M is considered to be less generic than N if Mt does not refer to a type parameter and Nt does.

For example:

```
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
```

Extension method type parameters that were fixed during currying are considered type parameters on the type, not type parameters on the method. For example:

```
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
```

11.8.1.3 Depth of genericity

A member `M` is determined to have *greater depth of genericity* than a member `N` if, for each pair of matching parameters `Mj` and `Nj`, `Mj` has greater or equal *depth of genericity* than `Nj`, and at least one `Mj` is has greater depth of genericity. Depth of genericity is defined as follows:

- Anything other than a type parameter has greater depth of genericity than a type parameter;
- Recursively, a constructed type has greater depth of genericity than another constructed type (with the same number of type arguments) if at least one type argument has greater depth of genericity and no type argument has less depth than the corresponding type argument in the other.
- An array type has greater depth of genericity than another array type (with the same number of dimensions) if the element type of the first has greater depth of genericity than the element type of the second.

For example:

```
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
```

11.8.2 Applicability To Argument List

A method is *applicable* to a set of type arguments, positional arguments, and named arguments if the method can be invoked using the argument lists. The argument lists are matched against the parameter lists as follows:

1. First, match each positional argument in order to the list of method parameters. If there are more positional arguments than parameters and the last parameter is not a paramarray, the method is not applicable. Otherwise, the paramarray parameter is expanded with parameters of the paramarray element type to match the number of positional arguments. If a positional argument is omitted that would go into a paramarray, the method is not applicable.
2. Next, match each named argument to a parameter with the given name. If one of the named arguments fails to match, matches a paramarray parameter, or matches an argument already matched with another positional or named argument, the method is not applicable.
3. Next, if type arguments have been specified, they are matched against the type parameter list . If the two lists do not have the same number of elements, the method is not applicable, unless the type argument list is empty. If the type argument list is empty, type inference is used to try and infer the type argument list. If type inferencing fails, the method is not applicable. Otherwise, the type arguments are filled in the place of the type parameters in the signature.If parameters that have not been matched are not optional, the method is not applicable.
4. If the argument expressions are not implicitly convertible to the types of the parameters they match, then the method is not applicable.
5. If a parameter is ByRef, and there is not an implicit conversion from the type of the parameter to the type of the argument, then the method is not applicable.
6. If type arguments violate the method's constraints (including the inferred type arguments from Step 3), the method is not applicable. For example:

```
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
```

If a single argument expression matches a paramarray parameter and the type of the argument expression is convertible to both the type of the paramarray parameter and the paramarray element type, the method is applicable in both its expanded and unexpanded forms, with two exceptions. If the conversion from the type of the argument expression to the paramarray type is narrowing, then the method is only applicable in its expanded form. If the argument expression is the literal `Nothing`, then the method is only applicable in its unexpanded form. For example:

```
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
```

The above example produces the following output:

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

In the first and last invocations of `F`, the normal form of `F` is applicable because a widening conversion exists from the argument type to the parameter type (both are of type `Object()`), and the argument is passed as a regular value parameter. In the second and third invocations, the normal form of `F` is not applicable because no widening conversion exists from the argument type to the parameter type (conversions from `Object` to `Object()` are narrowing). However, the expanded form of `F` is applicable, and a one-element `Object()` is created by the invocation. The single element of the array is initialized with the given argument value (which itself is a reference to an `Object()`).

11.8.3 Passing Arguments, and Picking Arguments for Optional Parameters

If a parameter is a value parameter, the matching argument expression must be classified as a value. The value is converted to the type of the parameter and passed in as the parameter at run time. If the parameter is a reference parameter and the matching argument expression is classified as a variable whose type is the same as the parameter, then a reference to the variable is passed in as the parameter at run time.

Otherwise, if the matching argument expression is classified as a variable, value, or property access, then a temporary variable of the type of the parameter is allocated. Before the method invocation at run time, the argument expression is reclassified as a value, converted to the type of the parameter, and assigned to the temporary variable. Then a reference to the temporary variable is passed in as the parameter. After the method invocation is evaluated, if the argument expression is classified as a variable or property access, the temporary variable is assigned to the variable expression or the property access expression. If the property access expression has no `Set` accessor, then the assignment is not performed.

For optional parameters where an argument has not been provided, the compiler picks arguments as described below. In all cases it tests against the parameter type after generic type substitution.

- If the optional parameter has the attribute `System.Runtime.CompilerServices.CallerLineNumber`, and the invocation is from a location in source code, and a numeric literal representing that location's line number has an intrinsic conversion to the parameter type, then the numeric literal is used. If the invocation spans multiple lines, then the choice of which line to use is implementation-dependent.
- If the optional parameter has the attribute `System.Runtime.CompilerServices.CallerFilePath`, and the invocation is from a location in source code, and a string literal representing that location's file path has an intrinsic conversion to the parameter type, then the string literal is used. The format of the file path is implementation-dependent.

- If the optional parameter has the attribute `System.Runtime.CompilerServices.CallerMemberName`, and the invocation is within the body of a type member or in an attribute applied to any part of that type member, and a string literal representing that member name has an intrinsic conversion to the parameter type, then the string literal is used. For invocations that are part of property accessors or custom event handlers, then the member name used is that of the property or event itself. For invocations that are part of an operator or constructor, then an implementation-specific name is used.

If none of the above apply, then the optional parameter's default value is used (or `Nothing` if no default value is supplied). And if more than one of the above apply, then the choice of which to use is implementation-dependent.

The `CallerLineNumber` and `CallerFilePath` attributes are useful for logging. The `CallerMemberName` is useful for implementing `INotifyPropertyChanged`. Here are examples.

```
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
```

In addition to the optional parameters above, Microsoft Visual Basic also recognizes some additional optional parameters if they are imported from metadata (i.e. from a DLL reference):

- Upon importing from metadata, Visual Basic also treats the parameter `<Optional>` as indicative that the parameter is optional: in this way it is possible to import a declaration which has an optional parameter but no default value, even though this can't be expressed using the `Optional` keyword.
- If the optional parameter has the attribute `Microsoft.VisualBasic.CompilerServices.OptionCompareAttribute`, and the numeric literal 1 or 0 has a conversion to the parameter type, then the compiler uses as argument either the literal 1 if `Option Compare Text` is in effect, or the literal 0 if `Optional Compare Binary` is in effect.
- If the optional parameter has the attribute `System.Runtime.CompilerServices.IDispatchConstantAttribute`, and it has type `Object`, and it does not specify a default value, then the compiler uses the argument `New System.Runtime.InteropServices.DispatchWrapper(Nothing)`.
- If the optional parameter has the attribute `System.Runtime.CompilerServices.IUnknownConstantAttribute`, and it has type `Object`, and it does not specify a default value, then the compiler uses the argument `New System.Runtime.InteropServices.UnknownWrapper(Nothing)`.
- If the optional parameter has type `Object`, and it does not specify a default value, then the compiler uses the argument `System.Reflection.Missing.Value`.

11.8.4 Conditional Methods

If the target method to which an invocation expression refers is a subroutine that is not a member of an interface and if the method has one or more `System.Diagnostics.ConditionalAttribute` attributes, evaluation of the expression depends on the conditional compilation constants defined at that point in the source file. Each instance of the attribute specifies a string, which names a conditional compilation constant. Each conditional compilation constant is evaluated as if it were part of a conditional compilation statement. If the constant evaluates to `True`, the expression is evaluated normally at run time. If the constant evaluates to `False`, the expression is not evaluated at all.

When looking for the attribute, the most derived declaration of an overridable method is checked.

Note. The attribute is not valid on functions or interface methods and is ignored if specified on either kind of method. Thus, conditional methods will only appear in invocation statements.

11.8.5 Type Argument Inference

When a method with type parameters is called without specifying type arguments, *type argument inference* is used to try and infer type arguments for the call. This allows a more natural syntax to be used for calling a method with type parameters when the type arguments can be trivially inferred. For example, given the following method declaration:

```
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
```

it is possible to invoke the `Choose` method without explicitly specifying a type argument:

```
' 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")
```

Through type argument inference, the type arguments `Integer` and `String` are determined from the arguments to the method.

Type argument inference occurs *before* expression reclassification is performed on lambda methods or method pointers in the argument list, since reclassification of those two kinds of expressions may require the type of the parameter to be known. Given a set of arguments `A1,...,An`, a set of matching parameters `P1,...,Pn` and a set of method type parameters `T1,...,Tn`, the dependencies between the arguments and method type parameters are first collected as follows:

- If `An` is the `Nothing` literal, no dependencies are generated.
- If `An` is a lambda method and the type of `Pn` is a constructed delegate type or `System.Linq.Expressions.Expression(Of T)`, where `T` is a constructed delegate type,
- If the type of a lambda method parameter will be inferred from the type of the corresponding parameter `Pn`, and the type of the parameter depends on a method type parameter `Tn`, then `An` has a dependency on `Tn`.
- If the type of a lambda method parameter is specified and the type of the corresponding parameter `Pn` depends on a method type parameter `Tn`, then `Tn` has a dependency on `An`.
- If the return type of `Pn` depends on a method type parameter `Tn`, then `Tn` has a dependency on `An`.
- If `An` is a method pointer and the type of `Pn` is a constructed delegate type,

- If the return type of `Pn` depends on a method type parameter `Tn`, then `Tn` has a dependency on `An`.
- If `Pn` is a constructed type and the type of `Pn` depends on a method type parameter `Tn`, then `Tn` has a dependency on `An`.
- Otherwise, no dependency is generated.

After collecting dependencies, any arguments that have no dependencies are eliminated. If any method type parameters have no outgoing dependencies (i.e. the method type parameter does not depend on an argument), then type inference fails. Otherwise, the remaining arguments and method type parameters are grouped into strongly connected components. A strongly connected component is a set of arguments and method type parameters, where any element in the component is reachable via dependencies on other elements.

The strongly connected components are then topologically sorted and processed in topological order:

- If the strongly typed component contains only one element,
  - If the element has already been marked complete, skip it.
  - If the element is an argument, then add type hints from the argument to the method type parameters that depend on it and mark the element as complete. If the argument is a lambda method with parameters that still need inferred types, then infer `Object` for the types of those parameters.
  - If the element is a method type parameter, then infer the method type parameter to be the dominant type among the argument type hints and mark the element as complete. If a type hint has an array element restriction on it, then only conversions that are valid between arrays of the given type are considered (i.e. covariant and intrinsic array conversions). If a type hint has a generic argument restriction on it, then only identity conversions are considered. If no dominant type can be chosen, inference fails. If any lambda method argument types depend on this method type parameter, the type is propagated to the lambda method.
- If the strongly typed component contains more than one element, then the component contains a cycle.
  - For each method type parameter that is an element in the component, if the method type parameter depends on an argument that is not marked complete, convert that dependency into an assertion that will be checked at the end of the inference process.
  - Restart the inference process at the point at which the strongly typed components were determined.

If type inference succeeds for all of the method type parameters, then any dependencies that were changed into assertions are checked. An assertion succeeds if the type of the argument is implicitly convertible to the inferred type of the method type parameter. If an assertion fails, then type argument inference fails.

Given an argument type `Ta` for an argument `A` and a parameter type `Tp` for a parameter `P`, type hints are generated as follows:

- If `Tp` does not involve any method type parameters then no hints are generated.
- If `Tp` and `Ta` are array types of the same rank, then replace `Ta` and `Tp` with the element types of `Ta` and `Tp` and restart this process with an array element restriction.
- If `Tp` is a method type parameter, then `Ta` is added as a type hint with the current restriction, if any.
- If `A` is a lambda method and `Tp` is a constructed delegate type or `System.Linq.Expressions.Expression(Of T)`, where `T` is a constructed delegate type, for each lambda method parameter type `TL` and corresponding delegate parameter type `TD`, replace `Ta` with `TL` and `Tp` with `TD` and restart the process with no restriction. Then, replace `Ta` with the return type of the lambda method and:
  - if `A` is a regular lambda method, replace `Tp` with the return type of the delegate type;
  - if `A` is an async lambda method and the return type of the delegate type has form `Task(Of T)` for some `T`, replace `Tp` with that `T`;

    - if `A` is an iterator lambda method and the return type of the delegate type has form `IEnumerator(Of T)` or `IEnumerable(Of T)` for some `T`, replace `Tp` with that `T`.
    - Next, restart the process with no restriction.
- If `A` is a method pointer and `Tp` is a constructed delegate type, use the parameter types of `Tp` to determine which method pointed is most applicable to `Tp`. If there is a method that is most applicable, replace `Ta` with the return type of the method and `Tp` with the return type of the delegate type and restart the process with no restriction.
- Otherwise, `Tp` must be a constructed type. Given `TG`, the generic type of `Tp`,
    - If `Ta` is `TG`, inherits from `TG`, or implements the type `TG` exactly once, then for each matching type argument `Tax` from `Ta` and `Tpx` from `Tp`, replace `Ta` with `Tax` and `Tp` with `Tpx` and restart the process with a generic argument restriction.
    - Otherwise, type inference fails for the generic method.

The success of type inference does not, in and of itself, guarantee that the method is applicable.

11.9 Index Expressions

An *index expression* results in an array element or reclassifies a property group into a property access. An index expression consists of, in order, an expression, an opening parenthesis, an index argument list, and a closing parenthesis.

```
IndexExpression:
    | Expression OpenParenthesis ArgumentList? CloseParenthesis
    ;
```

The target of the index expression must be classified as either a property group or a value. An index expression is processed as follows:

- If the target expression is classified as a value and if its type is not an array type, `Object`, or `System.Array`, the type must have a default property. The index is performed on a property group that represents all of the default properties of the type. Although it is not valid to declare a parameterless default property in Visual Basic, other languages may allow declaring such a property. Consequently, indexing a property with no arguments is allowed.
- If the expression results in a value of an array type, the number of arguments in the argument list must be the same as the rank of the array type and may not include named arguments. If any of the indexes are invalid at run time, a `System.IndexOutOfRangeException` exception is thrown. Each expression must be implicitly convertible to type `Integer`. The result of the index expression is the variable at the specified index and is classified as a variable.
- If the expression is classified as a property group, overload resolution is used to determine whether one of the properties is applicable to the index argument list. If the property group only contains one property that has a `Get` accessor and if that accessor takes no arguments, then the property group is interpreted as an index expression with an empty argument list. The result is used as the target of the current index expression. If no properties are applicable, then a compile-time error occurs. Otherwise, the expression results in a property access with the associated target expression (if any) of the property group.
- If the expression is classified as a late-bound property group or as a value whose type is `Object` or `System.Array`, the processing of the index expression is deferred until run time and the indexing is late-bound. The expression results in a late-bound property access typed as `Object`. The associated target expression is either the target expression, if it is a value, or the associated target expression of the property group. At run time the expression is processed as follows:

- If the expression is classified as a late-bound property group, the expression may result in a method group, a property group, or a value (if the member is an instance or shared variable). If the result is a method group or property group, overload resolution is applied to the group to determine the correct method for the argument list. If overload resolution fails, a `System.Reflection.AmbiguousMatchException` exception is thrown. Then the result is processed either as a property access or as an invocation and the result is returned. If the invocation is of a subroutine, the result is `Nothing`.
- If the run-time type of the target expression is an array type or `System.Array`, the result of the index expression is the value of the variable at the specified index.
- Otherwise, the run-time type of the expression must have a default property and the index is performed on the property group that represents all of the default properties on the type. If the type has no default property, then a `System.MissingMemberException` exception is thrown.

11.10 New Expressions

The `New` operator is used to create new instances of types. There are four forms of `New` expressions:

- Object-creation expressions are used to create new instances of class types and value types.
- Array-creation expressions are used to create new instances of array types.
- Delegate-creation expressions (which do not have a distinct syntax from object-creation expressions) are used to create new instances of delegate types.
- Anonymous object-creation expressions are used to create new instances of anonymous class types.

```
NewExpression:
    | ObjectCreationExpression
    | ArrayExpression
    | AnonymousObjectCreationExpression
    ;
```

A `New` expression is classified as a value and the result is the new instance of the type.

11.10.1 Object-Creation Expressions

An object-creation expression is used to create a new instance of a class type or a structure type.

```
ObjectCreationExpression:
    | 'New' NonArrayTypeName ( OpenParenthesis ArgumentList? CloseParenthesis  )?
      ObjectCreationExpressionInitializer?
    ;

ObjectCreationExpressionInitializer:
    | ObjectMemberInitializer
    | ObjectCollectionInitializer
    ;

ObjectMemberInitializer:
    | 'With' OpenCurlyBrace FieldInitializerList CloseCurlyBrace
    ;

FieldInitializerList:
    | FieldInitializer ( Comma FieldInitializer  )*
    ;

FieldInitializer:
    | 'Key'? ( '.' IdentifierOrKeyword Equals  )? Expression
    ;
```

```
ObjectCollectionInitializer:
    | 'From' CollectionInitializer
    ;

CollectionInitializer:
    | OpenCurlyBrace CollectionElementList? CloseCurlyBrace
    ;

CollectionElementList:
    | CollectionElement ( Comma CollectionElement  )*
    ;

CollectionElement:
    | Expression
    | CollectionInitializer
    ;
```

The type of an object creation expression must be a class type, a structure type, or a type parameter with a `New` constraint and cannot be a `MustInherit` class. Given an object creation expression of the form `New T(A)`, where `T` is a class type or structure type and `A` is an optional argument list, overload resolution determines the correct constructor of `T` to call. A type parameter with a `New` constraint is considered to have a single, parameterless constructor. If no constructor is callable, a compile-time error occurs; otherwise the expression results in the creation of a new instance of `T` using the chosen constructor. If there are no arguments, the parentheses may be omitted.

Where an instance is allocated depends on whether the instance is a class type or a value type. `New` instances of class types are created on the system heap, while new instances of value types are created directly on the stack.

An object-creation expression can optionally specify a list of member initializers after the constructor arguments. These member initializers are prefixed with the keyword `With`, and the initializer list is interpreted as if it was in the context of a `With` statement. For example, given the class:

```
Class Customer
    Dim Name As String
    Dim Address As String
End Class
```

The code:

```
Module Test
    Sub Main()
        Dim x As New Customer() With { .Name = "Bob Smith", _
            .Address = "123 Main St." }
    End Sub
End Module
```

Is roughly equivalent to:

```
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
```

Each initializer must specify a name to assign, and the name must be a non-`ReadOnly` instance variable or property of the type being constructed; the member access will not be late bound if the type being constructed is `Object`. Initializers may not use the `Key` keyword. Each member in a type can only be initialized once. The initializer expressions, however, may refer to each other. For example:

```
Module Test
    Sub Main()
        Dim x As New Customer() With { .Name = "Bob Smith", _
            .Address = .Name & " St." }
    End Sub
End Module
```

The initializers are assigned left-to-right, so if an initializer refers to a member that has not been initialized yet, it will see whatever value the instance variable after the constructor ran:

```
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
```

Initializers can be nested:

```
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
```

If the type being created is a collection type and has an instance method named `Add` (including extension methods and shared methods), then the object-creation expression can specify a collection initializer prefixed by the keyword `From`. An object-creation expression cannot specify both a member initializer and a collection initializer. Each element in the collection initializer is passed as an argument to an invocation of the `Add` function. For example:

```
Dim list = New List(Of Integer)() From { 1, 2, 3, 4 }
```

is equivalent to:

```
Dim list = New List(Of Integer)()
list.Add(1)
list.Add(2)
list.Add(3)
```

If an element is a collection initializer itself, each element of the sub-collection initializer will be passed as an individual argument to the `Add` function. For example, the following:

```
Dim dict = Dictionary(Of Integer, String) From { { 1, "One" },{ 2, "Two" } }
```

is equivalent to:

```
Dim dict = New Dictionary(Of Integer, String)
dict.Add(1, "One")
dict.Add(2, "Two")
```

This expansion is always done and is only ever done one level deep; after that, sub-initializers are considered array literals. For example:

```
' 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 } } }
```

11.10.2 Array Expressions

An array expression is used to create a new instance of an array type. There are two types of array expressions: array creation expressions, and array literals.

11.10.2.1 Array creation expressions

If an array size initialization modifier is supplied, the resulting array type is derived by deleting each of the individual arguments from the array size initialization argument list. The value of each argument determines the upper bound of the corresponding dimension in the newly allocated array instance. If the expression has a non-empty collection initializer, each argument in the argument list must be a constant, and the rank and dimension lengths specified by the expression list must match those of the collection initializer.

```
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 }
```

If an array size initialization modifier is not supplied, then the type name must be an array type and the collection initializer must be empty or have the same number of levels of nesting as the rank of the specified array type. All of the elements in the innermost nesting level must be implicitly convertible to the element type of the array and must be classified as a value. The number of elements in each nested collection initializer must always be consistent with the size of the other collections at the same level. The individual dimension lengths are inferred from the number of elements in each of the corresponding nesting levels of the collection initializer. If the collection initializer is empty, the length of each dimension is zero.

```
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 } }
```

The outermost nesting level of a collection initializer corresponds to the leftmost dimension of an array, and the innermost nesting level corresponds to the rightmost dimension. The example:

```
Dim array As Integer(,) = _
    { { 0, 1 }, { 2, 3 }, { 4, 5 }, { 6, 7 }, { 8, 9 } }
```

Is equivalent to the following:

```
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
```

If the collection initializer is empty (that is, one that contains curly braces but no initializer list) and the bounds of the dimensions of the array being initialized are known, the empty collection initializer represents an array instance of the specified size where all the elements have been initialized to the element type's default value. If the bounds of the dimensions of the array being initialized are not known, the empty collection initializer represents an array instance in which all dimensions are size zero.

An array instance's rank and length of each dimension are constant for the entire lifetime of the instance. In other words, it is not possible to change the rank of an existing array instance, nor is it possible to resize its dimensions.

11.10.2.2 Array Literals

An array literal denotes an array whose element type, rank, and bounds are inferred from a combination of the expression context and a collection initializer. This is explained in Section §11.1.1.

```
ArrayExpression:
    | ArrayCreationExpression
    | ArrayLiteralExpression
    ;

ArrayCreationExpression:
    | 'New' NonArrayTypeName ArrayNameModifier CollectionInitializer
    ;

ArrayLiteralExpression:
    | CollectionInitializer
    ;
```

For example:

```
' 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}}
```

The format and requirements for the collection initializer in an array literal is exactly the same as that for the collection initializer in an array creation expression.

Note. An array literal does not create the array in and of itself; instead, it is the reclassification of the expression into a value that causes the array to be created. For instance, the conversion `CType(new Integer() {1,2,3}, Short())` is not possible because there is no conversion from `Integer()` to `Short()`; but the expression `CType({1,2,3},Short())` is possible because it first reclassifies the array literal into the array creation expression `New Short() {1,2,3}`.

11.10.3 Delegate-Creation Expressions

A delegate-creation expression is used to create a new instance of a delegate type. The argument of a delegate-creation expression must be an expression classified as a method pointer or a lambda method.

If the argument is a method pointer, one of the methods referenced by the method pointer must be applicable to the signature of the delegate type. A method `M` is applicable to a delegate type `D` if:

- `M` is not `Partial` or has a body.
- Both `M` and `D` are functions, or `D` is a subroutine.
- `M` and `D` have the same number of parameters.
- The parameter types of `M` each have a conversion from the type of the corresponding parameter type of `D`, and their modifiers (i.e. `ByRef`, `ByVal`) match.
- The return type of `M`, if any, has a conversion to the return type of `D`.

If the method pointer references a late-bound access, then the late-bound access is assumed to be to a function that has the same number of parameters as the delegate type.

If strict semantics are not being used and there is only one method referenced by the method pointer, but it is not applicable due to the fact that it has no parameters and the delegate type does, then the method is considered applicable and the parameters or return value are simply ignored. For example:

```
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
```

Note. This relaxation is only allowed when strict semantics are not being used because of extension methods. Because extension methods are only considered if a regular method was not applicable, it is possible for an instance method with no parameters to hide an extension method with parameters for the purpose of delegate construction.

If more than one method referenced by the method pointer is applicable to the delegate type, then overload resolution is used to pick between the candidate methods. The types of the parameters to the delegate are used as the types of the arguments for the purposes of overload resolution. If no one method candidate is most applicable, a compile-time error occurs. In the following example, the local variable is initialized with a delegate that refers to the second `Square` method because that method is more applicable to the signature and return type of `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
```

Had the second `Square` method not been present, the first `Square` method would have been chosen. If strict semantics are specified by the compilation environment or by `Option Strict`, then a compile-time error occurs if the most specific method referenced by the method pointer is narrower than the delegate signature. A method `M` is considered narrower than a delegate type `D` if:

- A parameter type of `M` has a widening conversion to the corresponding parameter type of `D`.
- Or, the return type, if any, of `M` has a narrowing conversion to the return type of `D`.

If type arguments are associated with the method pointer, only methods with the same number of type arguments are considered. If no type arguments are associated with the method pointer, type inference is used when matching signatures against a generic method. Unlike other normal type inference, the return type of the delegate is used when inferring type arguments, but return types are still not considered when determining the least generic overload. The following example shows both ways of supplying a type argument to a delegate-creation expression:

```
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
```

In the above example, a non-generic delegate type was instantiated using a generic method. It is also possible to create an instance of a constructed delegate type using a generic method. For example:

```
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
```

If the argument to the delegate-creation expression is a lambda method, the lambda method must be applicable to the signature of the delegate type. A lambda method `L` is applicable to a delegate type `D` if:

- If `L` has parameters, `D` has the same number of parameters. (If `L` has no parameters, the parameters of `D` are ignored.)
- The parameter types of `L` each have a conversion to the type of the corresponding parameter type of `D`, and their modifiers (i.e. `ByRef`, `ByVal`) match.
- If `D` is a function, the return type of `L` has a conversion to the return type of `D`. (If `D` is a subroutine, the return value of `L` is ignored.)

If the parameter type of a parameter of `L` is omitted, then the type of the corresponding parameter in `D` is inferred; if the parameter of `L` has array or nullable name modifiers, a compile-time error results. Once all of the parameter types of `L` are available, then the type of the expression in the lambda method is inferred. For example:

```
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
```

In some situations where delegate signature does not exactly match the lambda method or method signature, the .NET Framework may not support the delegate creation natively. In that situation, a lambda method expression is used to match the two methods. For example:

```
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
```

The result of a delegate-creation expression is a delegate instance that refers to the matching method with the associated target expression (if any) from the method pointer expression. If the target expression is typed as a value type, then the value type is copied onto the system heap because a delegate can only point to a method of an object on the heap. The method and object to which a delegate refers remain constant for the entire lifetime of the delegate. In other words, it is not possible to change the target or object of a delegate after it has been created.

11.10.4 Anonymous Object-Creation Expressions

An object-creation expression with member initializers can also omit the type name entirely.

```
AnonymousObjectCreationExpression:
    | 'New' ObjectMemberInitializer
    ;
```

In that case, an anonymous type is constructed based on the types and names of the members initialized as a part of the expression. For example:

```
Module Test
    Sub Main()
        Dim Customer = New With { .Name = "John Smith", .Age = 34 }

        Console.WriteLine(Customer.Name)
    End Sub
End Module
```

The type created by an anonymous object-creation expression is a class that has no name, inherits directly from `Object`, and has a set of properties with the same name as the members assigned to in the member initializer list. The type of each property is inferred using the same rules as local variable type inference. Generated anonymous types also override `ToString`, returning a string representation of all members and their values. (The exact format of this string is beyond the scope of this specification).

By default, the properties generated by the anonymous type are read-write. It is possible to mark an anonymous type property as read-only by using the `Key` modifier. The `Key` modifier specifies that the field can be used to uniquely identify the value the anonymous type represents. In addition to making the property read-only, it also causes the anonymous type to override `Equals` and `GetHashCode` and to implement the interface `System.IEquatable(Of T)` (filling in the anonymous type for `T`). The members are defined as follows:

`Function Equals(obj As Object) As Boolean` and `Function Equals(val As T) As Boolean` are implemented by validating that the two instances are of the same type and then comparing each `Key` member using `Object.Equals`. If all `Key` members are equal, then `Equals` returns `True`, otherwise `Equals` returns `False`.

`Function GetHashCode() As Integer` is implemented such that that if `Equals` is true for two instances of the anonymous type, then `GetHashCode` will return the same value. The hash starts with a seed value and then, for each `Key` member, in order multiplies the hash by 31 and adds the `Key` member's hash value (provided by `GetHashCode`) if the member is not a reference type or nullable value type with the value of `Nothing`.

For example, the type created in the statement:

```
Dim zipState = New With { Key .ZipCode = 98112, .State = "WA" }
```

creates a class that looks approximately like this (although exact implementation may vary):

```
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
```

To simplify the situation where an anonymous type is created from the fields of another type, field names can be inferred directly from expressions in the following cases:

- A simple name expression `x` infers the name `x`.
- A member access expression `x.y` infers the name `y`.
- A dictionary lookup expression `x!y` infers the name `y`.
- An invocation or index expression with no arguments `x()` infers the name `x`.
- An XML member access expression `x.<y>`, `x...<y>`, `x.@y` infers the name `y`.
- An XML member access expression that is the target of a member access expression `x.<y>.z` infers the name `z`.
- An XML member access expression that is the target of an invocation or index expression with no arguments `x.<y>.z()` infers the name `z`.
- An XML member access expression that is the target of an invocation or index expression `x.<y>(0)` infers the name `y`.

The initializer is interpreted as an assignment of the expression to the inferred name. For example, the following initializers are equivalent:

```
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
```

If a member name is inferred that conflicts with an existing member of the type, such as `GetHashCode`, then a compile time error occurs. Unlike regular member initializers, anonymous object-creation expressions do not allow member initializers to have circular references, or to refer to a member before it has been initialized. For example:

```
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
```

If two anonymous class creation expressions occur within the same method and yield the same resulting shape -- if the property order, property names, and property types all match -- they will both refer to the same anonymous class. The method scope of an instance or shared member variable with an initializer is the constructor in which the variable is initialized.

Note. It is possible that a compiler may choose to unify anonymous types further, such as at the assembly level, but this cannot be relied upon at this time.

11.11 Cast Expressions

A cast expression coerces an expression to a given type. Specific cast keywords coerce expressions into the primitive types. Three general cast keywords, `CType`, `TryCast` and `DirectCast`, coerce an expression into a type.

```
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'
    ;
```

DirectCast and TryCast have special behaviors. Because of this, they only support native conversions. Additionally, the target type in a TryCast expression cannot be a value type. User-defined conversion operators are not considered when DirectCast or TryCast is used. (**Note.** The conversion set that DirectCast and TryCast support are restricted because they implement "native CLR" conversions. The purpose of DirectCast is to provide the functionality of the "unbox" instruction, while the purpose of TryCast is to provide the functionality of the "isinst" instruction. Since they map onto CLR instructions, supporting conversions not directly supported by the CLR would defeat the intended purpose.)

DirectCast converts expressions that are typed as Object differently than CType. When converting an expression of type Object whose run-time type is a primitive value type, DirectCast throws a System.InvalidCastException exception if the specified type is not the same as the run-time type of the expression or a System.NullReferenceException if the expression evaluates to Nothing. (**Note.** As noted above, DirectCast maps directly onto the CLR instruction "unbox" when the type of the expression is Object. In contrast, CType turns into a call to a runtime helper to do the conversion so that conversions between primitive types can be supported. In the case when an Object expression is being converted to a primitive value type and the type of the actual instance match the target type, DirectCast will be significantly faster than CType.)

TryCast converts expressions but does not throw an exception if the expression cannot be converted to the target type. Instead, TryCast will result in Nothing if the expression cannot be converted at runtime. (**Note.** As noted above, TryCast maps directly onto the CLR instruction "isinst". By combining the type check and the conversion into a single operation, TryCast can be cheaper than doing a TypeOf ... Is and then a CType.)

For example:

```
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
```

If no conversion exists from the type of the expression to the specified type, a compile-time error occurs. Otherwise, the expression is classified as a value and the result is the value produced by the conversion.

11.12 Operator Expressions

There are two kinds of operators. *Unary operators* take one operand and use prefix notation (for example, -x). *Binary operators* take two operands and use infix notation (for example, x + y). With the exception of the relational operators, which always result in Boolean, an operator defined for a particular type results in that type. The operands to an operator must always be classified as a value; the result of an operator expression is classified as a value.

```
OperatorExpression:
    | ArithmeticOperatorExpression
    | RelationalOperatorExpression
    | LikeOperatorExpression
    | ConcatenationOperatorExpression
    | ShortCircuitLogicalOperatorExpression
    | LogicalOperatorExpression
    | ShiftOperatorExpression
```

```
    | AwaitOperatorExpression
    ;
```

11.12.1 Operator Precedence and Associativity

When an expression contains multiple binary operators, the *precedence* of the operators controls the order in which the individual binary operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the `+` operator. The following table lists the binary operators in descending order of precedence:

Category	Operators
Primary	All non-operator expressions
Await	`Await`
Exponentiation	`^`
Unary negation	`+`, `-`
Multiplicative	`*`, `/`
Integer division	`\`
Modulus	`Mod`
Additive	`+`, `-`
Concatenation	`&`
Shift	`<<`, `>>`
Relational	`=`, `<>`, `<`, `>`, `<=`, `>=`, `Like`, `Is`, `IsNot`
Logical NOT	`Not`
Logical AND	`And`, `AndAlso`
Logical OR	`Or`, `OrElse`
Logical XOR	`Xor`

When an expression contains two operators with the same precedence, the *associativity* of the operators controls the order in which the operations are performed. All binary operators are left-associative, meaning that operations are performed from left to right. Precedence and associativity can be controlled using parenthetical expressions.

11.12.2 Object Operands

In addition to the regular types supported by each operator, all operators support operands of type `Object`. Operators applied to `Object` operands are handled similarly to method calls made on `Object` values: a late-bound method call might be chosen, in which case the run-time type of the operands, rather than the compile-time type, determines the validity and type of the operation. If strict semantics are specified by the compilation environment or by `Option Strict`, any operators with operands of type `Object` cause a compile-time error, except for the `TypeOf...Is`, `Is` and `IsNot` operators.

When operator resolution determines that an operation should be performed late-bound, the outcome of the operation is the result of applying the operator to the operand types if the run-time types of the operands are types that are supported by the operator. The value `Nothing` is treated as the default value of the type of the other operand in a binary operator expression. In a unary operator expression, or if both operands are `Nothing` in a binary operator expression, the type of the operation is `Integer` or the only result type of the operator, if the operator does not result in `Integer`. The result of the operation is always then cast back to `Object`. If the operand types have no valid operator, a `System.InvalidCastException` exception is thrown. Conversions at run time are done without regard to whether they are implicit or explicit.

If the result of a numeric binary operation would produce an overflow exception (regardless of whether integer overflow checking is on or off), then the result type is promoted to the next wider numeric type, if possible. For example, consider the following code:

```
Module Test
    Sub Main()
        Dim o As Object = CObj(CByte(2)) * CObj(CByte(255))

        Console.WriteLine(o.GetType().ToString() & " = " & o)
    End Sub
End Module
```

It prints the following result:

```
System.Int16 = 512
```

If no wider numeric type is available to hold the number, a `System.OverflowException` exception is thrown.

11.12.3 Operator Resolution

Given an operator type and a set of operands, operator resolution determines which operator to use for the operands. When resolving operators, user-defined operators will be considered first, using the following steps:

1. First, all of the candidate operators are collected. The candidate operators are all of the user-defined operators of the particular operator type in the source type and all of the user-defined operators of the particular type in the target type. If the source type and destination type are related, common operators are only considered once.
2. Then, overload resolution is applied to the operators and operands to select the most specific operator. In the case of binary operators, this may result in a late-bound call.

When collecting the candidate operators for a type `T?`, the operators of type `T` are used instead. Any of `T`'s user-defined operators that involve only non-nullable value types are also lifted. A lifted operator uses the nullable version of any value types, with the exception the return types of `IsTrue` and `IsFalse` (which must be `Boolean`). Lifted operators are evaluated by converting the operands to their non-nullable version, then evaluating the user-defined operator and then converting the result type to its nullable version. If ether operand is `Nothing`, the result of the expression is a value of `Nothing` typed as the nullable version of the result type. For example:

```
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
```

If the operator is a binary operator and one of the operands is reference type, the operator is also lifted, but any binding to the operator produces an error. For example:

```
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
```

Note. This rule exists because there has been consideration whether we wish to add null-propagating reference types in a future version, in which case the behavior in the case of binary operators between the two types would change.

As with conversions, user-defined operators are always preferred over lifted operators.

When resolving overloaded operators, there may be differences between classes defined in Visual Basic and those defined in other languages:

- In other languages, `Not`, `And`, and `Or` may be overloaded both as logical operators and bitwise operators. Upon import from an external assembly, either form is accepted as a valid overload for these operators. However, for a type which defines both logical and bitwise operators, only the bitwise implementation will be considered.
- In other languages, `>>` and `<<` may be overloaded both as signed operators and unsigned operators. Upon import from an external assembly, either form is accepted as a valid overload. However, for a type which defines both signed and unsigned operators, only the signed implementation will be considered.
- If no user-defined operator is most specific to the operands, then intrinsic operators will be considered. If no intrinsic operator is defined for the operands and either operand has type Object then the operator will be resolved late-bound; otherwise, a compile-time error results.

In prior versions of Visual Basic, if there was exactly one operand of type Object, and no applicable user-defined operators, and no applicable intrinsic operators, then it was an error. As of Visual Basic 11, it is now resolved late-bound. For example:

```
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
```

A type `T` that has an intrinsic operator also defines that same operator for `T?`. The result of the operator on `T?` will be the same as for `T`, except that if either operand is `Nothing`, the result of the operator will be `Nothing` (i.e. the null value is propagated). For the purposes of resolving the type of an operation, the `?` is removed from any operands that have them, the type of the operation is determined, and a `?` is added to the type of the operation if any of the operands were nullable value types. For example:

```
Dim v1? As Integer = 10
Dim v2 As Long = 20
```

```
' Type of operation will be Long?
Console.WriteLine(v1 + v2)
```

Each operator lists the intrinsic types it is defined for and the type of the operation performed given the operand types. The result of type of a intrinsic operation follows these general rules:

- If all operands are of the same type, and the operator is defined for the type, then no conversion occurs and the operator for that type is used.
- Any operand whose type is not defined for the operator is converted using the following steps and the operator is resolved against the new types:
  - The operand is converted to the next widest type that is defined for both the operator and the operand and to which it is implicitly convertible.
  - If there is no such type, then the operand is converted to the next narrowest type that is defined for both the operator and the operand and to which it is implicitly convertible.
  - If there is no such type or the conversion cannot occur, a compile-time error occurs.
- Otherwise, the operands are converted to the wider of the operand types and the operator for that type is used. If the narrower operand type cannot be implicitly converted to the wider operator type, a compile-time error occurs.

Despite these general rules, however, there are a number of special cases called out in the operator results tables.

Note. For formatting reasons, the operator type tables abbreviate the predefined names to their first two characters. So "By" is `Byte`, "UI" is `UInteger`, "St" is `String`, etc. "Err" means that there is no operation defined for the given operand types.

11.13 Arithmetic Operators

The `*`, `/`, `\`, `^`, `Mod`, `+`, and `-` operators are the *arithmetic operators*.

```
ArithmeticOperatorExpression:
    | UnaryPlusExpression
    | UnaryMinusExpression
    | AdditionOperatorExpression
    | SubtractionOperatorExpression
    | MultiplicationOperatorExpression
    | DivisionOperatorExpression
    | ModuloOperatorExpression
    | ExponentOperatorExpression
    ;
```

Floating-point arithmetic operations may be performed with higher precision than the result type of the operation. For example, some hardware architectures support an "extended" or "long double" floating-point type with greater range and precision than the `Double` type, and implicitly perform all floating-point operations using this higher-precision type. Hardware architectures can be made to perform floating-point operations with less precision only at excessive cost in performance; rather than require an implementation to forfeit both performance and precision, Visual Basic allows the higher-precision type to be used for all floating-point operations. Other than delivering more precise results, this rarely has any measurable effects. However, in expressions of the form `x * y / z`, where the multiplication produces a result that is outside the `Double` range, but the subsequent division brings the temporary result back into the `Double` range, the fact that the expression is evaluated in a higher-range format may cause a finite result to be produced instead of infinity.

11.13.1 Unary Plus Operator

```
UnaryPlusExpression:
    | '+' Expression
    ;
```

The unary plus operator is defined for the `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Single`, `Double`, and `Decimal` types.

Operation Type:

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

11.13.2 Unary Minus Operator

```
UnaryMinusExpression:
    | '-' Expression
    ;
```

The unary minus operator is defined for the following types:

`SByte`, `Short`, `Integer`, and `Long`. The result is computed by subtracting the operand from zero. If integer overflow checking is on and the value of the operand is the maximum negative `SByte`, `Short`, `Integer`, or `Long`, a `System.OverflowException` exception is thrown. Otherwise, if the value of the operand is the maximum negative `SByte`, `Short`, `Integer`, or `Long`, the result is that same value, and the overflow is not reported.

`Single` and `Double`. The result is the value of the operand with its sign inverted, including the values 0 and Infinity. If the operand is NaN, the result is also NaN.

`Decimal`. The result is computed by subtracting the operand from zero.

Operation Type:

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

11.13.3 Addition Operator

The addition operator computes the sum of the two operands.

```
AdditionOperatorExpression:
    | Expression '+' LineTerminator? Expression
    ;
```

The addition operator is defined for the following types:

- `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, and `Long`. If integer overflow checking is on and the sum is outside the range of the result type, a `System.OverflowException` exception is thrown. Otherwise, overflows are not reported, and any significant high-order bits of the result are discarded.
- `Single` and `Double`. The sum is computed according to the rules of IEEE 754 arithmetic.
- `Decimal`. If the resulting value is too large to represent in the decimal format, a `System.OverflowException` exception is thrown. If the result value is too small to represent in the decimal format, the result is 0.
- `String`. The two `String` operands are concatenated together.

- `Date`. The `System.DateTime` type defines overloaded addition operators. Because `System.DateTime` is equivalent to the intrinsic `Date` type, these operators is also available on the `Date` type.

Operation Type:

	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

11.13.4 Subtraction Operator

The subtraction operator subtracts the second operand from the first operand.

```
SubtractionOperatorExpression:
    | Expression '-' LineTerminator? Expression
    ;
```

The subtraction operator is defined for the following types:

- `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, and `Long`. If integer overflow checking is on and the difference is outside the range of the result type, a `System.OverflowException` exception is thrown. Otherwise, overflows are not reported, and any significant high-order bits of the result are discarded.
- `Single` and `Double`. The difference is computed according to the rules of IEEE 754 arithmetic.
- `Decimal`. If the resulting value is too large to represent in the decimal format, a `System.OverflowException` exception is thrown. If the result value is too small to represent in the decimal format, the result is 0.
- `Date`. The `System.DateTime` type defines overloaded subtraction operators. Because `System.DateTime` is equivalent to the intrinsic `Date` type, these operators is also available on the `Date` type.

Operation Type:

	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

11.13.5 Multiplication Operator

The multiplication operator computes the product of two operands.

```
MultiplicationOperatorExpression:
    | Expression '*' LineTerminator? Expression
    ;
```

The multiplication operator is defined for the following types:

- `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, and `Long`. If integer overflow checking is on and the product is outside the range of the result type, a `System.OverflowException` exception is thrown. Otherwise, overflows are not reported, and any significant high-order bits of the result are discarded.
- `Single` and `Double`. The product is computed according to the rules of IEEE 754 arithmetic.
- `Decimal`. If the resulting value is too large to represent in the decimal format, a `System.OverflowException` exception is thrown. If the result value is too small to represent in the decimal format, the result is 0.

Operation Type:

	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

11.13.6 Division Operators

Division operators compute the quotient of two operands. There are two division operators: the regular (floating-point) division operator and the integer division operator.

```
DivisionOperatorExpression:
    | FPDivisionOperatorExpression
    | IntegerDivisionOperatorExpression
    ;

FPDivisionOperatorExpression:
    | Expression '/' LineTerminator? Expression
    ;

IntegerDivisionOperatorExpression:
    | Expression '\\' LineTerminator? Expression
    ;
```

The regular division operator is defined for the following types:

- `Single` and `Double`. The quotient is computed according to the rules of IEEE 754 arithmetic.
- `Decimal`. If the value of the right operand is zero, a `System.DivideByZeroException` exception is thrown. If the resulting value is too large to represent in the decimal format, a `System.OverflowException` exception is thrown. If the result value is too small to represent in the decimal format, the result is zero. The scale of the result, before any rounding, is the closest scale to the preferred scale which will preserve a result equal to the exact result. The preferred scale is the scale of the first operand less the scale of the second operand.

According to normal operator resolution rules, regular division purely between operands of types such as `Byte`, `Short`, `Integer`, and `Long` would cause both operands to be converted to type `Decimal`. However, when doing operator resolution on the division operator when neither type is `Decimal`, `Double` is considered narrower than `Decimal`. This convention is followed because `Double` division is more efficient than `Decimal` division.

Operation Type:

	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

The integer division operator is defined for `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, and `Long`. If the value of the right operand is zero, a `System.DivideByZeroException` exception is thrown. The division rounds the result towards zero, and the absolute value of the result is the largest possible integer that is less than the absolute value of the quotient of the two operands. The result is zero or positive when the two operands have the same sign, and zero or negative when the two operands have opposite signs. If the left operand is the maximum negative `SByte`, `Short`, `Integer`, or `Long`, and the right operand is `-1`, an overflow occurs; if integer overflow checking is on, a `System.OverflowException` exception is thrown. Otherwise, the overflow is not reported and the result is instead the value of the left operand.

Note. As the two operands for unsigned types will always be zero or positive, the result is always zero or positive. As the result of the expression will always be less than or equal to the largest of the two operands, it is not possible for an overflow to occur. As such integer overflow checking is not performed for integer divide with two unsigned integers. The result is the type as that of the left operand.

Operation Type:

	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

11.13.7 Mod Operator

The `Mod` (modulo) operator computes the remainder of the division between two operands.

```
ModuloOperatorExpression:
    | Expression 'Mod' LineTerminator? Expression
    ;
```

The `Mod` operator is defined for the following types:

- `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong` and `Long`. The result of `x Mod y` is the value produced by `x - (x \ y) * y`. If `y` is zero, a `System.DivideByZeroException` exception is thrown. The modulo operator never causes an overflow.
- `Single` and `Double`. The remainder is computed according to the rules of IEEE 754 arithmetic.
- `Decimal`. If the value of the right operand is zero, a `System.DivideByZeroException` exception is thrown. If the resulting value is too large to represent in the decimal format, a `System.OverflowException` exception is thrown. If the result value is too small to represent in the decimal format, the result is zero.

Operation Type:

	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

11.13.8 Exponentiation Operator

The exponentiation operator computes the first operand raised to the power of the second operand.

```
ExponentOperatorExpression:
    | Expression '^' LineTerminator? Expression
    ;
```

The exponentiation operator is defined for type `Double`. The value is computed according to the rules of IEEE 754 arithmetic.

Operation Type:

	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

11.14 Relational Operators

The *relational operators* compare values to one other. The comparison operators are `=`, `<>`, `<`, `>`, `<=`, and `>=`.

```
RelationalOperatorExpression:
    | Expression '=' LineTerminator? Expression
    | Expression '<' '>' LineTerminator? Expression
    | Expression '<' LineTerminator? Expression
    | Expression '>' LineTerminator? Expression
    | Expression '<' '=' LineTerminator? Expression
    | Expression '>' '=' LineTerminator? Expression
    ;
```

All of the relational operators result in a `Boolean` value.

The relational operators have the following general meaning:

- The `=` operator tests whether the two operands are equal.
- The `<>` operator tests whether the two operands are not equal.
- The `<` operator tests whether the first operand is less than the second operand.
- The `>` operator tests whether the first operand is greater than the second operand.
- The `<=` operator tests whether the first operand is less than or equal to the second operand.
- The `>=` operator tests whether the first operand is greater than or equal to the second operand.

The relational operators are defined for the following types:

- `Boolean`. The operators compare the truth values of the two operands. `True` is considered to be less than `False`, which matches with their numeric values.

- `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, and `Long`. The operators compare the numeric values of the two integral operands.
- `Single` and `Double`. The operators compare the operands according to the rules of the IEEE 754 standard.
- `Decimal`. The operators compare the numeric values of the two decimal operands.
- `Date`. The operators return the result of comparing the two date/time values.
- `Char`. The operators return the result of comparing the two Unicode values.
- `String`. The operators return the result of comparing the two values using either a binary comparison or a text comparison. The comparison used is determined by the compilation environment and the `Option Compare` statement. A binary comparison determines whether the numeric Unicode value of each character in each string is the same. A text comparison does a Unicode text comparison based on the current culture in use on the .NET Framework. When doing a string comparison, a null value is equivalent to the string literal `""`.

Operation Type:

	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

11.15 Like Operator

The `Like` operator determines whether a string matches a given pattern.

```
LikeOperatorExpression:
    | Expression 'Like' LineTerminator? Expression
    ;
```

The `Like` operator is defined for the `String` type. The first operand is the string being matched, and the second operand is the pattern to match against. The pattern is made up of Unicode characters. The following character sequences have special meanings:

- The character `?` matches any single character.

- The character `*` matches zero or more characters.
- The character `#` matches any single digit (0-9).
- A list of characters surrounded by brackets (`[ab...]`) matches any single character in the list.
- A list of characters surrounded by brackets and prefixed by an exclamation point (`[!ab...]`) matches any single character not in the character list.
- Two characters in a character list separated by a hyphen (`-`) specify a range of Unicode characters starting with the first character and ending with the second character. If the second character is not later in the sort order than the first character, a run-time exception occurs. A hyphen that appears at the beginning or end of a character list specifies itself.

To match the special characters left bracket (`[`), question mark (`?`), number sign (`#`), and asterisk (`*`), brackets must enclose them. The right bracket (`]`) cannot be used within a group to match itself, but it can be used outside a group as an individual character. The character sequence `[]` is considered to be the string literal `""`.

Note that character comparisons and ordering for character lists are dependent on the type of comparisons being used. If binary comparisons are being used, character comparisons and ordering are based on the numeric Unicode values. If text comparisons are being used, character comparisons and ordering are based on the current locale being used on the .NET Framework.

In some languages, special characters in the alphabet represent two separate characters and vice versa. For example, several languages use the character `æ` to represent the characters `a` and `e` when they appear together, while the characters `^` and `O` can be used to represent the character `Ô`. When using text comparisons, the `Like` operator recognizes such cultural equivalences. In that case, an occurrence of the single special character in either pattern or string matches the equivalent two-character sequence in the other string. Similarly, a single special character in pattern enclosed in brackets (by itself, in a list, or in a range) matches the equivalent two-character sequence in the string and vice versa.

In a `Like` expression where both operands are `Nothing` or one operand has an intrinsic conversion to `String` and the other operand is `Nothing`, `Nothing` is treated as if it were the empty string literal `""`.

Operation Type:

	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

11.16 Concatenation Operator

```
ConcatenationOperatorExpression:
    | Expression '&' LineTerminator? Expression
    ;
```

The *concatenation operator* is defined for all of the intrinsic types, including the nullable versions of the intrinsic value types. It is also defined for concatenation between the types mentioned above and `System.DBNull`, which is treated as a `Nothing` string. The concatenation operator converts all of its operands to `String`; in the expression, all conversions to `String` are considered to be widening, regardless of whether strict semantics are used. A `System.DBNull` value is converted to the literal `Nothing` typed as `String`. A nullable value type whose value is `Nothing` is also converted to the literal `Nothing` typed as `String`, rather than throwing a run-time error.

A concatenation operation results in a string that is the concatenation of the two operands in order from left to right. The value `Nothing` is treated as if it were the empty string literal `""`.

Operation Type:

	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

11.17 Logical Operators

The `And`, `Not`, `Or`, and `Xor` operators are called the logical operators.

```
LogicalOperatorExpression:
    | 'Not' Expression
    | Expression 'And' LineTerminator? Expression
    | Expression 'Or' LineTerminator? Expression
```

```
    | Expression 'Xor' LineTerminator? Expression
    ;
```

The logical operators are evaluated as follows:

- For the `Boolean` type:
  - A logical `And` operation is performed on its two operands.
  - A logical `Not` operation is performed on its operand.
  - A logical `Or` operation is performed on its two operands.
  - A logical exclusive-`Or` operation is performed on its two operands.
- For `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, and all enumerated types, the specified operation is performed on each bit of the binary representation of the two operand(s):
  - `And`: The result bit is 1 if both bits are 1; otherwise the result bit is 0.
  - `Not`: The result bit is 1 if the bit is 0; otherwise the result bit is 1.
  - `Or`: The result bit is 1 if either bit is 1; otherwise the result bit is 0.
  - `Xor`: The result bit is 1 if either bit is 1 but not both bits; otherwise the result bit is 0 (that is, 1 `Xor` 0 = 1, 1 `Xor` 1 = 0).
- When the logical operators `And` and `Or` are lifted for the type `Boolean?`, they are extended to encompass three-valued Boolean logic as such:
  - `And` evaluates to true if both operands are true; false if one of the operands is false; `Nothing` otherwise.
  - `Or` evaluates to true if either operand is true; false is both operands are false; `Nothing` otherwise.

For example:

```
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
```

Note. Ideally, the logical operators `And` and `Or` would be lifted using three-valued logic for any type that can be used in a Boolean expression (i.e. a type that implements `IsTrue` and `IsFalse`), in the same way that `AndAlso` and `OrElse` short circuit across any type that can be used in a Boolean expression. Unfortunately, three-valued lifting is only applied to `Boolean?`, so user-defined types that desire three-valued logic must do so manually by defining `And` and `Or` operators for their nullable version.

No overflows are possible from these operations. The enumerated type operators do the bitwise operation on the underlying type of the enumerated type, but the return value is the enumerated type.

Not Operation Type:

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

And, Or, Xor Operation Type:

	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

11.17.1 Short-circuiting Logical Operators

The `AndAlso` and `OrElse` operators are the short-circuiting versions of the `And` and `Or` logical operators.

```
ShortCircuitLogicalOperatorExpression:
    | Expression 'AndAlso' LineTerminator? Expression
    | Expression 'OrElse' LineTerminator? Expression
    ;
```

Because of their short circuiting behavior, the second operand is not evaluated at run time if the operator result is known after evaluating the first operand.

The short-circuiting logical operators are evaluated as follows:

- If the first operand in an `AndAlso` operation evaluates to `False` or returns True from its `IsFalse` operator, the expression returns its first operand. Otherwise, the second operand is evaluated and a logical `And` operation is performed on the two results.
- If the first operand in an `OrElse` operation evaluates to `True` or returns True from its `IsTrue` operator, the expression returns its first operand. Otherwise, the second operand is evaluated and a logical `Or` operation is performed on its two results.

The `AndAlso` and `OrElse` operators are defined for the type `Boolean`, or for any type `T` that overloads the following operators:

```
Public Shared Operator IsTrue(op As T) As Boolean
Public Shared Operator IsFalse(op As T) As Boolean
```

as well as overloading the corresponding `And` or `Or` operator:

```
Public Shared Operator And(op1 As T, op2 As T) As T
Public Shared Operator Or(op1 As T, op2 As T) As T
```

When evaluating the `AndAlso` or `OrElse` operators, the first operand is evaluated only once, and the second operand is either not evaluated or evaluated exactly once. For example, consider the following code:

```
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
```

It prints the following result:

```
And: False True
Or: True False
AndAlso: False
OrElse: True
```

In the lifted form of the `AndAlso` and `OrElse` operators, if the first operand was a null `Boolean?`, then the second operand is evaluated but the result is always a null `Boolean?`.

Operation Type:

	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

11.18 Shift Operators

The binary operators `<<` and `>>` perform bit shifting operations.

```
ShiftOperatorExpression:
    | Expression '<' '<' LineTerminator? Expression
    | Expression '>' '>' LineTerminator? Expression
    ;
```

The operators are defined for the `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong` and `Long` types. Unlike the other binary operators, the result type of a shift operation is determined as if the operator was a unary operator with just the left operand. The type of the right operand must be implicitly convertible to `Integer` and is not used in determining the result type of the operation.

The `<<` operator causes the bits in the first operand to be shifted left the number of places specified by the shift amount. The high-order bits outside the range of the result type are discarded and the low-order vacated bit positions are zero-filled.

The `>>` operator causes the bits in the first operand to be shifted right the number of places specified by the shift amount. The low-order bits are discarded and the high-order vacated bit positions are set to zero if the left operand is positive or to one if negative. If the left operand is of type `Byte`, `UShort`, `UInteger`, or `ULong` the vacant high-order bits are zero-filled.

The shift operators shift the bits of the underlying representation of the first operand by the amount of the second operand. If the value of the second operand is greater than the number of bits in the first operand, or is negative, then the shift amount is computed as `RightOperand And SizeMask` where `SizeMask` is:

LeftOperand Type	SizeMask
`Byte`, `SByte`	7 (`&H7`)
`UShort`, `Short`	15 (`&HF`)
`UInteger`, `Integer`	31 (`&H1F`)
`ULong`, `Long`	63 (`&H3F`)

If the shift amount is zero, the result of the operation is identical to the value of the first operand. No overflows are possible from these operations.

Operation Type:

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

11.19 Boolean Expressions

A Boolean expression is an expression that can be tested to see if it is true or if it is false.

```
BooleanExpression:
    | Expression
    ;
```

A type `T` can be used in a Boolean expression if, in order of preference:

- `T` is `Boolean` or `Boolean?`
- `T` has a widening conversion to `Boolean`
- `T` has a widening conversion to `Boolean?`
- `T` defines two pseudo operators, `IsTrue` and `IsFalse`.
- `T` has a narrowing conversion to `Boolean?` that does not involve a conversion from `Boolean` to `Boolean?`.
- `T` has a narrowing conversion to `Boolean`.

Note. It is interesting to note that if `Option Strict` is off, an expression that has a narrowing conversion to `Boolean` will be accepted without a compile-time error but the language will still prefer an `IsTrue` operator if it exists. This is because `Option Strict` only changes what is and isn't accepted by the language, and never changes the actual meaning of an expression. Thus, `IsTrue` has to always be preferred over a narrowing conversion, regardless of `Option Strict`.

For example, the following class does not define a widening conversion to `Boolean`. As a result, its use in the `If` statement causes a call to the `IsTrue` operator.

```
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
```

If a Boolean expression is typed as or converted to `Boolean` or `Boolean?`, then it is true if the value is `True` and false otherwise.

Otherwise, a Boolean expression calls the `IsTrue` operator and returns `True` if the operator returned `True`; otherwise it is false (but never calls the `IsFalse` operator).

In the following example, `Integer` has a narrowing conversion to `Boolean`, so a null `Integer?` has a narrowing conversion to both `Boolean?` (yielding a null `Boolean`) and to `Boolean` (which throws an exception). The narrowing conversion to `Boolean?` is preferred, and so the value of "`i`" as a Boolean expression is therefore `False`.

```
Dim i As Integer? = Nothing
If i Then Console.WriteLine()
```

11.20 Lambda Expressions

A *lambda expression* defines an anonymous method called a *lambda method*. Lambda methods make it easy to pass "in-line" methods to other methods that take delegate types.

```
LambdaExpression:
    | SingleLineLambda
    | MultiLineLambda
    ;

SingleLineLambda:
    | LambdaModifier* 'Function' ( OpenParenthesis ParameterList? CloseParenthesis  )? Expression
    | 'Sub' ( OpenParenthesis ParameterList? CloseParenthesis  )? Statement
    ;

MultiLineLambda:
    | MultiLineFunctionLambda
    | MultiLineSubLambda
    ;

MultiLineFunctionLambda:
    | LambdaModifier* 'Function' ( OpenParenthesis ParameterList? CloseParenthesis  )? ( 'As'
TypeName  )? LineTerminator
      Block
      'End' 'Function'
    ;

MultiLineSubLambda:
    | LambdaModifier* 'Sub' ( OpenParenthesis ParameterList? CloseParenthesis  )? LineTerminator
      Block
      'End' 'Sub'
    ;

LambdaModifier:
    | 'Async' | 'Iterator'
    ;
```

The example:

```
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
```

will print out:

```
2 4 6 8
```

A lambda expression begins with the optional modifiers `Async` or `Iterator`, followed by the keyword `Function` or `Sub` and a parameter list. Parameters in a lambda expression cannot be declared `Optional` or `ParamArray` and cannot have attributes. Unlike regular methods, omitting a parameter type for a lambda method does not automatically infer `Object`. Instead, when a lambda method is reclassified, the omitted parameter types and `ByRef` modifiers are inferred from the target type. In the previous example, the lambda expression could have been written as `Function(x) x * 2`, and it would have inferred the type of `x` to be `Integer` when the lambda method was used to create an instance of the `IntFunc` delegate type. Unlike local variable inference, if a lambda method parameter omits a type but has an array or nullable name modifier, a compile-time error occurs.

A **regular lambda expression** is one with neither `Async` nor `Iterator` modifiers.

An **iterator lambda expression** is one with the `Iterator` modifier and no `Async` modifier. It must be a function. When it is reclassified to a value, it can only be reclassified to a value of delegate type whose return type is `IEnumerator`, or `IEnumerable`, or `IEnumerator(Of T)` or `IEnumerable(Of T)` for some `T`, and which has no ByRef parameters.

An **async lambda expression** is one with the `Async` modifier and no `Iterator` modifier. An async sub lambda may only be reclassified to a value of sub delegate type with no ByRef parameters. An async function lambda may only be reclassified to a value of function delegate type whose return type is `Task` or `Task(Of T)` for some `T`, and which has no ByRef parameters.

Lambda expressions can either be single-line or multi-line. Single-line `Function` lambda expressions contain a single expression that represents the value returned from the lambda method. Single-line `Sub` lambda expressions contain a single statement without its closing `StatementTerminator`. For example:

```
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
```

Single-line lambda constructs bind less tightly than all other expressions and statements. Thus, for example, "`Function() x + 5`" is equivalent to "`Function() (x+5)`" rather than "`(Function() x) + 5`". To avoid ambiguity, a single-line `Sub` lambda expression may not contain a Dim statement or a label declaration statement. Also, unless it

is enclosed in parentheses, a single-line `Sub` lambda expression may not be immediately followed by a colon ":", a member access operator ".", a dictionary member access operator "!" or an open parenthesis "(". It may not contain any block statement (`With`, `SyncLock`, `If...EndIf`, `While`, `For`, `Do`, `Using`) nor `OnError` nor `Resume`.

Note. In the lambda expression `Function(i) x=i`, the body is interpreted as an *expression* (which tests whether `x` and `i` are equal). But in the lambda expression `Sub(i) x=i`, the body is interpreted as a statement (which assigns `i` to `x`).

A multi-line lambda expression contains a statement block and must end with an appropriate `End` statement (i.e. `End Function` or `End Sub`). As with regular methods, a multi-line lambda method's `Function` or `Sub` statement and `End` statements must be on their own lines. For example:

```
' 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
```

Multi-line `Function` lambda expressions can declare a return type but cannot put attributes on it. If a multi-line `Function` lambda expression does not declare a return type but the return type can be inferred from the context in which the lambda expression is used , then that return type is used. Otherwise the return type of the function is calculated as follows:

- In a regular lambda expression, the return type is the dominant type of the expressions in all the `Return` statements in the statement block.
- In an async lambda expression, the return type is `Task(Of T)` where `T` is the dominant type of the expressions in all the `Return` statements in the statement block.
- In an iterator lambda expression, the return type is `IEnumerable(Of T)` where `T` is the dominant type of the expressions in all the `Yield` statements in the statement block.

For example:

```
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
```

In all cases, if there are no `Return` (respectively `Yield`) statements, or if there is no dominant type among them, and strict semantics are being used, a compile-time error occurs; otherwise the dominant type is implicitly `Object`.

Note that the return type is calculated from all `Return` statements, even if they are not reachable. For example:

```
' Return type is Double
Dim x = Function()
              Return 10
               Return 10.50
            End Function
```

There is no implicit return variable, as there is no name for the variable.

The statement blocks inside multi-line lambda expressions have the following restrictions:

- `On Error` and `Resume` statements are not allowed, although `Try` statements are allowed.
- Static locals cannot be declared in multi-line lambda expressions.
- It is not possible to branch into or out of the statement block of a multi-line lambda expression, although the normal branching rules apply within it. For example:

```
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
```

A lambda expression is roughly equivalent to an anonymous method declared on the containing type. The initial example is roughly equivalent to:

```
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
```

11.20.1 Closures

Lambda expressions have access to all of the variables in scope, including local variables or parameters defined in the containing method and lambda expressions. When a lambda expression refers to a local variable or parameter, the lambda expression captures the variable being referred to into a closure. A closure is an object that lives on the heap instead of on the stack, and when a variable is captured, all references to the variable are redirected to the closure. This enables lambda expressions to continue to refer to local variables and parameters even after the containing method is complete. For example:

```
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
```

is roughly equivalent to:

```
    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
```

A closure captures a new copy of a local variable each time it enters the block in which the local variable is declared, but the new copy is initialized with the value of the previous copy, if any. For example:

```
    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
```

prints

```
1 2 3 4 5 6 7 8 9 10
```

instead of

```
9 9 9 9 9 9 9 9 9 9
```

Because closures have to be initialized when entering a block, it is not allowed to `GoTo` into a block with a closure from outside of that block, although it is allowed to `Resume` into a block with a closure. For example:

```
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
```

Because they cannot be captured into a closure, the following cannot appear inside of a lambda expression:

- Reference parameters.
- Instance expressions (`Me`, `MyClass`, `MyBase`), if the type of `Me` is not a class.

The members of an anonymous type-creation expression, if the lambda expression is part of the expression. For example:

```
' Error: Lambda cannot refer to anonymous type field
Dim x = New With { .a = 12, .b = Function() .a }
```

`ReadOnly` instance variables in instance constructors or `ReadOnly` shared variables in shared constructors where the variables are used in a non-value context. For example:

```
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
```

11.21 Query Expressions

A *query expression* is an expression that applies a series of *query operators* to the elements of a *queryable* collection. For example, the following expression takes a collection of `Customer` objects and returns the names of all the customers in the state of Washington:

```
Dim names = _
    From cust In Customers _
    Where cust.State = "WA" _
    Select cust.Name
```

A query expression must start with a `From` or an `Aggregate` operator and can end with any query operator. The result of a query expression is classified as a value; the result type of the expression depends on the result type of the last query operator in the expression.

```
QueryExpression:
    | FromOrAggregateQueryOperator QueryOperator*
    ;

FromOrAggregateQueryOperator:
    | FromQueryOperator
    | AggregateQueryOperator
    ;

QueryOperator:
    | FromQueryOperator
    | AggregateQueryOperator
    | SelectQueryOperator
    | DistinctQueryOperator
    | WhereQueryOperator
    | OrderByQueryOperator
    | PartitionQueryOperator
    | LetQueryOperator
    | GroupByQueryOperator
    | JoinOrGroupJoinQueryOperator
    ;

JoinOrGroupJoinQueryOperator:
    | JoinQueryOperator
    | GroupJoinQueryOperator
    ;
```

11.21.1 Range Variables

Some query operators introduce a special kind of variable called a *range variable*. Range variables are not real variables; instead, they represent the individual values during the evaluation of the query over the input collections.

```
CollectionRangeVariableDeclarationList:
    | CollectionRangeVariableDeclaration ( Comma CollectionRangeVariableDeclaration  )*
    ;

CollectionRangeVariableDeclaration:
    | Identifier ( 'As' TypeName  )? 'In' LineTerminator? Expression
    ;

ExpressionRangeVariableDeclarationList:
    | ExpressionRangeVariableDeclaration ( Comma ExpressionRangeVariableDeclaration  )*
    ;

ExpressionRangeVariableDeclaration:
```

```
    | Identifier ( 'As' TypeName  )? Equals Expression
    ;
```

Range variables are scoped from the introducing query operator to the end of a query expression, or to a query operator such as `Select` that hides them. For example, in the following query

```
Dim waCusts = _
    From cust As Customer In Customers _
    Where cust.State = "WA"
```

the `From` query operator introduces a range variable `cust` typed as `Customer` that represents each customer in the `Customers` collection. The following `Where` query operator then refers to the range variable `cust` in the filter expression to determine whether to filter an individual customer out of the resulting collection.

There are two types of range variables: *collection range variables* and *expression range variables*. Collection range variables take their values from the elements of the collections being queried. The collection expression in a collection range variable declaration must be classified as a value whose type is queryable. If the type of a collection range variable is omitted, it is inferred to be the element type of the collection expression, or `Object` if the collection expression does not have an element type (i.e. only defines a `Cast` method). If the collection expression is not queryable (i.e. the element type of the collection cannot be inferred), a compile-time error results.

An expression range variable is a range variable whose value is calculated by an expression rather than a collection. In the following example, the `Select` query operator introduces an expression range variable named `cityState` calculated from two fields:

```
Dim cityStates = _
    From cust As Customer In Customers _
    Select cityState = cust.City & "," & cust.State _
    Where cityState.Length() < 10
```

An expression range variable is not required to reference another range variable, although such a variable may be of dubious value. The expression assigned to an expression range variable must be classified as a value and must be implicitly convertible to the type of the range variable, if given.

Only in a Let operator may an expression range variable have its type specified. In other operators, or if its type is not specified, then local variable type inference is used to determine the type of the range variable.

A range variable must follow the rules for declaring local variables in respect to shadowing. Thus, a range variable cannot hide the name of a local variable or parameter in the enclosing method or another range variable (unless the query operator specifically hides all current range variables in scope).

11.21.2 Queryable Types

Query expressions are implemented by translating the expression into calls to well-known methods on a collection type. These well-defined methods define the element type of the queryable collection as well as the result types of query operators executed on the collection. Each query operator specifies the method or methods that the query operator is generally translated into, although the specific translation is implementation dependent. The methods are given in the specification using a general format that looks like:

```
Function Select(selector As Func(Of T, R)) As CR
```

The following applies to the methods:

- The method must be an instance or extension member of the collection type and must be accessible.
- The method may be generic, provided that is possible to infer all type arguments.
- The method may be overloaded, in which case overload resolution is used to determine the exactly method to use.
- Another delegate type may be used in place of the delegate `Func` type, provided that it has the same signature, including return type, as the matching `Func` type.

- The type `System.Linq.Expressions.Expression(Of D)` may be used in place of the delegate `Func` type, provided that `D` is a delegate type that has the same signature, including return type, as the matching `Func` type.
- The type `T` represents the element type of the input collection. All of the methods defined by a collection type must have the same input element type for the collection type to be queryable.
- The type `S` represents the element type of the second input collection in the case of query operators that perform joins.
- The type `K` represents a key type in the case of query operators that have a set of range variables that act as keys.
- The type `N` represents a type that is used as a numeric type (although it could still be a user-defined type and not an intrinsic numeric type).
- The type `B` represents a type that can be used in a Boolean expression.
- The type `R` represents the element type of the result collection, if the query operator produces a result collection. `R` depends on the number of range variables in scope at the conclusion of the query operator. If a single range variable is in scope, then `R` is the type of that range variable. In the example

  ```
  Dim custNames = From c In Customers
                  Select c.Name
  ```

  the result of the query will be a collection type with an element type of `String`. If multiple range variables are in scope, then `R` is an anonymous type that contains all of the range variables in scope as `Key` fields. In the example:

  ```
  Dim custNames = From c In Customers, o In c.Orders
                  Select Name = c.Name, ProductName = o.ProductName
  ```

  the result of the query will be a collection type with an element type of an anonymous type with a read-only property named `Name` of type `String` and a read-only property named `ProductName` of type `String`.

  Within a query expression, anonymous types generated to contain range variables are *transparent*, which means that range variables are always available without qualification. For example, in the previous example the range variables `c` and `o` could be accessed without qualification in the `Select` query operator, even though the input collection's element type was an anonymous type.
- The type `CX` represents a collection type, not necessarily the input collection type, whose element type is some type `X`.

A queryable collection type must satisfy one of the following conditions, in order of preference:

- It must define a conforming `Select` method.
- It must have have one of the following methods

  ```
  Function AsEnumerable() As CT
  Function AsQueryable() As CT
  ```

  which can be called to obtain a queryable collection. If both methods are provided, `AsQueryable` is preferred over `AsEnumerable`.
- It must have a method

  ```
  Function Cast(Of T)() As CT
  ```

  which can be called with the type of the range variable to produce a queryable collection.

Because determining the element type of a collection occurs independently of an actual method invocation, the applicability of specific methods cannot be determined. Thus, when determining the element type of a collection if

there are instance methods that match well-known methods, then any extension methods that match well-known methods are ignored.

Query operator translation occurs in the order in which the query operators occur in the expression. It is not necessary for a collection object to implement all of the methods needed by all the query operators, although every collection object must at least support the `Select` query operator. If a needed method is not present, a compile-time error occurs. When binding well-known method names, non-methods are ignored for the purpose of multiple inheritance in interfaces and extension method binding, although shadowing semantics still apply. For example:

```
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
```

11.21.3 Default Query Indexer

Every queryable collection type whose element type is `T` and does not already have a default property is considered to have a default property of the following general form:

```
Public ReadOnly Default Property Item(index As Integer) As T
    Get
        Return Me.ElementAtOrDefault(index)
    End Get
End Property
```

The default property can only be referred to using the default property access syntax; the default property cannot be referred to by name. For example:

```
Dim customers As IEnumerable(Of Customer) = ...
Dim customerThree = customers(2)

' Error, no such property
Dim customerFour = customers.Item(4)
```

If the collection type does not have an `ElementAtOrDefault` member, a compile-time error will occur.

11.21.4 From Query Operator

The `From` query operator introduces a collection range variable that represents the individual members of a collection to be queried.

```
FromQueryOperator:
    | LineTerminator? 'From' LineTerminator? CollectionRangeVariableDeclarationList
    ;
```

For example, the query expression:

```
From c As Customer In Customers ...
```

can be thought of as equivalent to

```
For Each c As Customer In Customers
        ...
Next c
```

When a `From` query operator declares multiple collection range variables or is not the first `From` query operator in the query expression, each new collection range variable is *cross joined* to the existing set of range variables. The result is that the query is evaluated over the cross-product of all the elements in the joined collections. For example, the expression:

```
From c In Customers _
From e In Employees _
...
```

can be thought of as equivalent to:

```
For Each c In Customers
    For Each e In Employees
            ...
    Next e
Next c
```

and is exactly equivalent to:

```
From c In Customers, e In Employees ...
```

The range variables introduced in previous query operators can be used in a later `From` query operator. For example, in the following query expression the second `From` query operator refers to the value of the first range variable:

```
From c As Customer In Customers _
From o As Order In c.Orders _
Select c.Name, o
```

Multiple range variables in a `From` query operator or multiple `From` query operators are only supported if the collection type contains one or both of the following methods:

```
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
```

The code

```
Dim xs() As Integer = ...
Dim ys() As Integer = ...
Dim zs = From x In xs, y In ys ...
```

is generally translated to

```
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})...
```

Note. `From` is not a reserved word.

11.21.5 Join Query Operator

The `Join` query operator joins existing range variables with a new collection range variable, producing a single collection whose elements have been joined together based on an equality expression.

```
JoinQueryOperator:
    | LineTerminator? 'Join' LineTerminator? CollectionRangeVariableDeclaration
      JoinOrGroupJoinQueryOperator? LineTerminator? 'On' LineTerminator? JoinConditionList
    ;

JoinConditionList:
    | JoinCondition ( 'And' LineTerminator? JoinCondition  )*
    ;

JoinCondition:
    | Expression 'Equals' LineTerminator? Expression
    ;
```

For example:

```
Dim customersAndOrders = _
    From cust In Customers _
    Join ord In Orders On cust.ID Equals ord.CustomerID
```

The equality expression is more restricted than a regular equality expression:

- Both expressions must be classified as a value.
- Both expressions must reference at least one range variable.
- The range variable declared in the join query operator must be referenced by one of the expressions, and that expression must not reference any other range variables.

If the types of the two expressions are not the exact same type, then

- If the equality operator is defined for the two types, both expressions are implicitly convertible to it, and it is not `Object`, then convert both expressions to that type.
- Otherwise, if there is a dominant type that both expressions can be implicitly converted to, then convert both expressions to that type.
- Otherwise, a compile-time error occurs.

The expressions are compared using hash values (i.e. by calling `GetHashCode()`) rather than by using equality operators for efficiency. A `Join` query operator may do multiple joins or equality conditions in the same operator. A `Join` query operator is only supported if the collection type contains a method:

```
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
```

The code

```
Dim xs() As Integer = ...
Dim ys() As Integer = ...
Dim zs = From x In xs _
           Join y In ys On x Equals y _
           ...
```

is generally translated to

```
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})...
```

Note. `Join`, `On` and `Equals` are not reserved words.

11.21.6 Let Query Operator

The `Let` query operator introduces an expression range variable. This allows calculating an intermediate value once that will be used multiple times in later query operators.

```
LetQueryOperator:
    | LineTerminator? 'Let' LineTerminator? ExpressionRangeVariableDeclarationList
    ;
```

For example:

```
Dim taxedPrices = _
    From o In Orders _
    Let tax = o.Price * 0.088 _
    Where tax > 3.50 _
    Select o.Price, tax, total = o.Price + tax
```

can be thought of as equivalent to:

```
For Each o In Orders
    Dim tax = o.Price * 0.088
    ...
Next o
```

A `Let` query operator is only supported if the collection type contains a method:

```
Function Select(selector As Func(Of T, R)) As CR
```

The code

```
Dim xs() As Integer = ...
Dim zs = From x In xs _
           Let y = x * 10 _
           ...
```

is generally translated to

```
Dim xs() As Integer = ...
Dim zs = _
    xs.Select(Function(x As Integer) New With {x, .y = x * 10})...
```

11.21.7 Select Query Operator

The `Select` query operator is like the `Let` query operator in that it introduces expression range variables; however, a `Select` query operator hides the currently available range variables instead of adding to them. Also, the type of an expression range variable introduced by a `Select` query operator is always inferred using local variable type inference rules; an explicit type cannot be specified, and if no type can be inferred, a compile-time error occurs.

```
SelectQueryOperator:
    | LineTerminator? 'Select' LineTerminator? ExpressionRangeVariableDeclarationList
    ;
```

For example, in the query:

```
Dim smiths = _
    From cust In Customers _
    Select name = cust.name _
    Where name.EndsWith("Smith")
```

the `Where` query operator only has access to the `name` range variable introduced by the `Select` operator; if the `Where` operator had tried to reference `cust`, a compile-time error would have occurred.

Instead of explicitly specifying the names of the range variables, a `Select` query operator can infer the names of the range variables, using the same rules as anonymous type object creation expressions. For example:

```
Dim custAndOrderNames = _
      From cust In Customers, ord In cust.Orders _
      Select cust.name, ord.ProductName _
        Where name.EndsWith("Smith")
```

If the name of the range variable is not supplied and a name cannot be inferred, a compile-time error occurs. If the `Select` query operator contains only a single expression, no error occurs if a name for that range variable cannot be inferred but the range variable is nameless. For example:

```
Dim custAndOrderNames = _
      From cust In Customers, ord In cust.Orders _
      Select cust.Name & " bought " & ord.ProductName _
        Take 10
```

If there is an ambiguity in a `Select` query operator between assigning a name to a range variable and an equality expression, the name assignment is preferred. For example:

```
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)
```

Each expression in the `Select` query operator must be classified as a value. A `Select` query operator is supported only if the collection type contains a method:

```
Function Select(selector As Func(Of T, R)) As CR
```

The code

```
Dim xs() As Integer = ...
Dim zs = From x In xs _
           Select x, y = x * 10 _
           ...
```

is generally translated to

```
Dim xs() As Integer = ...
Dim zs = _
    xs.Select(Function(x As Integer) New With {x, .y = x * 10})...
```

11.21.8 Distinct Query Operator

The `Distinct` query operator restricts the values in a collection only to those with distinct values, as determined by comparing the element type for equality.

```
DistinctQueryOperator:
    | LineTerminator? 'Distinct' LineTerminator?
    ;
```

For example, the query:

```
Dim distinctCustomerPrice = _
    From cust In Customers, ord In cust.Orders _
    Select cust.Name, ord.Price _
    Distinct
```

will only return one row for each distinct pairing of customer name and order price, even if the customer has multiple orders with the same price. A `Distinct` query operator is supported only if the collection type contains a method:

```
Function Distinct() As CT
```

The code

```
Dim xs() As Integer = ...
Dim zs = From x In xs _
            Distinct _
            ...
```

is generally translated to

```
Dim xs() As Integer = ...
Dim zs = xs.Distinct()...
```

Note. `Distinct` is not a reserved word.

11.21.9 Where Query Operator

The `Where` query operator restricts the values in a collection to those that satisfy a given condition.

```
WhereQueryOperator:
    | LineTerminator? 'Where' LineTerminator? BooleanExpression
    ;
```

A `Where` query operator takes a Boolean expression that is evaluated for each set of range variable values; if the value of the expression is true, then the values appear in the output collection, otherwise the values are skipped. For example, the query expression:

```
From cust In Customers, ord In Orders _
Where cust.ID  ord.CustomerID _
...
```

can be thought of as equivalent to the nested loop

```
For Each cust In Customers
    For Each ord In Orders
            If cust.ID = ord.CustomerID Then
                ...
            End If
    Next ord
Next cust
```

A `Where` query operator is supported only if the collection type contains a method:

```
Function Where(predicate As Func(Of T, B)) As CT
```

The code

```
Dim xs() As Integer = ...
Dim zs = From x In xs _
            Where x < 10 _
            ...
```

is generally translated to

```
Dim xs() As Integer = ...
Dim zs = _
    xs.Where(Function(x As Integer) x < 10)...
```

Note. `Where` is not a reserved word.

11.21.10 Partition Query Operators

```
PartitionQueryOperator:
    | LineTerminator? 'Take' LineTerminator? Expression
    | LineTerminator? 'Take' 'While' LineTerminator? BooleanExpression
    | LineTerminator? 'Skip' LineTerminator? Expression
    | LineTerminator? 'Skip' 'While' LineTerminator? BooleanExpression
    ;
```

The `Take` query operator results in the first `n` elements of a collection. When used with the `While` modifier, the `Take` operator results in the first `n` elements of a collection that satisfy a Boolean expression. The `Skip` operator skips the first `n` elements of a collection and then returns the remainder of the collection. When used in conjunction with the `While` modifier, the `Skip` operator skips the first `n` elements of a collection that satisfy a Boolean expression and then returns the rest of the collection. The expressions in a `Take` or `Skip` query operator must be classified as a value.

A `Take` query operator is supported only if the collection type contains a method:

```
Function Take(count As N) As CT
```

A `Skip` query operator is supported only if the collection type contains a method:

```
Function Skip(count As N) As CT
```

A `Take While` query operator is supported only if the collection type contains a method:

```
Function TakeWhile(predicate As Func(Of T, B)) As CT
```

A `Skip While` query operator is supported only if the collection type contains a method:

```
Function SkipWhile(predicate As Func(Of T, B)) As CT
```

The code

```
Dim xs() As Integer = ...
Dim zs = From x In xs _
            Skip 10 _
            Take 5 _
            Skip While x < 10 _
            Take While x > 5 _
            ...
```

is generally translated to

```
Dim xs() As Integer = ...
Dim zs = _
    xs.Skip(10). _
        Take(5). _
        SkipWhile(Function(x) x < 10). _
        TakeWhile(Function(x) x > 5)...
```

Note. `Take` and `Skip` are not reserved words.

11.21.11 Order By Query Operator

The `Order By` query operator orders the values that appear in the range variables.

```
OrderByQueryOperator:
    | LineTerminator? 'Order' 'By' LineTerminator? OrderExpressionList
    ;
```

```
OrderExpressionList:
    | OrderExpression ( Comma OrderExpression  )*
    ;

OrderExpression:
    | Expression Ordering?
    ;

Ordering:
    | 'Ascending' | 'Descending'
    ;
```

An `Order By` query operator takes expressions that specify the key values that should be used to order the iteration variables. For example, the following query returns products sorted by price:

```
Dim productsByPrice = _
    From p In Products _
    Order By p.Price _
    Select p.Name
```

An ordering can be marked as `Ascending`, in which case smaller values come before larger values, or `Descending`, in which case larger values come before smaller values. The default for an ordering if none is specified is `Ascending`. For example, the following query returns products sorted by price with the most expensive product first:

```
Dim productsByPriceDesc = _
    From p In Products _
    Order By p.Price Descending _
    Select p.Name
```

The `Order By` query operator may specify multiple expressions for ordering, in which case the collection is ordered in a nested manner. For example, the following query orders customers by state, then by city within each state and then by ZIP code within each city:

```
Dim customersByLocation = _
    From c In Customers _
    Order By c.State, c.City, c.ZIP _
    Select c.Name, c.State, c.City, c.ZIP
```

The expressions in an `Order By` query operator must be classified as a value. An `Order By` query operator is supported only if the collection type contains one or both of the following methods:

```
Function OrderBy(keySelector As Func(Of T, K)) As CT
Function OrderByDescending(keySelector As Func(Of T, K)) As CT
```

The return type `CT` must be an *ordered collection*. An ordered collection is a collection type that contains one or both of the methods:

```
Function ThenBy(keySelector As Func(Of T, K)) As CT
Function ThenByDescending(keySelector As Func(Of T, K)) As CT
```

The code

```
Dim xs() As Integer = ...
Dim zs = From x In xs _
            Order By x Ascending, x Mod 2 Descending _
            ...
```

is generally translated to

```
Dim xs() As Integer = ...
Dim zs = _
    xs.OrderBy(Function(x) x).ThenByDescending(Function(x) x Mod 2)...
```

Note. Because query operators simply map syntax to methods that implement a particular query operation, order preservation is not dictated by the language and is determined by the implementation of the operator itself. This is very similar to user-defined operators in that the implementation to overload the addition operator for a user-defined numeric type may not perform anything resembling an addition. Of course, to preserve predictability, implementing something that does not match user expectations is not recommended.

Note. `Order` and `By` are not reserved words.

11.21.12 Group By Query Operator

The `Group By` query operator groups the range variables in scope based on one or more expressions, and then produces new range variables based on those groupings.

```
GroupByQueryOperator:
    | LineTerminator? 'Group' ( LineTerminator? ExpressionRangeVariableDeclarationList  )?
      LineTerminator? 'By' LineTerminator? ExpressionRangeVariableDeclarationList
      LineTerminator? 'Into' LineTerminator? ExpressionRangeVariableDeclarationList
    ;
```

For example, the following query groups all customers by `State`, and then computes the count and average age of each group:

```
Dim averageAges = _
    From cust In Customers _
    Group By cust.State _
    Into Count(), Average(cust.Age)
```

The `Group By` query operator has three clauses: the optional `Group` clause, the `By` clause, and the `Into` clause. The `Group` clause has the same syntax and effect as a `Select` query operator, except that it only affects the range variables available in the `Into` clause and not the `By` clause. For example:

```
Dim averageAges = _
    From cust In Customers _
    Group cust.Age By cust.State _
    Into Count(), Average(Age)
```

The `By` clause declares expression range variables that are used as key values in the grouping operation. The `Into` clause allows the declaration of expression range variables that calculate aggregations over each of the groups formed by the `By` clause. Within the `Into` clause, the expression range variable can only be assigned an expression which is a method invocation of an *aggregate function*. An aggregate function is a function on the collection type of the group (which may not necessarily be the same collection type of the original collection) which looks like either of the following methods:

```
Function _name_() As _type_
Function _name_(selector As Func(Of T, R)) As R
```

If an aggregate function takes a delegate argument, then the invocation expression can have an argument expression that must be classified as a value. The argument expression can use the range variables that are in scope; within the call to an aggregate function, those range variables represent the values in the group being formed, not all of the values in the collection. For example, in the original example in this section the `Average` function calculates the average of the customers' ages per state rather than for all of the customers together.

All collection types are considered to have the aggregate function `Group` defined on it, which takes no parameters and simply returns the group. Other standard aggregate functions that a collection type may provide are:

`Count` and `LongCount`, which return the count of the elements in the group or the count of the elements in the group that satisfy a Boolean expression. `Count` and `LongCount` are supported only if the collection type contains one of the methods:

```
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`, which returns the sum of an expression across all the elements in the group. `Sum` is supported only if the collection type contains one of the methods:

```
Function Sum() As N
Function Sum(selector As Func(Of T, N)) As N
```

`Min` which returns the minimum value of an expression across all the elements in the group. `Min` is supported only if the collection type contains one of the methods:

```
Function Min() As N
Function Min(selector As Func(Of T, N)) As N
```

`Max`, which returns the maximum value of an expression across all the elements in the group. `Max` is supported only if the collection type contains one of the methods:

```
Function Max() As N
Function Max(selector As Func(Of T, N)) As N
```

`Average`, which returns the average of an expression across all the elements in the group. `Average` is supported only if the collection type contains one of the methods:

```
Function Average() As N
Function Average(selector As Func(Of T, N)) As N
```

`Any`, which determines whether a group contains members or if a Boolean expression is true for any element in the group. `Any` returns a value that can be used in a Boolean expression and is supported only if the collection type contains one of the methods:

```
Function Any() As B
Function Any(predicate As Func(Of T, B)) As B
```

`All`, which determines whether a Boolean expression is true for all elements in the group. `All` returns a value that can be used in a Boolean expression and is supported only if the collection type contains a method:

```
Function All(predicate As Func(Of T, B)) As B
```

After a `Group By` query operator, the range variables previously in scope are hidden, and the range variables introduced by the `By` and `Into` clauses are available. A `Group By` query operator is supported only if the collection type contains the method:

```
Function GroupBy(keySelector As Func(Of T, K), _
                 resultSelector As Func(Of K, CT, R)) As CR
```

Range variable declarations in the `Group` clause are supported only if the collection type contains the method:

```
Function GroupBy(keySelector As Func(Of T, K), _
                 elementSelector As Func(Of T, S), _
                 resultSelector As Func(Of K, CS, R)) As CR
```

The code

```
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) _
           ...
```

is generally translated to

```
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)})...
```

Note. `Group`, `By`, and `Into` are not reserved words.

11.21.13 Aggregate Query Operator

The `Aggregate` query operator performs a similar function as the `Group By` operator, except it allows aggregating over groups that have already been formed. Because the group has already been formed, the `Into` clause of an `Aggregate` query operator does not hide the range variables in scope (in this way, `Aggregate` is more like a `Let`, and `Group By` is more like a `Select`).

```
AggregateQueryOperator:
    | LineTerminator? 'Aggregate' LineTerminator? CollectionRangeVariableDeclaration QueryOperator*
      LineTerminator? 'Into' LineTerminator? ExpressionRangeVariableDeclarationList
    ;
```

For example, the following query aggregates the total of all the orders placed by customers in Washington:

```
Dim orderTotals = _
    From cust In Customers _
    Where cust.State = "WA" _
    Aggregate order In cust.Orders _
    Into Sum(order.Total)
```

The result of this query is a collection whose element type is an anonymous type with a property named `cust` typed as `Customer` and a property named `Sum` typed as `Integer`.

Unlike `Group By`, additional query operators can be placed between the `Aggregate` and `Into` clauses. Between an `Aggregate` clause and the end of the `Into` clause, all range variables in scope, including those declared by the `Aggregate` clause can be used. For example, the following query aggregates the sum total of all the orders placed by customers in Washington before 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)
```

The `Aggregate` operator can also be used to start a query expression. In this case, the result of the query expression will be the single value computed by the `Into` clause. For example, the following query calculates the sum of all the order totals before January 1st, 2006:

```
Dim ordersTotal = _
    Aggregate order In Orders _
    Where order.Date <= #01/01/2006# _
    Into Sum(order.Total)
```

The result of the query is a single `Integer` value. An `Aggregate` query operator is always available (although the aggregate function must be also be available for the expression to be valid). The code

```
Dim xs() As Integer = ...
Dim zs = _
    Aggregate x In xs _
    Where x < 5 _
    Into Sum()
```

is generally translated to

```
Dim xs() As Integer = ...
Dim zs = _
    xs.Where(Function(x) x < 5).Sum()
```

Note. `Aggregate` and `Into` are not reserved words.

11.21.14 Group Join Query Operator

The `Group Join` query operator combines the functions of the `Join` and `Group By` query operators into a single operator. `Group Join` joins two collections based on matching keys extracted from the elements, grouping together all of the elements on the right side of the join that match a particular element on the left side of the join. Thus, the operator produces a set of hierarchical results.

```
GroupJoinQueryOperator:
    | LineTerminator? 'Group' 'Join' LineTerminator? CollectionRangeVariableDeclaration
      JoinOrGroupJoinQueryOperator? LineTerminator? 'On' LineTerminator? JoinConditionList
      LineTerminator? 'Into' LineTerminator? ExpressionRangeVariableDeclarationList
    ;
```

For example, the following query produces elements that contain a single customer's name, a group of all of their orders, and the total amount of all of those orders:

```
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
```

The result of the query is a collection whose element type is an anonymous type with three properties: `Name`, typed as `String`, `Orders` typed as a collection whose element type is `Order`, and `OrdersTotal`, typed as `Integer`. A `Group Join` query operator is supported only if the collection type contains the method:

```
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
```

The code

```
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 _
            ...
```

is generally translated to

```
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})...
```

Note. `Group`, `Join`, and `Into` are not reserved words.

11.22 Conditional Expressions

A conditional `If` expression tests an expression and returns a value.

```
ConditionalExpression:
    | 'If' OpenParenthesis BooleanExpression Comma Expression Comma Expression CloseParenthesis
    | 'If' OpenParenthesis Expression Comma Expression CloseParenthesis
    ;
```

Unlike the `IIF` runtime function, however, a conditional expression only evaluates its operands if necessary. Thus, for example, the expression `If(c Is Nothing, c.Name, "Unknown")` will not throw an exception if the value of `c` is `Nothing`. The conditional expression has two forms: one that takes two operands and one that takes three operands.

If three operands are provided, all three expressions must be classified as values, and the first operand must be a Boolean expression. If the result is of the expression is true, then the second expression will be the result of the operator, otherwise the third expression will be the result of the operator. The result type of the expression is the dominant type between the types of the second and third expression. If there is no dominant type, then a compile-time error occurs.

If two operands are provided, both of the operands must be classified as values, and the first operand must be either a reference type or a nullable value type. The expression `If(x, y)` is then evaluated as if was the expression `If(x IsNot Nothing, x, y)`, with two exceptions. First, the first expression is only ever evaluated once, and second, if the second operand's type is a non-nullable value type and the first operand's type is, the `?` is removed from the type of the first operand when determining the dominant type for the result type of the expression. For example:

```
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
```

In both forms of the expression, if an operand is `Nothing`, its type is not used to determine the dominant type. In the case of the expression `If(<expression>, Nothing, Nothing)`, the dominant type is considered to be `Object`.

11.23 XML Literal Expressions

An XML literal expression represents an XML (eXtensible Markup Language) 1.0 value.

```
XMLLiteralExpression:
    | XMLDocument
    | XMLElement
    | XMLProcessingInstruction
    | XMLComment
    | XMLCDATASection
    ;
```

The result of an XML literal expression is a value typed as one of the types from the `System.Xml.Linq` namespace. If the types in that namespace are not available, then an XML literal expression will cause a compile-time error. The values are generated through constructor calls translated from the XML literal expression. For example, the code:

```
Dim book As System.Xml.Linq.XElement = _
    <book title="My book"></book>
```

is roughly equivalent to the code:

```
Dim book As System.Xml.Linq.XElement = _
    New System.Xml.Linq.XElement( _
        "book", _
        New System.Xml.Linq.XAttribute("title", "My book"))
```

An XML literal expression can take the form of an XML document, an XML element, an XML processing instruction, an XML comment, or a CDATA section.

Note. This specification contains only enough of a description of XML to describe the behavior of the Visual Basic language. More information on XML can be found at http://www.w3.org/TR/REC-xml/.

11.23.1 Lexical rules

```
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
    ;
```

XML literal expressions are interpreted using the lexical rules of XML instead of the lexical rules of regular Visual Basic code. The two sets of rules generally differ in the following ways:

- White space is significant in XML. As a result, the grammar for XML literal expressions explicitly states where white space is allowed. Whitespace is not preserved, except when it occurs in the context of character data within an element. For example:

```
' The following element preserves no whitespace
Dim e1 = _
    <customer>
        <name>Bob</>
    </>

' The following element preserves all of the whitespace
Dim e2 = _
    <customer>
        Bob
    </>
```

- XML end-of-line whitespace is normalized according to the XML specification.
- XML is case-sensitive. Keywords must match casing exactly, or else a compile-time error will occur.
- Line terminators are considered white space in XML. As a result, no line-continuation characters are needed in XML literal expressions.
- XML does not accept full-width characters. If full-width characters are used, a compile-time error will occur.

11.23.2 Embedded expressions

XML literal expressions can contain *embedded expressions*. An embedded expression is a Visual Basic expression that is evaluated and used to fill in one or more values at the location of embedded expression.

```
XMLEmbeddedExpression:
    | '<' '%' '=' LineTerminator? Expression LineTerminator? '%' '>'
    ;
```

For example, the following code places the string `John Smith` as the value of the XML element:

```
Dim name as String = "John Smith"
Dim element As System.Xml.Linq.XElement = <customer><%= name %></customer>
```

Expressions can be embedded in a number of contexts. For example, the following code produces an element named `customer`:

```
Dim name As String = "customer"
Dim element As System.Xml.Linq.XElement = <<%= name %>>John Smith</>
```

Each context where an embedded expression can be used specifies the types that will be accepted. When within the context of the expression part of an embedded expression, the normal lexical rules for Visual Basic code still apply so, for example, line continuations must be used:

```
        ' Visual Basic expression uses line continuation, XML does not
        Dim element As System.Xml.Linq.XElement = _
            <<%= name & _
                  name %>>John
                             Smith</>
```

11.23.3 XML Documents

```
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
    ;
```

An XML document results in a value typed as `System.Xml.Linq.XDocument`. Unlike the XML 1.0 specification, XML documents in XML literal expressions are required to specify the XML document prologue; XML literal expressions without the XML document prologue are interpreted as their individual entity. For example:

```
Dim doc As System.Xml.Linq.XDocument = _
    <?xml version="1.0"?>
    <?instruction?>
    <customer>Bob</>

Dim pi As System.Xml.Linq.XProcessingInstruction = _
    <?instruction?>
```

An XML document can contain an embedded expression whose type can be any type; at runtime, however, the object must satisfy the requirements of the `XDocument` constructor or a run-time error will occur.

Unlike regular XML, XML document expressions do not support DTDs (Document Type Declarations). Also, the encoding attribute, if supplied, will be ignored since the encoding of the Xml literal expression is always the same as the encoding of the source file itself.

Note. Although the encoding attribute is ignored, it is still valid attribute in order to maintain the ability to include any valid Xml 1.0 documents in source code.

11.23.4 XML Elements

```
XMLElement:
    | XMLEmptyElement
    | XMLElementStart XMLContent XMLElementEnd
    ;

XMLEmptyElement:
    | '<' XMLQualifiedNameOrExpression XMLAttribute* XMLWhitespace? '/' '>'
    ;

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+ ';'
    ;
```

An XML element results in a value typed as `System.Xml.Linq.XElement`. Unlike regular XML, XML elements can omit the name in the closing tag and the current most-nested element will be closed. For example:

```
Dim name = <name>Bob</>
```

Attribute declarations in an XML element result in values typed as `System.Xml.Linq.XAttribute`. Attribute values are normalized according to the XML specification. When the value of an attribute is `Nothing` the attribute will not be created, so the attribute value expression will not have to be checked for `Nothing`. For example:

```
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 %>/>
```

XML elements and attributes can contain nested expressions in the following places:

The name of the element, in which case the embedded expression must be a value of a type implicitly convertible to `System.Xml.Linq.XName`. For example:

```
Dim name = <<%= "name" %>>Bob</>
```

The name of an attribute of the element, in which case the embedded expression must be a value of a type implicitly convertible to `System.Xml.Linq.XName`. For example:

```
Dim name = <name <%= "length" %>="3">Bob</>
```

The value of an attribute of the element, in which case the embedded expression can be a value of any type. For example:

```
Dim name = <name length=<%= 3 %>>Bob</>
```

An attribute of the element, in which case the embedded expression can be a value of any type. For example:

```
Dim name = <name <%= new XAttribute("length", 3) %>>Bob</>
```

The content of the element, in which case the embedded expression can be a value of any type. For example:

```
Dim name = <name><%= "Bob" %></>
```

If the type of the embedded expression is `Object()`, the array will be passed as a paramarray to the `XElement` constructor.

11.23.5 XML Namespaces

XML elements can contain XML namespace declarations, as defined by the XML namespaces 1.0 specification.

```
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
    ;
```

The restrictions on defining the namespaces `xml` and `xmlns` are enforced and will produce compile-time errors. XML namespace declarations cannot have an embedded expression for their value; the value supplied must be a non-empty string literal. For example:

```
' 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</>
```

Note. This specification contains only enough of a description of XML namespace to describe the behavior of the Visual Basic language. More information on XML namespaces can be found at http://www.w3.org/TR/REC-xml-names/.

XML element and attribute names can be qualified using namespace names. Namespaces are bound as in regular XML, with the exception that any namespace imports declared at the file level are considered to be declared in a context enclosing the declaration, which is itself enclosed by any namespace imports declared by the compilation environment. If a namespace name cannot be found, a compile-time error occurs. For example:

```
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
```

XML namespaces declared in an element do not apply to XML literals inside embedded expressions. For example:

```
' Error: Namespace prefix 'db' is not declared
Dim customer = _
    <db:customer xmlns:db="http://example.org/database">
        <%= <db:customer>Bob</> %>
    </>
```

Note. This is because the embedded expression can be anything, including a function call. If the function call contained an XML literal expression, it is not clear whether programmers would expect the XML namespace to be applied or ignored.

11.23.6 XML Processing Instructions

An XML processing instruction results in a value typed as `System.Xml.Linq.XProcessingInstruction`. XML processing instructions cannot contain embedded expressions, as they are valid syntax within the processing instruction.

```
XMLProcessingInstruction:
    | '<' '?' XMLProcessingTarget ( XMLWhitespace XMLProcessingValue?  )? '?' '>'
    ;
```

```
XMLProcessingTarget:
    | Any XMLName except a casing permutation of the string "xml"
    ;

XMLProcessingValue:
    | Any XMLString that does not contain a question-mark followed by ">"
    ;
```

11.23.7 XML Comments

An XML comment results in a value typed as `System.Xml.Linq.XComment`. XML comments cannot contain embedded expressions, as they are valid syntax within the comment.

```
XMLComment:
    | '<' '!' '-' '-' XMLCommentCharacter* '-' '-' '>'
    ;

XMLCommentCharacter:
    | Any XMLCharacter except dash (0x002D)
    | '-' Any XMLCharacter except dash (0x002D)
    ;
```

11.23.8 CDATA sections

A CDATA section results in a value typed as `System.Xml.Linq.XCData`. CDATA sections cannot contain embedded expressions, as they are valid syntax within the CDATA section.

```
XMLCDATASection:
    | '<' '!' ( 'CDATA' '[' XMLCDATASectionString? ']'  )? '>'
    ;

XMLCDATASectionString:
    | Any XMLString that does not contain the string "]]>"
    ;
```

11.24 XML Member Access Expressions

An XML member access expression accesses the members of an XML value.

```
XMLMemberAccessExpression:
    | Expression '.' LineTerminator? '<' XMLQualifiedName '>'
    | Expression '.' LineTerminator? '@' LineTerminator? '<' XMLQualifiedName '>'
    | Expression '.' LineTerminator? '@' LineTerminator? IdentifierOrKeyword
    | Expression '.' '.' '.' LineTerminator? '<' XMLQualifiedName '>'
    ;
```

There are three types of XML member access expressions:

- *Element access*, in which an XML name follows a single dot. For example:

  ```
  Dim customer = _
      <customer>
          <name>Bob</>
      </>
  Dim customerName = customer.<name>.Value
  ```

  Element access maps to the function:

```
Function Elements(name As System.Xml.Linq.XName) As _
    System.Collections.Generic.IEnumerable(Of _
        System.Xml.Linq.XNode)
```

So the above example is equivalent to:

```
Dim customerName = customer.Elements("name").Value
```

- *Attribute access*, in which a Visual Basic identifier follows a dot and an at sign, or an XML name follows a dot and an at sign. For example:

  ```
  Dim customer = <customer age="30"/>
  Dim customerAge = customer.@age
  ```

  Attribute access maps to the function:

  ```
  Function AttributeValue(name As System.Xml.Linq.XName) as String
  ```

  So the above example is equivalent to:

  ```
  Dim customerAge = customer.AttributeValue("age")
  ```

  Note. The `AttributeValue` extension method (as well as the related extension property `Value`) is not currently defined in any assembly. If the extension members are needed, they are automatically defined in the assembly being produced.

- *Descendents access*, in which an XML names follows three dots. For example:

  ```
  Dim company = _
      <company>
          <customers>
              <customer>Bob</>
              <customer>Mary</>
              <customer>Joe</>
          </>
      </>
  Dim customers = company...<customer>
  ```

  Descendents access maps to the function:

  ```
  Function Descendents(name As System.Xml.Linq.XName) As _
      System.Collections.Generic.IEnumerable(Of _
          System.Xml.Linq.XElement)
  ```

  So the above example is equivalent to:

  ```
  Dim customers = company.Descendants("customer")
  ```

The base expression of an XML member access expression must be a value and must be of the type:

- If an element or descendents access, `System.Xml.Linq.XContainer` or a derived type, or `System.Collections.Generic.IEnumerable(Of T)` or a derived type, where `T` is `System.Xml.Linq.XContainer` or a derived type.
- If an attribute access, `System.Xml.Linq.XElement` or a derived type, or `System.Collections.Generic.IEnumerable(Of T)` or a derived type, where `T` is `System.Xml.Linq.XElement` or a derived type.

Names in XML member access expressions cannot be empty. They can be namespace qualified, using any namespaces defined by imports. For example:

```
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
```

Whitespace is not allowed after the dot(s) in an XML member access expression, or between the angle brackets and the name. For example:

```
    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 >
```

If the types in the `System.Xml.Linq` namespace are not available, then an XML member access expression will cause a compile-time error.

11.25 Await Operator

The await operator is related to async methods, which are described in Section §10.1.3.

```
AwaitOperatorExpression:
    | 'Await' Expression
    ;
```

`Await` is a reserved word if the immediately enclosing method or lambda expression in which it appears has an `Async` modifier, and if the `Await` appears after that `Async` modifier; it is unreserved elsewhere. It is also unreserved in preprocessor directives. The await operator is only allowed in the body of a method or lambda expressions where it is a reserved word. Within the immediately enclosing method or lambda, an await expression may not occur inside the body of a `Catch` or `Finally` block, nor inside the body of a `SyncLock` statement, nor inside a query expression.

The await operator takes a single expression which must be classified as a value and whose type must be an *awaitable* type, or `Object`. If its type is `Object` then all processing is deferred until run-time. A type `C` is said to be awaitable if all of the following are true:

- `C` contains an accessible instance or extension method named `GetAwaiter` which has no arguments and which returns some type `E`;
- `E` contains a readable instance or extension property named `IsCompleted` which takes no arguments and has type Boolean;
- `E` contains an accessible instance or extension method named `GetResult` which takes no arguments;
- `E` implements either `System.Runtime.CompilerServices.INotifyCompletion` or `ICriticalNotifyCompletion`.

If `GetResult` was a `Sub`, then the await expression is classified as void. Otherwise, the await expression is classified as a value and its type is the return type of the `GetResult` method.

Here is an example of a class that can be awaited:

```
    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
```

Note. Library authors are recommended to follow the pattern that they invoke the continuation delegate on the same `SynchronizationContext` as their `OnCompleted` was itself invoked on. Also, the resumption delegate should not be executed synchronously within the `OnCompleted` method since that can lead to stack overflow: instead, the delegate should be queued for subsequent execution.

When control flow reaches an `Await` operator, behavior is as follows.

1. The `GetAwaiter` method of the await operand is invoked. The result of this invocation is termed the *awaiter*.
2. The awaiter's `IsCompleted` property is retrieved. If the result is true then:
    a. The `GetResult` method of the awaiter is invoked. If `GetResult` was a function, then the value of the await expression is the return value of this function.
3. If the IsCompleted property isn't true then:
    a. Either `ICriticalNotifyCompletion.UnsafeOnCompleted` is invoked on the awaiter (if the awaiter's type `E` implements `ICriticalNotifyCompletion`) or `INotifyCompletion.OnCompleted` (otherwise). In both cases it passes a *resumption delegate* associated with the current instance of the async method.
    b. The control point of the current async method instance is suspended, and control flow resumes in the *current caller* (defined in Section §10.1.3).
    c. If later the resumption delegate is invoked,
        i. the resumption delegate first restores `System.Threading.Thread.CurrentThread.ExecutionContext` to what it was at the time `OnCompleted` was called,
        ii. then it resumes control flow at the control point of the async method instance (see Section §10.1.3),
        iii. where it calls the `GetResult` method of the awaiter, as in 2.1 above.

If the await operand has type Object, then this behavior is deferred until runtime:

- Step 1 is accomplished by calling GetAwaiter() with no arguments; it may therefore bind at runtime to instance methods which take optional parameters.
- Step 2 is accomplished by retrieving the IsCompleted() property with no arguments, and by attempting an intrinsic conversion to Boolean.
- Step 3.a is accomplished by attempting `TryCast(awaiter, ICriticalNotifyCompletion)`, and if this fails then `DirectCast(awaiter, INotifyCompletion)`.

The resumption delegate passed in 3.a may only be invoked once. If it is invoked more than once, the behavior is undefined.

12. Documentation Comments

Documentation comments are specially formatted comments in the source that can be analyzed to produce documentation about the code they are attached to. The basic format for documentation comments is XML. When the compiling code with documentation comments, the compiler may optionally emit an XML file that represents the sum total of the documentation comments in the source. This XML file can then be used by other tools to produce printed or online documentation.

This chapter describes document comments and recommended XML tags to use with document comments.

12.1 Documentation Comment Format

Document comments are special comments that begin with `'''`, three single quote marks. They must immediately precede the type (such as a class, delegate, or interface) or type member (such as a field, event, property, or method) that they document. A document comment on a partial method declaration will be replaced by the document comment on the method that supplies its body, if there is one. All adjacent document comments are appended together to produce a single document comment. If there is a whitespace character following the `'''` characters, then that whitespace character is not included in the concatenation. For example:

```
''' <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
```

Documentation comments must be well formed XML according to http://www.w3.org/TR/REC-xml. If the XML is not well formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.

Although developers are free to create their own set of tags, a recommended set is defined in the next section. Some of the recommended tags have special meanings:

- The `<param>` tag is used to describe parameters. The parameter specified by a `<param>` tag must exist and all parameters of the type member must be described in the documentation comment. If either condition is not true, the compiler issues a warning.
- The `cref` attribute can be attached to any tag to provide a reference to a code element. The code element must exist; at compile-time the compiler replaces the name with the ID string representing the member. If the code element does not exist, the compiler issues a warning. When looking for a name described in a `cref` attribute, the compiler respects `Imports` statements that appear within the containing source file.
- The `<summary>` tag is intended to be used by a documentation viewer to display additional information about a type or member.

Note that the documentation file does not provide full information about a type and members, only what is contained in the document comments. To get more information about a type or member, the documentation file must be used in conjunction with reflection on the actual type or member.

12.2 Recommended tags

The documentation generator must accept and process any tag that is valid according to the rules of XML. The following tags provide commonly used functionality in user documentation:

`<c>` Sets text in a code-like font

`<code>` Sets one or more lines of source code or program output in a code-like font

`<example>` Indicates an example

`<exception>` Identifies the exceptions a method can throw

`<include>` Includes an external XML document

`<list>` Creates a list or table

`<para>` Permits structure to be added to text

`<param>` Describes a parameter for a method or constructor

`<paramref>` Identifies that a word is a parameter name

`<permission>` Documents the security accessibility of a member

`<remarks>` Describes a type

`<returns>` Describes the return value of a method

`<see>` Specifies a link

`<seealso>` Generates a See Also entry

`<summary>` Describes a member of a type

`<typeparam>` Describes a type parameter

`<value>` Describes a property

12.2.1 <c>

This tag specifies that a fragment of text within a description should use a font like that used for a block of code. (For lines of actual code, use `<code>`.)

Syntax:

```
<c>text to be set like code</c>
```

Example:

```
''' <remarks>
''' Class <c>Point</c> models a point in a two-dimensional plane.
''' </remarks>
Public Class Point
End Class
```

12.2.2 <code>

This tag specifies that one or more lines of source code or program output should use a fixed-width font. (For small code fragments, use `<c>`.)

Syntax:

```
<code>source code or program output</code>
```

Example:

```
''' <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
```

12.2.3 <example>

This tag allows example code within a comment to show how an element can be used. Ordinarily, this will involve use of the tag `<code>` as well.

Syntax:

```
<example>description</example>
```

Example:

See `<code>` for an example.

12.2.4 <exception>

This tag provides a way to document the exceptions a method can throw.

Syntax:

```
<exception cref="member">description</exception>
```

Example:

```
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
```

12.2.5 <include>

This tag is used to include information from an external well-formed XML document. An XPath expression is applied to the XML document to specify what XML should be included from the document. The `<include>` tag is then replaced with the selected XML from the external document.

Syntax:

```
<include file="filename" path="xpath">
```

Example:

If the source code contained a declaration like the following:

```
''' <include file="docs.xml" path="extra/class[@name="IntList"]/*" />
```

and the external file docs.xml had the following contents

```
<?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>
```

then the same documentation is output as if the source code contained:

```
''' <summary>
''' Contains a list of integers.
''' </summary>
```

12.2.6 <list>

This tag is used to create a list or table of items. It may contain a `<listheader>` block to define the heading row of either a table or definition list. (When defining a table, only an entry for term in the heading need be supplied.)

Each item in the list is specified with an `<item>` block. When creating a definition list, both term and description must be specified. However, for a table, bulleted list, or numbered list, only description need be specified.

Syntax:

```
<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>
```

Example:

```
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
```

12.2.7 <para>

This tag is for use inside other tags, such as <remarks> or <returns>, and permits structure to be added to text.

Syntax:

```
<para>content</para>
```

Example:

```
''' <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
```

12.2.8 <param>

This tag describes a parameter for a method, constructor, or indexed property.

Syntax:

```
<param name="name">description</param>
```

Example:

```
''' <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
```

12.2.9 <paramref>

This tag indicates that a word is a parameter. The documentation file can be processed to format this parameter in some distinct way.

Syntax:

```
<paramref name="name"/>
```

Example:

```
''' <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
```

12.2.10 <permission>

This tag documents the security accessibility of a member

Syntax:

```
<permission cref="member">description</permission>
```

Example:

```
''' <permission cref="System.Security.PermissionSet">Everyone can
''' access this method.</permission>
Public Shared Sub Test()
End Sub
```

12.2.11 <remarks>

This tag specifies overview information about a type. (Use `<summary>` to describe the members of a type.)

Syntax:

```
<remarks>description</remarks>
```

Example:

```
''' <remarks>
''' Class <c>Point</c> models a point in a two-dimensional plane.
''' </remarks>
Public Class Point
End Class
```

12.2.12 <returns>

This tag describes the return value of a method.

Syntax:

```
<returns>description</returns>
```

Example:

```
''' <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
```

12.2.13 <see>

This tag allows a link to be specified within text. (Use `<seealso>` to indicate text that is to appear in a See Also section.)

Syntax:

```
<see cref="member"/>
```

Example:

```
''' <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
```

12.2.14 <seealso>

This tag generates an entry for the See Also section. (Use `<see>` to specify a link from within text.)

Syntax:

```
<seealso cref="member"/>
```

Example:

```
''' <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
```

12.2.15 <summary>

This tag describes a type member. (Use `<remarks>` to describe a type itself.)

Syntax:

```
<summary>description</summary>
```

Example:

```
''' <summary>
''' This constructor initializes the new Point to (0,0).
''' </summary>
Public Sub New()
    Me.New(0,0)
End Sub
```

12.2.16 <typeparam>

This tag describes a type parameter.

Syntax:

```
<typeparam name="name">description</typeparam>
```

Example:

```
''' <typeparam name="T">
''' The base item type. Must implement IComparable.
''' </typeparam>
Public Class ItemManager(Of T As IComparable)
End Class
```

12.2.17 <value>

This tag describes a property.

Syntax:

```
<value>property description</value>
```

Example:

```
''' <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
```

12.3 ID Strings

When generating the documentation file, the compiler generates an ID string for each element in the source code that is tagged with a documentation comment that uniquely identifies it. This ID string can be used by external tools to identify which element in a compiled assembly corresponds to the document comment.

ID strings are generated as follows:

No white space is placed in the string.

The first part of the string identifies the kind of member being documented, via a single character followed by a colon. The following kinds of members are defined, with the corresponding character in parenthesis after it: events (E), fields (F), methods including constructors and operators (M), namespaces (N), properties (P) and types (T). An exclamation point (!) indicates an error occurred while generating the ID string, and the rest of the string provides information about the error.

The second part of the string is the fully qualified name of the element, starting at the global namespace. The name of the element, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by the pound sign (#). (It is assumed that no element has this character in its name.) The name of a type with type parameters ends with a backquote (`) followed by a number that represents the number of type parameters on the type. It is important to remember that because nested types have access to the type parameters of the types containing them, nested types implicitly contain the type parameters of their containing types, and those types are counted in their type parameter totals in this case.

For methods and properties with arguments, the argument list follows, enclosed in parentheses. For those without arguments, the parentheses are omitted. The arguments are separated by commas. The encoding of each argument is the same as a CLI signature, as follows: Arguments are represented by their fully qualified name. For example, `Integer` becomes `System.Int32`, `String` becomes `System.String`, `Object` becomes `System.Object`, and so on. Arguments having the `ByRef` modifier have a '@' following their type name. Arguments having the `ByVal`, `Optional` or `ParamArray` modifier have no special notation. Arguments that are arrays are represented as `[lowerbound:size, ..., lowerbound:size]` where the number of commas is the rank - 1, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is omitted. If the lower bound and size for a particular dimension are omitted, the ':' is omitted as well. Arrays of arrays are represented by one "`[]`" per level.

12.3.1 ID string examples

The following examples each show a fragment of VB code, along with the ID string produced from each source element capable of having a documentation comment:

Types are represented using their fully qualified name.

```
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"
```

Fields are represented by their fully qualified name.

```
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"
```

Constructors.

```
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)"
```

Methods.

```
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[])"
```

Properties.

```
    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)"
```

Events

```
Namespace Acme
    Class Widget
        Public Event AnEvent As EventHandler
        Public Event AnotherEvent()
    End Class
End Namespace

"E:Acme.Widget.AnEvent"
"E:Acme.Widget.AnotherEvent"
```

Operators.

```
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)"
```

Conversion operators have a trailing ~ followed by the return type.

```
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"
```

12.4 Documentation comments example

The following example shows the source code of a `Point` class:

```
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
```

Here is the output produced when given the source code for class `Point`, shown above:

```
<?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>
```