The Microsoft® Visual Basic® Language Specification

Version 8.0

Paul Vick
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.

> **Note** The grammars in this specification are designed to be human readable, not formal (that is, usable by LEX or YACC).

All of the grammars use a modified BNF notation, which consists of a set of productions made up of terminal and non-terminal names. A terminal name represents one or more Unicode characters. Each nonterminal name is defined by one or more productions. In a production, nonterminal names are shown in *italic type*, and terminal names are shown in a `fixed-width type`. Text in normal type and surrounded by angle-bracket metasymbols are informal terminals (for example, "< all Unicode characters >"). Each grammar starts with the nonterminal *Start*.

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.

A set of productions begins with the name of a nonterminal, followed by two colons and an equal sign. The right side contains a terminal or nonterminal production. A nonterminal may have multiple productions that are separated by the vertical-bar metasymbol (|). Items included in square-bracket metasymbols ([]) are optional. A plus metasymbol (+) following an item means the item may occur one or more times.

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:

- 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.
- A 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.
- A full version *C* of Visual Studio must be released with compiler warnings that cannot be turned off.
- 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*.
- 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.

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

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.

The following program shows some line continuations:

```
Module Test
    Function Func( _
        ByVal Param1 As Integer, _
        ByVal Param2 As Integer )

        If (Param1 < Param2) Or _
            (Param1 > Param2) Then
            Console.WriteLine("Not equal")
        End If
```

```
        End Function
    End Module
```

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

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 blank characters (class Zs) > |
< Unicode tab character (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 left single-quote character (0x2018) > |
< Unicode right single-quote character (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.

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](ByVal [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.

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

NonEscapedIdentifier ::= < *IdentifierName* but not *Keyword* >

EscapedIdentifier ::= `[` *IdentifierName* `]`

IdentifierName ::= *IdentifierStart* [*IdentifierCharacter+*]

IdentifierStart ::=
    AlphaCharacter |
    UnderscoreCharacter *IdentifierCharacter*

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

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

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

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

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

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

IdentifierOrKeyword ::= *Identifier* | *Keyword*

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. 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$(ByVal Param&)
        ' The following line causes an error because the type character
        ' conflicts with the declared type of StringFunc and LongParam.
        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.

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

IntegerTypeCharacter ::= `%`

LongTypeCharacter ::= `&`

DecimalTypeCharacter ::= `@`

SingleTypeCharacter ::= `!`

DoubleTypeCharacter ::= `#`

StringTypeCharacter ::= `$`

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 ::= < member of keyword table >

AddHandler	AddressOf	Alias	And
AndAlso	As	Boolean	ByRef
Byte	ByVal	Call	Case
Catch	CBool	CByte	CChar
CDate	CDbl	CDec	Char
CInt	Class	CLng	CObj
Const	Continue	CSByte	CShort
CSng	CStr	CType	CUInt
CULng	CUShort	Date	Decimal
Declare	Default	Delegate	Dim
DirectCast	Do	Double	Each
Else	ElseIf	End	EndIf
Enum	Erase	Error	Event
Exit	False	Finally	For
Friend	Function	Get	GetType
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).

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.

Annotation

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.

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`

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.

Annotation:

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. 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
```

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 refer to the same string instance. For instance, the output of the following program is `True` because the two literals refer to the same string instance.

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

StringLiteral ::=
  DoubleQuoteCharacter [*StringCharacter+*] *DoubleQuoteCharacter*

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

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

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.

```
Module Test
    Sub Main()
```

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

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

CharacterLiteral ::= *DoubleQuoteCharacter* *StringCharacter* *DoubleQuoteCharacter* `C`

2.4.6 Date Literals

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

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

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

DateValue ::=
MonthValue `/` *DayValue* `/` *YearValue* |
MonthValue `-` *DayValue* `-` *YearValue*

TimeValue ::=
HourValue `:` *MinuteValue* [`:` *SecondValue*] [*WhiteSpace+*] [*AMPM*]

MonthValue ::= *IntLiteral*

DayValue ::= *IntLiteral*

YearValue ::= *IntLiteral*

HourValue ::= *IntLiteral*

MinuteValue ::= *IntLiteral*

SecondValue ::= *IntLiteral*

AMPM ::= AM | PM

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. Because the conditional compilation process is done after lexical analysis, even disabled lines must be lexically valid.

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.

Start ::= [*CCStatement+*]

CCStatement ::=
CCConstantDeclaration |
CCIfGroup |
LogicalLine

CCExpression ::=
LiteralExpression |
CCParenthesizedExpression |
SimpleNameExpression |
CCCastExpression |
CCOperatorExpression

CCParenthesizedExpression ::= `(` *CCExpression* `)`

CCCastExpression ::= *CastTarget* `(` *CCExpression* `)`

CCOperatorExpression ::=
CCUnaryOperator *CCExpression*
CCExpression *CCBinaryOperator* *CCExpression*

CCUnaryOperator ::= `+` | `-` | `Not`

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

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. 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(ByVal 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.

CCConstantDeclaration ::= `# Const` *Identifier* `=` *CCExpression LineTerminator*

3.1.2 Conditional Compilation Directives

Conditional compilation directives control conditional compilation and 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
        CommitHelper()
    End Sub
End Class
```

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

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

CCElseGroup ::=
  `# Else` *LineTerminator*
  [*CCStatement+*]

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. 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
```

Start ::= [*ExternalSourceStatement+*]

ExternalSourceStatement ::= *ExternalSourceGroup* | *LogicalLine*

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

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). These directives are special in that they can neither start nor terminate within a method body. For example:

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

Start ::= [*RegionStatement+*]

RegionStatement ::= *RegionGroup* | *LogicalLine*

RegionGroup ::=
  `# Region` *StringLiteral* *LineTerminator*
  [*LogicalLine+*]
  `# End Region` *LineTerminator*

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.

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
```

Start ::= [*ExternalChecksumStatement+*]

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

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.

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 name of the type member, 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 F()                                 ' Signature is F().
    Sub F(x As Integer)                     ' Signature is F(Integer).
    Sub F(ByRef x As Integer)               ' Signature is F(Integer).
    Sub F(x As Integer, y As Integer)       ' Signature is F(Integer,
Integer).
    Function F(s As String) As Integer      ' Signature is F(String).
    Function F(x As Integer) As Integer     ' Signature is F(Integer).
    Sub F(a() As String)                    ' Signature is F(String()).
    Sub F(ParamArray ByVal a() As String)   ' Signature is F(String()).
    Sub F(Of T)()                           ' Signature is F!1().
    Sub F(Of T, U)(x As T, y As U)          ' Signature is F!2(!1, !2)
    Sub F(Of U, T)(x As U, y As T)          ' Signature is F!2(!2, !1)
    Sub F(Of T)(x As T)                     ' Signature is F!1(!1)
    Sub F(Of T As IDisposable)(x As T)      ' Signature is F!1(!1)
End Interface
```

A method with optional parameters is considered to have multiple signatures, one for each set of parameters that can be passed in by the caller. For example, the following method has three corresponding signatures:

```
Sub F(ByVal x As Short, _
      ByVal Optional y As Integer = 10, _
      ByVal Optional z As Long = 20)
```

These are the method's signatures:

- F(Short)
- F(Short, Integer)
- F(Short, Integer, Long)

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(ByVal x As Integer)
    End Sub

    Sub F(ByVal x As T)
    End Sub

    Sub G(Of U)(ByVal x As T, ByVal y As U)
    End Sub

    Sub G(Of U)(ByVal x As U, ByVal 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 nonconstructor 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 can not 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(ByVal i As Integer)
End Interface

Interface IListCounter
    Inherits IList
    Inherits ICounter
End Interface

Module Test
    Sub F(ByVal 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(ByVal 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(ByVal 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(ByVal i As Integer)
End Interface

Interface ILeft
    Inherits IBase
    Shadows Sub F(ByVal i As Integer)
End Interface

Interface IRight
    Inherits IBase
    Sub G()
End Interface

Interface IDerived
    Inherits ILeft, IRight
End Interface

Class Test
    Sub H(ByVal 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 redeclaring 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(ByVal i As Integer)
    End Sub

    Sub G()
    End Sub

    Sub G(ByVal i As Integer)
    End Sub
End Class

Class Derived
    Inherits Base

    ' Only hides F(Integer).
    Overloads Sub F(ByVal i As Integer)
    End Sub

    ' Hides G() and G(Integer).
    Shadows Sub G(ByVal i As Integer)
    End Sub
End Class

Module Test
    Sub Main()
        Dim x As Derived = New Derived()
```

```
        x.F() ' Calls Base.F().
        x.G() ' Error: No such method.
    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(ByVal ParamArray x As Integer())
        Console.WriteLine("Base")
    End Sub
End Class

Class Derived
    Inherits Base

    Overloads Sub F(ByVal x As Integer())
        Console.WriteLine("Derived")
    End Sub
End Class

Module Test
    Sub Main
        Dim d As Derived = 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`.

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(ByVal 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(ByVal 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(ByVal value As Byte) As Boolean
End Interface

Class TestableClass
    Implements ITestable

    Function Test(ByVal 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

    ' This causes an error because 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, and all of the constraints on method parameters must match. For example:

```
Interface ITest
    Sub F(ByRef x As Integer)
    Sub G(ByVal Optional y As Integer = 20)
    Sub H(ByVal Paramarray z() As Integer)
End Interface

Class Test
    Implements ITest

    ' Error: ByRef/ByVal mismatch.
    Sub F(ByVal x As Integer) Implements ITest.F
    End Sub
```

```
        ' Error: Defaults do not match.
        Sub G(ByVal Optional y As Integer = 10) Implements ITest.G
        End Sub

        ' Error: Paramarray does not match.
        Sub H(ByVal 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(ByVal i As Integer)
        Sub G(ByVal i As Integer)
    End Interface

    Class Test
        Implements ITest

        Sub F(ByVal 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:

```
    Class I1(Of U, V)
        Public Sub M(ByVal x As U, ByVal y As List(Of V))
    End Class

    Class C1(Of W, X)
        Implements I1(Of W, X)

        ' W corresponds to U and X corresponds to V
        Public Sub M(ByVal x As W, ByVal 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(ByVal x As String, ByVal 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(ByVal x As T)
    Sub S1(ByVal 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
    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(ByVal g As Graphics, _
                              ByVal r As Rectangle)
End Class

Public Class Ellipse
    Inherits Shape

    Public Overrides Sub Paint(ByVal g As Graphics, ByVal r As Rectangle)
        g.drawEllipse(r)
    End Sub
End Class

Public Class Box
    Inherits Shape

    Public Overrides Sub Paint(ByVal g As Graphics, ByVal 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:

```
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. 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 MustOverride 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 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 two methods are not in the same program.
- 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(ByVal x As U, ByVal y As List(Of V))
    End Sub
End Class

Class Derived(Of W, X)
    Inherits C(Of W, X)

    ' W corresponds to U and X corresponds to V
    Public Overrides Sub M(ByVal x As W, ByVal 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(ByVal x As String, ByVal 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(ByVal x As T)
    End Sub

    Public MustOverride Sub S1(ByVal y As U)
    End Sub
End Class

Class Derived
    Inherits Base(Of Integer, Integer)

    ' Error: Can't override both S1's at once
    Public Overrides Sub S1(ByVal 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`.

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.
- 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.

For a generic type, the declaration context includes type parameters. This means that a generic type with one set of type arguments does not have access to the Private or Protected members of the generic type with a different set of type arguments. For example:

```
Class C(Of T)
    Private x As T
    Protected y As T

    Sub F()
        Dim z As C(Of String)
        ' Error: C(Of T) cannot access C(Of String)'s private members
        z.x = "a"
    End Sub
End Class
```

Annotation

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

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

```
    Public Class User
        Protected Password As String
    End Class

    Public Class Employee
        Inherits User
    End Class

    Public Class Guest
        Inherits User

        Public Function GetPassword(ByVal 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`.

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

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

Private 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
```

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:

```
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
```

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.

QualifiedIdentifier ::=
  Identifier |
  `Global` `.` *IdentifierOrKeyword* |
  QualifiedIdentifier `.` *IdentifierOrKeyword*

4.7.1 Qualified Name Resolution

Given a qualified namespace or type name of the form `N.R`, where `R` is the rightmost identifier in the qualified name, the following steps describe how to determine to which namespace or type the qualified name refers:

- Resolve `N`, which may be either a qualified or unqualified name.
- If resolution of `N` fails, resolves to a type parameter, or does not resolve to a namespace or type, a compile-time error occurs. If `R` matches the name of a namespace or type in `N`, then the qualified name refers to that namespace or type.
- If `N` contains one or more standard modules, and `R` matches the name of a type in exactly one standard module, then the qualified name refers to that type. If `R` matches the name of types in more than one standard module, a compile-time error occurs.
- 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

Given an unqualified name `R`, the following steps describe how to determine to which namespace or type an unqualified name refers:

- For each nested type containing the name reference, starting from the innermost type and going to the outermost, if `R` matches the name of an accessible nested type or a type parameter in the current type, then the unqualified name refers to that type or type parameter.
- For each nested namespace containing the name reference, starting from the innermost namespace and going to the outermost namespace, do the following:
  - If `R` matches the name of an accessible type or nested namespace in the current namespace, then the unqualified name refers to that type or nested namespace.
  - If the namespace contains one or more accessible standard modules, and `R` matches the name of an accessible nested type in exactly one standard module, then the unqualified name refers to that nested type. If `R` matches the name of accessible nested types in more than one standard module, a compile-time error occurs.
- 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 the source file containing the name reference has one or more imports:

- If `R` matches the name of an accessible type in exactly one import, then the unqualified name refers to that type. If `R` matches the name of an accessible type in more than one import and all are not the same entity, a compile-time error occurs.
- If `R` matches the name of a namespace in exactly one import, then the unqualified name refers to that namespace. If `R` matches the name of a namespace in more than one import and all are not the same entity, a compile-time error occurs.
- If the imports contain one or more accessible standard modules, and `R` matches the name of an accessible nested type in exactly one standard module, then the unqualified name refers to that type. If `R` matches the name of accessible nested types in more than one standard module, a compile-time error occurs.

- 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 the compilation environment defines one or more imports:
  - If `R` matches the name of an accessible type in exactly one import, then the unqualified name refers to that type. If `R` matches the name of an accessible type in more than one import, a compile-time error occurs.
  - If `R` matches the name of a namespace in exactly one import, then the unqualified name refers to that namespace. If `R` matches the name of a namespace in more than one import, a compile-time error occurs.
  - If the imports contain one or more accessible standard modules, and `R` matches the name of an accessible nested type in exactly one standard module, then the unqualified name refers to that type. If `R` matches the name of accessible nested types in more than one standard module, a compile-time error occurs.
- 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.

If the type name is a constructed type name (i.e. it includes a type argument list), then only types with the same arity as the type argument list are matched.

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 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.

Annotation

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(ByVal 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 Stack(Of Integer)
        Dim s2 As 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.

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(ByVal 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)
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` 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.

TypeParameterList ::=
  `( Of` *TypeParameters* `)`

TypeParameters ::=
  TypeParameter |
  TypeParameters `,` *TypeParameter*

TypeParameter ::=
  Identifier [*TypeParameterConstraints*]

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.

```
Public Class DisposableStack(Of ItemType As IDisposable)
    Implements IDisposable

    Protected Items(0 To 99) As ItemType
    Protected CurrentIndex As Integer = 0

    Public Sub Push(ByVal 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
                Item.Dispose()
            Next
        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.

Type constraints that are classes specify a common base class that a type argument must either be the same as or inherit from. Type constraints that are interfaces specify an interface that a type argument must implement. Whenever a generic type or method is referenced, the type arguments supplied must derive from or implement all of the bounds given for the matching type parameter.

Type constraints must satisfy the following rules:

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

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 preferred in the following order, with the class constraint being the most preferred: the class constraint, the interface constraints, `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.

```
    Class C1
        Sub S1(ByVal x As Integer)
        End Sub
    End Class

    Interface I1
        Sub S1(ByVal x As Integer)
    End Interface
```

```
Interface I2
    Sub S1(ByVal 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
```

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
```

In the case where a type constraint uses a containing type's type parameter, the constraint requires that the type parameter be the same as or inherit from the type supplied for the type parameter. For example:

```
Class List(Of T)
    Sub AddRange(Of S As T)(ByVal 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`.

When overriding a method or implementing an interface member, it is possible that a type parameter constrained on another type parameter would require specifying a type constraint that is illegal. In those cases, the following relaxations apply:

- Multiple class constraints may be applied as long as they have an inheritance relationship. The most derived class is considered to be the constraint.
- The same interface constraint, `Class` constraint or `Structure` constraint can be applied multiple times.
- The type may be `NotInheritable`.

- The type may be one of the following special types: `System.Array`, `System.Delegate`, `System.MulticastDelegate`, `System.Enum`, or `System.ValueType`.

A type parameter constrained to one of the types allowed by the above relaxations can only use the conversions allowed by the `DirectCast` operator. For example:

```
MustInherit Class Base(Of T)
    MustOverride Sub S1(Of U As T)(ByVal 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)(ByVal x As U)
        Dim y As Integer = x    ' OK
        Dim z As Long = x       ' Error: Can't convert
    End Sub
End Class
```

There is a special type constraint called `New` that requires that the type be able to be used in `New` expressions. If `New` is specified as a type parameter bound, any type argument used for that type parameter 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
```

There are also two special type constraints, `Class` and `Structure`, which limit the kind of type that can be used to satisfy the constraint. The `Class` constraint constrains the type parameter to references types only. The `Structure` constraint constrains the type parameter to value types only, with the exception of `System.Nullable(Of T)`.

Annotation

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.

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.

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

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(ByVal v1 As V)
        V1.Print()
    End Sub
End Class
```

TypeParameterConstraints ::=
  `As` *Constraint* |
  `As` `{` *ConstraintList* `}`

ConstraintList ::=
  ConstraintList , *Constraint* |
  Constraint

Constraint ::= *TypeName* | `New`

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(ByVal 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 must 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
```

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
```

```
    Public Sub New(ByVal Value As String)
    End Sub

    Public ReadOnly Property Value() As String
        Get
        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(ByVal Value As Boolean)
    End Sub
End Class

<AttributeUsage(AttributeTargets.Class, Inherited := True)> _
Class SingleUseAttribute
    Inherits Attribute

    Public Sub New(ByVal 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`, `Optional` or `Paramarray`. 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.

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(ByVal 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 either the `Assembly` or `Module` modifier and a colon.

Attributes ::=
  AttributeBlock |
  Attributes *AttributeBlock*

AttributeBlock ::= `<` *AttributeList* `>`

AttributeList ::=
  Attribute |
  AttributeList , *Attribute*

Attribute ::=
  [*AttributeModifier* `:`] *SimpleTypeName* [`(` [*AttributeArguments*] `)`]

AttributeModifier ::= `Assembly` | `Module`

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 `X` and `XAttribute`.

```
<AttributeUsage(AttributeTargets.All)> _
Public Class X
    Inherits System.Attribute
End Class

<AttributeUsage(AttributeTargets.All)> _
Public Class XAttribute
    Inherits System.Attribute
End Class

' Refers to XAttribute.
<X()> _
Class Class1
End Class

' Refers to XAttribute.
<XAttribute()> _
Class Class2
End Class
```

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

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:

- 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.
- 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.
- 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(ByVal x As Integer)
    End Sub

    Public Sub New(Byval 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 Foo
End Class

' Calls the second constructor.
<General(10.5, z := 10)> _
Class Bar
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(ByVal 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 `,` *VariablePropertyInitializerList* |
  VariablePropertyInitializerList

AttributePositionalArgumentList ::=
  AttributeArgumentExpression |
  AttributePositionalArgumentList `,` *AttributeArgumentExpression*

VariablePropertyInitializerList ::=
  VariablePropertyInitializer |
  VariablePropertyInitializerList `,` *VariablePropertyInitializer*

VariablePropertyInitializer ::=
  IdentifierOrKeyword `:=` *AttributeArgumentExpression*

AttributeArgumentExpression ::=
  ConstantExpression |
  GetTypeExpression |
  ArrayCreationExpression

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, and must have one of the following signatures:

```
Sub Main()
Sub Main(ByVal Args() As String)
Function Main() As Integer
```

```
Function Main(ByVal 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*. 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 three compilation options: strict type semantics, explicit declaration semantics, and comparison 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 fourth compilation option, integer overflow checks, which can only be specified through the compilation environment.

OptionStatement ::=
    OptionExplicitStatement |
    OptionStrictStatement |
    OptionCompareStatement

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.

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`.

OptionExplicitStatement ::= `Option Explicit` [*OnOff*] *StatementTerminator*

OnOff ::= `On` | `Off`

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.

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.Foo() ' 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`, and `Is`.
- Omitting the `As` clause in a declaration.

OptionStrictStatement ::= `Option Strict` [*OnOff*] *StatementTerminator*

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.

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

```
Option Compare Text

Module M
    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`.

OptionCompareStatement ::= `Option Compare` *CompareOption StatementTerminator*

CompareOption ::= `Binary` | `Text`

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.

6.3 Imports Statement

`Imports` statements import the names of entities into a source file, allowing the names to be referenced without qualification.

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
```

ImportsStatement ::= `Imports` *ImportsClauses* *StatementTerminator*

ImportsClauses ::=
  ImportsClause |
  ImportsClauses `,` *ImportsClause*

ImportsClause ::= *ImportsAliasClause* | *ImportsNamespaceClause*

6.3.1 Import Aliases

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

```
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.

Annotation

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.

```
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`.

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.

ImportsAliasClause ::=
  Identifier = *QualifiedIdentifier* |
  Identifier = *ConstructedTypeName*

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. 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.

ImportsNamespaceClause ::=
QualifiedIdentifier |
ConstructedTypeName

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

A namespace declaration consists of the keyword `Namespace` followed by a qualified identifier and optional namespace member declarations. If the 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
```

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.

NamespaceDeclaration ::=
    `Namespace` *QualifiedIdentifier* *StatementTerminator*
    [*NamespaceMemberDeclaration+*]
    `End Namespace` *StatementTerminator*

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 |
    ConstructedTypeName

SimpleTypeName ::=
    QualifiedIdentifier |
    BuiltInTypeName

BuiltInTypeName ::= `Object` | *PrimitiveTypeName*

TypeModifier ::= *AccessModifier* | `Shadows`

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 reference. A *null reference* is a reference that refers to nothing; it is not possible to do anything with a null reference 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 reference.

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 reference, 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`..

7.2 Interface Implementation

Structure and class declarations may declare that they implement a set of interface types through one or more `Implements` clauses. 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(ByVal 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(ByVal 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(ByVal Text As String)
End Interface

Structure TextBox
    Implements ITextBox

    Public Sub Paint() Implements ITextBox.Paint
    End Sub

    Public Sub SetText(ByVal 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
```

TypeImplementsClause ::= `Implements` *Implements* *StatementTerminator*

Implements ::=
  NonArrayTypeName |
  Implements `,` *NonArrayTypeName*

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`.

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 reference.

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

NumericTypeName ::= *IntegralTypeName* | *FloatingPointTypeName* | `Decimal`

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

FloatingPointTypeName ::= `Single` | `Double`

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. 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.

EnumDeclaration ::=
  [*Attributes*] [*TypeModifier+*] `Enum` *Identifier* [`As` *QualifiedName*] *StatementTerminator*
  EnumMemberDeclaration+
  `End Enum` *StatementTerminator*

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* [`=` *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(ByVal 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 Class
```

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

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

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. 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(ByVal 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 (ByVal value As Integer)
            MyVariable = value
        End Set
    End Property

    Default Public Property Item(ByVal index As Integer) As Integer
        Get
            Return 0
        End Get

        Set (ByVal 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.

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

ClassModifier ::= *TypeModifier* | `MustInherit` | `NotInheritable` | `Partial`

7.5.1 Class Base Specification

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

ClassBase ::= `Inherits` *NonArrayTypeName* *StatementTerminator*

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.

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 declaration occurs. If the member has `Friend` access, its scope extends to the class body of any derived class in the same program, 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.

ClassMemberDeclaration ::=
    NonModuleDeclaration |

EventMemberDeclaration |
VariableMemberDeclaration |
ConstantMemberDeclaration |
MethodMemberDeclaration |
PropertyMemberDeclaration |
ConstructorMemberDeclaration |
OperatorDeclaration

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.

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 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(ByVal x As Integer, ByVal y As Integer)
        Me.x = x
        Me.y = y
    End Sub
End Structure
```

Given the above declaration the following code fragment outputs the value `10`:

```
Point a = new Point(10, 10)
Point b = a

a.x = 100
System.Console.WriteLine(b.x)
```

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.

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

StructureModifier ::= *TypeModifier* | `Partial`

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`. Structures must contain at least one instance variable.

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(ByVal x As Integer, ByVal 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)
```

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.

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

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.

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.

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

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`.

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.

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

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.

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(ByVal index As Integer) As String
    Event E()
    Sub F(ByVal 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(ByVal Text As String)
End Interface

Interface IListBox
    Inherits IControl
    Sub SetItems(ByVal 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(ByVal b As Binder)
End Interface

Public Class EditBox
    Inherits Control
    Implements IControl, IDataBound
```

```
    Public Sub Paint() Implements IControl.Paint
        ' ...
    End Sub

    Public Sub Bind(ByVal b As Binder) Implements IDataBound.Bind
    End Sub
End Class
```

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

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`). 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(ByVal Text As String)
End Interface

Interface IListBox
    Inherits IControl
    Sub SetItems(ByVal 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(ByVal Text As String) Implements IComboBox.SetText
    End Sub

    Sub SetItems(ByVal 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.

InterfaceBase ::= `Inherits` *InterfaceBases* *StatementTerminator*

InterfaceBases ::=
  NonArrayTypeName |
  InterfaceBases `,` *NonArrayTypeName*

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.

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. Interface members have no scope, and they must always be qualified.

InterfaceMemberDeclaration ::=
 NonModuleDeclaration |
 InterfaceEventMemberDeclaration |
 InterfaceMethodMemberDeclaration |
 InterfacePropertyMemberDeclaration

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. 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 interface `IList(Of T)`; if `T` is a reference type, then the array type also implements `IList(Of U)`, where `U` is a base type of `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.

ArrayTypeName ::= *NonArrayTypeName* *ArrayTypeModifiers*

ArrayTypeModifiers ::= *ArrayTypeModifier*+

ArrayTypeModifier ::= `(` [*RankList*] `)`

RankList ::=
`,` |
RankList `,`

ArrayNameModifier ::=
ArrayTypeModifiers |
ArraySizeInitializationModifier

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. 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 method declaration may not have attributes, modifiers, a `Handles` clause, an `Implements` clause, a method body, or an `End` construct. The parameter list of the method 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(ByVal d As SimpleDelegate, ByVal 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 nonshared. All that matters is that the signature of the target method is compatible with `SimpleDelegate`.

DelegateDeclaration ::=
  [*Attributes*] [*TypeModifier+*] `Delegate` *MethodSignature* *StatementTerminator*

MethodSignature ::= *SubSignature* | *FunctionSignature*

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.

Annotation

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 on the type parameters, but the constraints from each partial declaration must match. Thus, constraints 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 INullable)
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 but their containing type, by definition, must be partial 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. In other words, the accessibility domain for a constructed type is the intersection of the accessibility domains of its constituent parts.

Annotation

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.

ConstructedTypeName ::=
  QualifiedIdentifier `(` `Of` *TypeArgumentList* `)`

TypeArgumentList ::=
  TypeName |
  TypeArgumentList `,` *TypeName*

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, `ByRef` parameter types or the type of a value being converted to `Object` or `System.ValueType`.

8. Conversions

Conversion is the process of changing a value from one type to another. Conversions may either be widening or narrowing. A *widening conversion* is a conversion from one type to another type that is guaranteed to be able to contain it. Widening conversions never fail. A *narrowing conversion* may fall into one of two categories. One category is a conversion from one type to another type that is not guaranteed to be able to contain it. The other category is a conversion between types that are sufficiently unrelated as to have no obvious conversion. Narrowing conversions, which entail loss of information, may fail.

The identity conversion (i.e. a conversion from a type to itself) is 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 may occur implicitly.

8.2 Boolean Conversions

Although `Boolean` is not a numeric type, it does have 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.

8.3 Numeric Conversions

Numeric conversions are conversions between the types Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long, Decimal, Single and Double, and enumerated types. Enumerated types are treated as if they were their underlying types for the purpose of conversions. When converting from a numeric type to an enumerated type, the numeric type is never required to conform to the set of values defined in the enumerated type.

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

A value typed as a reference type may always be converted to a base type. Conversions from a base type to a derived type only succeed at run time if the value being converted is a null reference or a reference type that is either the derived type itself or a more derived type.

Within the .NET Framework, it is possible to know at compile time whether a class type implements a particular interface type. Consequently, a class type can always be cast to an interface type that it implements. Similarly, conversions from an interface type to a class type that implements it only succeed at run time if the value being converted is a null reference or a reference type that is either the class type itself or a type derived from the class type. Because an interface type will always contain an instance of a class that derives from `Object`, an interface type can always be cast to `Object`.

However, classes that represent COM classes may have interface implementations that are not known until run time. Consequently, a class type may also be converted to an interface type that it does not implement, an interface type may be converted to a class type that does not implement it, and an interface type may be converted to another interface type with which it has no inheritance relationship. In all these cases, a check is performed at run time by the .NET Framework to determine if the class type involved implements the required interface types.

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

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 reference types `A` and `B`, if `A` is a derived type of `B` or implements `B`, a conversion exists from an array of type `A` to a compatible array of type `B`. A *compatible array* is an array of the same rank and type. 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` is a base type of `A` or is implemented by `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.

Conversions also exist between an array of an enumerated type and an array of the enumerated type's underlying type of 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`.

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 reference or an instance of the value type. Otherwise a `System.InvalidCastException` exception is thrown. A null reference is treated as if it were the literal `Nothing`.

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.

Annotation

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 boxed value 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 Object
        Dim val2 As Object

        val1 = New 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 outer boxed value is copied. Thus, both `val1.Value` and `val2.Value` have a reference to the same boxed value type.

Annotation

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 legal.

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()
End Interface

Structure Counter
   Implements ICounter

   Public value As Integer

   Sub Increment() Implements ICounter.Increment
      value += 1
   End Sub
End Structure

Module Test
      Sub Test(Of T As ICounter)(ByVal 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
```

8.7 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:

- Conversions from any type to itself.
- Conversions from any derived type to one of its base types.
- Conversions from `Byte` to `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single`, or `Double`.
- Conversions from `SByte` to `Short`, `Integer`, `Long`, `Decimal`, `Single`, or `Double`.
- Conversions from `UShort` to `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single`, or `Double`.
- Conversions from `Short` to `Integer`, `Long`, `Decimal`, `Single` or `Double`.
- Conversions from `UInteger` to `ULong`, `Long`, `Decimal`, `Single`, or `Double`.
- Conversions from `Integer` to `Long`, `Decimal`, `Single` or `Double`.
- Conversions from `ULong` to `Decimal`, `Single`, or `Double`.
- Conversions from `Long` to `Decimal`, `Single` or `Double`.
- Conversions from `Decimal` to `Single` or `Double`.
- Conversions from `Single` to `Double`.
- Conversions from the literal `Nothing` to any type.

- Conversions from the literal `0` to any enumerated type.
- Conversions from any enumerated type to its underlying type, or to any type that its underlying type has a widening conversion to.
- Conversions from any reference or value type to an interface type that the value or reference type implements.
- Conversions from any interface type to `Object`.
- Conversions from an array type `S` with an element type `SE` to a covariant-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.
  - A widening reference conversion exists from `SE` to `TE`.
- Conversions 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`.
- Conversions from `Char` to `String`.
- Conversions from `Char()` to `String`.
- Conversions 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.

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:

- Conversions from any type to a more derived type.
- Conversions from `Byte` to `SByte`.
- Conversions from `SByte` to `Byte`, `UShort`, `UInteger`, or `ULong`.
- Conversions from `UShort` to `Byte`, `SByte`, or `Short`.
- Conversions from `Short` to `Byte`, `SByte`, `UShort`, `UInteger`, or `ULong`.
- Conversions from `UInteger` to `Byte`, `SByte`, `UShort`, `Short`, or `Integer`.
- Conversions from `Integer` to `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, or `ULong`.
- Conversions from `ULong` to `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, or `Long`.
- Conversions from `Long` to `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, or `ULong`.

- Conversions from `Decimal` to `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, or `Long`.
- Conversions from Single to Byte, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, or `Decimal`.
- Conversions from Double to Byte, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, or `Single`.
- Conversions from `Boolean` to any numeric type.
- Conversions from any numeric type to `Boolean`.
- Conversions from any numeric type to any enumerated type.
- Conversions from any enumerated type to any type its underlying type has a narrowing conversion to.
- Conversions from any enumerated type to any other enumerated type.
- Conversions from any class type to any interface type, provided the class type does not implement the interface type.
- Conversions from any interface type to any class type.
- Conversions from any interface type to any value type that implements the interface type.
- Conversions from any interface type to any other interface type, provided there is no inheritance relationship between the two types.
- Conversions from an array type `S` with an element type `SE`, to a covariant-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.
  - A narrowing reference conversion exists from `SE` to `TE`.
- Conversions 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`.
- Conversions from `String` to `Char`.
- Conversions from `String` to `Char()`.
- Conversions 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`.
- Conversions from `String` to `Date` and from `Date` to `String`.

8.10 Type Parameter Conversions

For the purposes of determining permitted conversions, type parameters without a class constraint are considered to be a type derived from `Object`. This means that a type parameter `T` can be converted 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 an interface constraint does not define any additional conversions, but it does make conversions from the type parameter to the interface constraint into a widening 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. It also makes conversions from the type parameter to any interfaces `C` implements into widening conversions.

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`. 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(ByVal 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 the predefined 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(ByVal t As T) As Long
        Return CLng(CObj(t))    ' OK, conversions permitted
    End Function
End Class
```

8.11 User-defined conversions

User-defined conversions are defined by overloading the `CType` operator. When converting between types, if no predefined 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 a predefined 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.

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

- First, the value is converted from the source type to the operand type using a predefined conversion, if necessary.
- Then, the user-defined conversion is invoked.
- Finally, the result of the user-defined conversion is converted to the target type using a predefined 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:

- 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.
- 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 a predefined widening conversion operator from the source type to the operand type and there is a predefined 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.
- 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.
- 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.
- 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:

- 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.
- 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 a predefined conversion operator from the source type to the operand type and there is a predefined 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.
- 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.
- 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.
- 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.

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. 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(ByVal x As T)
End Interface

Class C1
    Implements I1(Of Integer)

    Sub F(ByVal x As Integer) Implements I1(Of Integer).F
    End Sub
End Class

Class C2(Of U)
    Implements I1(Of U)

    Sub F(ByVal 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(ByVal x As Integer, ByVal y As Integer)
    Event RightClick(ByVal x As Integer, ByVal y As Integer)
```

```
End Interface

Class Button
    Implements ClickEvents

    ' OK. Signatures match, delegate type = ClickEvents.LeftClickHandler.
    Event LeftClick(ByVal x As Integer, ByVal y As Integer) _
        Implements ClickEvents.LeftClick

    ' OK. Signatures match, delegate type = ClickEvents.RightClickHandler.
    Event RightClick(ByVal x As Integer, ByVal y As Integer) _
        Implements ClickEvents.RightClick
End Class

Class Label
    Implements ClickEvents

    ' Error. Signatures match, but can't be both delegate types.
    Event Click(ByVal x As Integer, ByVal 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 Sub
```

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)

    Sub F1() As Integer Implements I1(Of Integer).F
    End Sub
```

```
        Sub F2() As Double Implements I1(Of Double).F
        End Sub

        ' Error: I1(Of String) is not implemented by C1
        Sub F3() As String Implements I1(Of String).F
        End Sub
    End Class

    Class C2(Of U)
        Implements I1(Of U)

        Sub F() As U Implements I1(Of U).F
        End Sub
    End Class
```

ImplementsClause ::= [`Implements` *ImplementsList*]

ImplementsList ::=
  InterfaceMemberSpecifier |
  ImplementsList `,` *InterfaceMemberSpecifier*

InterfaceMemberSpecifier ::= *NonArrayTypeName* `.` *IdentifierOrKeyword*

9.2 Methods

Methods contain the executable statements of a program. Methods, which have an optional list of parameters and an optional return value, are either shared or nonshared. Shared methods are accessed through the class or instances of the class. Nonshared 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(ByVal s As Stack) As Stack
        End Function

        Public Shared Function Flip(ByVal s As Stack) As Stack
        End Function

        Public Function Pop() As Object
        End Function

        Public Sub Push(ByVal 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 name of the method and the number and types of its parameters. The signature of a method specifically does not include the return type or parameter modifiers. The following example shows a class with a number of `F` methods:

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

    Sub F(ByVal o As Object)
        Console.WriteLine("F(Object)")
    End Sub

    Sub F(ByVal value As Integer)
        Console.WriteLine("F(Integer)")
    End Sub

    Sub F(ByVal a As Integer, ByVal b As Integer)
        Console.WriteLine("F(Integer, Integer)")
    End Sub
```

```
        Sub F(ByVal values() As Integer)
            Console.WriteLine("F(Integer())")
        End Sub

        Sub Main()
            F()
            F(1)
            F(CType(1, Object))
            F(1, 2)
            F(New Integer() { 1, 2, 3 })
        End Sub
    End Module
```

The output of the program is:

```
    F()
    F(Integer)
    F(Object)
    F(Integer, Integer)
    F(Integer())
```

MethodMemberDeclaration ::= *MethodDeclaration* | *ExternalMethodDeclaration*

InterfaceMethodMemberDeclaration ::= *InterfaceMethodDeclaration*

9.2.1 Regular 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.

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
```

MethodDeclaration ::=
    SubDeclaration |
    MustOverrideSubDeclaration |
    FunctionDeclaration |
    MustOverrideFunctionDeclaration

InterfaceMethodDeclaration ::=
    InterfaceSubDeclaration |
    InterfaceFunctionDeclaration

SubSignature ::= *Identifier* [*TypeParameterList*] [`(` [*ParameterList*] `)`]

FunctionSignature ::= *SubSignature* [`As` [*Attributes*] *TypeName*]

SubDeclaration ::=
    [Attributes] [ProcedureModifier+] Sub SubSignature [*HandlesOrImplements*] *LineTerminator*
    Block
    `End Sub` *StatementTerminator*

MustOverrideSubDeclaration ::=
    [*Attributes*] [*MustOverrideProcedureModifier+*] *Sub SubSignature* [*HandlesOrImplements*]
        StatementTerminator

InterfaceSubDeclaration ::=
    [*Attributes*] [*InterfaceProcedureModifier+*] *Sub SubSignature StatementTerminator*

FunctionDeclaration ::=
    [*Attributes*] [*ProcedureModifier+*] `Function` *FunctionSignature* [*HandlesOrImplements*]
        LineTerminator
    Block
    `End Function` *StatementTerminator*

MustOverrideFunctionDeclaration ::=
    [*Attributes*] [*MustOverrideProcedureModifier+*] `Function` *FunctionSignature*
        [*HandlesOrImplements*] *StatementTerminator*

InterfaceFunctionDeclaration ::=
    [*Attributes*] [*InterfaceProcedureModifier+*] `Function` *FunctionSignature StatementTerminator*

ProcedureModifier ::=
    AccessModifier |
    `Shadows` |
    `Shared` |
    `Overridable` |
    `NotOverridable` |
    `Overrides` |
    `Overloads`

MustOverrideProcedureModifier ::= *ProcedureModifier* | `MustOverride`

InterfaceProcedureModifier ::= `Shadows` | `Overloads`

HandlesOrImplements ::= *HandlesClause* | *ImplementsClause*

9.2.2 External Method Declarations

An external method declaration introduces a new method whose implementation is provided external to the program. 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" ( _
        ByVal Name As String, ByVal sa As SecurityAttributes) As Boolean
    Declare Function RemoveDirectory Lib "kernel32" ( _
```

```
            ByVal Name As String) As Boolean
        Declare Function GetCurrentDirectory Lib "kernel32" ( _
            ByVal BufSize As Integer, ByVal Buf As String) As Integer
        Declare Function SetCurrentDirectory Lib "kernel32" ( _
            ByVal Name As String) As Boolean
    End Class
```

ExternalMethodDeclaration ::=
  ExternalSubDeclaration |
  ExternalFunctionDeclaration

ExternalSubDeclaration ::=
  [*Attributes*] [*ExternalMethodModifier+*] `Declare` [*CharsetModifier*] `Sub` *Identifier*
    LibraryClause [*AliasClause*] [`(` [*ParameterList*] `)`] *StatementTerminator*

ExternalFunctionDeclaration ::=
  [*Attributes*] [*ExternalMethodModifier+*] `Declare` [*CharsetModifier*] `Function` *Identifier*
    LibraryClause [*AliasClause*] [`(` [*ParameterList*] `)`] [`As` [*Attributes*] *TypeName*]
    StatementTerminator

ExternalMethodModifier ::= *AccessModifier* | `Shadows` | `Overloads`

CharsetModifier ::= `Ansi` | `Unicode` | `Auto`

LibraryClause ::= `Lib` *StringLiteral*

AliasClause ::= `Alias` *StringLiteral*

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`. `MustOverride` methods cannot override other methods, and so cannot be combined with `Overrides`.
- `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 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. 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.

ParameterList ::=
  Parameter |
  ParameterList `,` *Parameter*

Parameter ::=
  [*Attributes*] *ParameterModifier+* *ParameterIdentifier* [`As` *TypeName*] [`=` *ConstantExpression*]

ParameterModifier ::= `ByVal` | `ByRef` | `Optional` | `ParamArray`

ParameterIdentifier ::= *Identifier* [*ArrayNameModifier*]

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(ByVal 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.

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`.

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 is passed as an argument to a reference parameter, a temporary variable may be 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) 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 TypeMismatchException 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 must specify a constant expression to be used as a replacement value if no argument is specified. The expression must be implicitly convertible to the type of the parameter, and the result of the conversion must be a constant expression. Consequently, parameters typed as structures cannot be optional parameters. 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(ByVal 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.

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(ByVal 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. 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 `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.

For a handler to be valid, the handled event's parameter types must exactly match those of the event handler. 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
    Public Event Constructed()

    Public Sub New()
        RaiseEvent Constructed
    End Sub
End Class

Module Test
    Private WithEvents x As Raiser

    Public Sub Constructed() Handles x.Constructed
        Console.WriteLine("Constructed")
    Public End Sub

    Public Sub Main()
        x = New Raiser()
        x = New Raiser()
```

```
        End Sub
    End Module
```

This will print out:

```
    Constructed
    Constructed
```

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.

HandlesClause ::= [`Handles` *EventHandlesList*]

EventHandlesList ::=
  EventMemberSpecifier |
  EventHandlesList `,` *EventMemberSpecifier*

EventMemberSpecifier ::=
  QualifiedIdentifier `.` *IdentifierOrKeyword* |
  `MyBase` `.` *IdentifierOrKeyword* |
  `Me` `.` *IdentifierOrKeyword*

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` [([*ParameterList*])] *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 always `Public`.

In the following example a default constructor is provided because the class contains no constructor declarations:

```
Class Message
    Private sender As Object
    Private text As String
End Class
```

Thus, the example is precisely equivalent to the following:

```
Class Message
    Private sender As Object
    Private text As String

    Public Sub New()
    End Sub
End Class
```

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 IntegerOrDouble(Of T)
    Shared Sub New()
        If SafeCast(T, Integer) Is Nothing AndAlso _
           SafeCast(T, Double) Is Nothing Then
            Throw New ArgumentException("T must be Integer or Double")
        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. 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`.
- A method named `add_X`, which takes the delegate type and has the same access type as the event.
- A method named `remove_X`, which takes the delegate type and has the same access type as the event.

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(ByVal i As Integer)
End Class
```

is equivalent to the following declaration

```
Class Raiser
    Public Delegate Sub ConstructedEventHandler(ByVal i As Integer)

    Protected ConstructedEvent As ConstructedEventHandler

    Public Sub add_Constructed(ByVal d As ConstructedEventHandler)
```

```
            ConstructedEvent = _
                CType( _
                    [Delegate].Combine(ConstructedEvent, d), _
                        Raiser.ConstructedEventHandler)
        End Sub

        Public Sub remove_Constructed(ByVal 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 System.EventArgs)
    Public Event DoubleClick(sender As Object, e As System.EventArgs)
    Public Event RightClick(sender As Object, e As System.EventArgs)
End Class
```

In the following example, the events simply use the same delegate, `EventHandler`:

```
Delegate Sub EventHandler(sender As Object, e As System.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 Button = New Button()
    Public Sub Button1_Click(sender As Object, e As System.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.

EventMemberDeclaration ::=
  RegularEventMemberDeclaration |
  CustomEventMemberDeclaration

RegularEventMemberDeclaration ::=
  [*Attributes*] [*EventModifiers+*] `Event` *Identifier* *ParametersOrType* [*ImplementsClause*]
    StatementTerminator

InterfaceEventMemberDeclaration ::=
  [*Attributes*] [*InterfaceEventModifiers+*] `Event` *Identifier* *ParametersOrType* *StatementTerminator*

ParametersOrType ::=
  [`(` [*ParameterList*] `)`] |
  `As` *NonArrayTypeName*

EventModifiers ::= *AccessModifier* | `Shadows` | `Shared`

InterfaceEventModifiers ::= `Shadows`

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. For example:

```
Class Test
    Private Handlers As EventHandler

    Public Custom Event TestEvent() As EventHandler
        AddHandler(ByVal value As EventHandler)
            Handlers = CType([Delegate].Combine(Handlers, value), _
                EventHandler)
        End AddHandler

        RemoveHandler(ByVal value as EventHandler)
            Handlers = CType([Delegate].Remove(Handlers, value), _
                EventHandler)
        End RemoveHandler

        RaiseEvent(ByVal sender As Object, ByVal 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 `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.

CustomEventMemberDeclaration ::=
  [*Attributes*] [*EventModifiers+*] `Custom Event` *Identifier* `As` *TypeName* [*ImplementsClause*]
    StatementTerminator
    EventAccessorDeclaration+
  `End Event` *StatementTerminator*

EventAccessorDeclaration ::=
  AddHandlerDeclaration |

RemoveHandlerDeclaration |
RaiseEventDeclaration

AddHandlerDeclaration ::=
[*Attributes*] `AddHandler (` *ParameterList* `)` *LineTerminator*
[*Block*]
`End AddHandler` *StatementTerminator*

RemoveHandlerDeclaration ::=
[*Attributes*] `RemoveHandler (` *ParameterList* `)` *LineTerminator*
[*Block*]
`End RemoveHandler` *StatementTerminator*

RaiseEventDeclaration ::=
[*Attributes*] `RaiseEvent (` *ParameterList* `)` *LineTerminator*
[*Block*]
`End RaiseEvent` *StatementTerminator*

9.5 Constants

A *constant* is a constant value that is a member of a type. 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 and strict semantics are being used, a compile-time error occurs; otherwise the type of the constant is implicitly `Object`. 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 A As Integer = 1
    Public 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.

ConstantMemberDeclaration ::=
  [*Attributes*] [*ConstantModifier+*] `Const` *ConstantDeclarators* *StatementTerminator*

ConstantModifier ::= *AccessModifier* | `Shadows`

ConstantDeclarators ::=
  ConstantDeclarator |
  ConstantDeclarators `,` *ConstantDeclarator*

ConstantDeclarator ::= *Identifier* [`As` *TypeName*] `=` *ConstantExpression* *StatementTerminator*

9.6 Instance and Shared Variables

An instance or shared variable is a member of a type that can store information. 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:

```
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)    ' Prints 1

        Dim x2 As New C(Of Double)()
        Console.WriteLine(C(Of Integer).Count)    ' Prints 1

        Dim x3 As New C(Of Integer)()
        Console.WriteLine(C(Of Integer).Count)    ' Prints 2
    End Sub
End Class
```

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
```

VariableMemberDeclaration ::=
    [*Attributes*] *VariableModifier+* *VariableDeclarators* *StatementTerminator*

VariableModifier ::=
    AccessModifier |
    `Shadows` |
    `Shared` |
    `ReadOnly` |
    `WithEvents` |
    `Dim`

VariableDeclarators ::=
    VariableDeclarator |
    VariableDeclarators `,` *VariableDeclarator*

VariableDeclarator ::=
    VariableIdentifiers [`As` [`New`] *TypeName* [`(` *ArgumentList* `)`]] |
    VariableIdentifier [`As` *TypeName*] [`=` *VariableInitializer*]

VariableIdentifiers ::=
    VariableIdentifier |
    VariableIdentifiers `,` *VariableIdentifier*

VariableIdentifier ::= *Identifier* [*ArrayNameModifier*]

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 Constructed()

    Public Sub New()
        RaiseEvent Constructed
    End Sub
End Class

Module Test
    Private WithEvents x As Raiser

    Private Sub HandleConstructed() Handles x.Constructed
        Console.WriteLine("Constructed")
    End Sub

    Public Sub Main()
        x = New Raiser()
    End Sub
End Module
```

In this example, the method `HandleConstructed` handles the event `Constructed` 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 (ByVal Value As Raiser)
                ' Unhook any existing handlers.
                If _x IsNot Nothing Then
                    RemoveHandler _x.Constructed, AddressOf HandleConstructed
                End If

                ' Change value.
                _x = Value

                ' Hook-up new handlers.
                If _x IsNot Nothing Then
                    AddHandler _x.Constructed, AddressOf HandleConstructed
                End If
            End Set
        End Property

        Sub HandleConstructed()
            Console.WriteLine("Constructed")
        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 it does not raise any events or 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 Test = New Test()

        Console.WriteLine("x = " & x & ", i = " & a.i & ", s = " & a.s)
    End Sub
End Module
```

The example produces the following output:

```
x = 1.414213562373095, 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 Test = New Test()
            Console.WriteLine("b = " & 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 four forms of variable initializers: regular initializers, array-element 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.

VariableInitializer ::= *RegularInitializer* | *ArrayElementInitializer*

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
```

RegularInitializer ::= *Expression*

9.6.3.2 Object Initializers

An object initializer is specified using the `New` keyword before the type name and an optional parameter list after the type name. An object initializer is equivalent to a regular initializer of the form `= New T(A)`, where `T` is the type name and `A` is the supplied formal parameter list, if any. 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 parameters to the constructor and never as array type modifiers.

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. 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 T(,) = 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 may not include both an array-size initializer and an array type modifier on its type or an array-element initializer.

ArraySizeInitializationModifier ::=
    `(` *BoundList* `)` [*ArrayTypeModifiers*]

BoundList::=
    Expression |
    `0 To` Expression |
    UpperBoundList `,` *Expression*

9.6.3.4 Array-Element Initializers

An array-element initializer consists of a sequence of variable initializers, enclosed by curly braces ({}) and separated by commas. Each variable initializer is an expression or, in the case of a multidimensional array, a nested array-element initializer. Each expression must be classified as a value. Array-element initializers may also be used in array-creation expressions.

The type of expression or statement in which an array-element initializer is used determines the type of the array being initialized. In an array-creation expression, the array type immediately precedes the initializer. In a variable declaration, the array type is the type of the variable being declared. When an array-element initializer is used in a variable declaration, such as:

```
    Private a As Integer() = { 0, 2, 4, 6, 8 }
```

it is simply shorthand for an equivalent array-creation expression:

```
    Private a As Integer() = New Integer() { 0, 2, 4, 6, 8 }
```

In an array-element initializer, the outermost nesting level corresponds to the leftmost dimension, and the innermost nesting level corresponds to the rightmost dimension. The initializer must have the same number levels of nesting as there are dimensions in the array. All of the elements in the innermost nesting level must be implicitly convertible to the element type of the array. The number of elements in each nested array-element initializer must always be consistent with the size of the other array-element initializers at the same level.

The following example creates a two-dimensional array with a length of five for the leftmost dimension and a length of two for the rightmost dimension:

```
    Module Test
        Sub Main()
```

```
        Dim b As Integer(,) = _
            { { 0, 1 }, { 2, 3 }, { 4, 5 }, { 6, 7 }, { 8, 9 } }
    End Sub
End Module
```

The example is equivalent to the following:

```
Module Test
    Sub Main()
        Private b(4, 1) As Integer

        b(0, 0) = 0: b(0, 1) = 1
        b(1, 0) = 2: b(1, 1) = 3
        b(2, 0) = 4: b(2, 1) = 5
        b(3, 0) = 6: b(3, 1) = 7
        b(4, 0) = 8: b(4, 1) = 9
    End Sub
End Module
```

If the array-creation expression specifies the bounds of the dimensions, the bounds must be specified using constant expressions and the number of elements at any particular level must be the same as the size of the corresponding dimension. If the bounds are unspecified, the length of each dimension is the number of elements in the corresponding level of nesting.

Some valid and invalid examples follow:

```
' OK.
Private x() As Integer = New Integer(2) {0, 1, 2}
' Error, length/initializer mismatch.
Private y() As Integer = New Integer(2) {0, 1, 2, 3}
```

Here, the initializer for `y` is in error because the length and the number of elements in the initializer do not agree.

An empty array-element initializer (that is, one that contains curly braces but no initializer list) is always valid regardless of the number of dimensions of the array. If the size of the dimensions of the array being initialized is known in an array-creation expression, the empty array-element 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 dimensions of the array being initialized are not known, the empty array-element initializer represents an array instance in which all dimensions are size zero.

Because context is required to determine the type of an array initializer, it is not possible to use an array initializer in an expression context. Therefore, the following code is not valid:

```
Module Test
    Sub F(ByVal a() As Integer)
    End Sub

    Sub Main()
        ' Error, can't use without array creation expression.
        F({1, 2, 3})
```

```
        ' OK.
        F(New Integer() {1, 2, 3})
    End Sub
End Module
```

At run time, the expressions in an array-element initializer are evaluated in textual order from left to right.

ArrayElementInitializer ::= `{` [*VariableInitializerList*] `}`

VariableInitializerList ::=
  VariableInitializer |
  VariableInitializerList `,` *VariableInitializer*

VariableInitializer ::= *Expression* | *ArrayElementInitializer*

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. 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 (ByVal 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 (ByVal 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 the same or more restrictive than the accessibility of the property. At least one of the accessors must have the same accessibility as 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 (ByVal 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 (ByVal 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 (ByVal 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 zValue 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 (ByVal 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 zValue
            End Get
            Set (ByVal Value As Integer)
                zValue = 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 = New StreamReader(File.OpenStandardInput())
            End If
            Return reader
        End Get
    End Property

    Public Shared ReadOnly Property Out() As TextWriter
        Get
            If writer Is Nothing Then
                writer = New StreamWriter(File.OpenStandardOutput())
            End If
            Return writer
        End Get
    End Property

    Public Shared ReadOnly Property [Error]() As TextWriter
        Get
            If errors Is Nothing Then
                errors = New StreamWriter(File.OpenStandardError())
            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 created. But if the application makes no reference to the `In` and `Error` properties, then no objects are created for those devices.

PropertyMemberDeclaration ::=
  RegularPropertyMemberDeclaration |
  MustOverridePropertyMemberDeclaration

RegularPropertyMemberDeclaration ::=
  [*Attributes*] [*PropertyModifier+*] `Property` *FunctionSignature* [*ImplementsClause*]
    LineTerminator
  PropertyAccessorDeclaration+
  `End Property` *StatementTerminator*

MustOverridePropertyMemberDeclaration ::=
  [*Attributes*] [*MustOverridePropertyModifier+*] `Property` *FunctionSignature* [*ImplementsClause*]
    StatementTerminator

InterfacePropertyMemberDeclaration ::=
  [*Attributes*] [*InterfacePropertyModifier+*] `Property` *FunctionSignature* *StatementTerminator*

PropertyModifier ::= *ProcedureModifier* | `Default` | `ReadOnly` | `WriteOnly`

MustOverridePropertyModifier ::= *PropertyModifier* | `MustOverride`

InterfacePropertyModifier ::=
  `Shadows` |
  `Overloads` |
  `Default` |
  `ReadOnly` |
  `WriteOnly`

PropertyAccessorDeclaration ::= *PropertyGetDeclaration* | *PropertySetDeclaration*

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(ByVal 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(ByVal i As Integer) As Integer()
        Get
            If i = 0 Then
                F = new Integer(3) { 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 [ ( ParameterList ) ] 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(ByVal 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(ByVal 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(ByVal 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(ByVal 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`.

9.8 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. 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
    ' Error: Cannot redefine conversion from Derived to Base
    Public Shared Widening Operator CType(ByVal 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(ByVal b As Base) As Integer
        ...
    End Operator
```

```
        Public Shared Narrowing Operator CType(ByVal 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 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.

OperatorDeclaration ::=
  UnaryOperatorDeclaration |
  BinaryOperatorDeclaration |
  ConversionOperatorDeclaration

OperatorModifier ::= `Public` | `Shared` | `Overloads` | `Shadows`

Operand ::= [`ByVal`] *Identifier* [`As` *TypeName*]

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`)

  > **Note** The .NET Framework distinguishes between overloading bitwise and logical operators, while Visual Basic does not. Overloading the `Not` operator will overload only the bitwise operator from the perspective of other languages that make this distinction.

- 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 +(ByVal 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.

UnaryOperatorDeclaration ::=
  [*Attributes*] [*OperatorModifier+*] `Operator` *OverloadableUnaryOperator* `(` *Operand* `)`
    [`As` [*Attributes*] *TypeName*] *LineTerminator*
  [*Block*]
  `End Operator` *StatementTerminator*

OverloadableUnaryOperator ::= `+` | `-` | `Not` | `IsTrue` | `IsFalse`

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`)

  > **Note** The .NET Framework distinguishes between overloading bitwise and logical operators, while Visual Basic does not. Overloading the `And`, `Or` and `Xor` operators will overload only the bitwise operators from the perspective of other languages that make this distinction.

- The shift operators `<<` and `>>` (corresponding methods: `op_LeftShift`, `op_RightShift`)

  > **Note** The .NET Framework distinguishes between overloading signed and unsigned shift operators, while Visual Basic does not. Overloading the `<<` and `>>` operators will overload only the signed operators from the perspective of other languages that make this distinction.

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.

Annotation

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. Because other languages may not distinguish between division and integral division as separate operators, defining an integral division overload will automatically define a regular division operator (usable only from other languages) that will call the integral division operator.

Annotation

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 +(ByVal x As Complex, ByVal 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
```

BinaryOperatorDeclaration ::=
  [*Attributes*] [*OperatorModifier+*] `Operator` *OverloadableBinaryOperator*
    `(` *Operand* `,` *Operand* `)` [`As` [*Attributes*] *TypeName*] *LineTerminator*
  [*Block*]
  `End Operator` *StatementTerminator*

OverloadableBinaryOperator ::=
  `+` | `-` | `*` | `/` | `\` | `&` | `Like` | `Mod` | `And` | `Or` | `Xor` |
  `^` | `<<` | `>>` | `=` | `<>` | `>` | `<` | `>=` | `<=`

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 Digit(ByVal value As Byte)
      if value < 0 OrElse value > 9 Then Throw New ArgumentException()
      Me.value = value
   End Sub

   Public Shared Widening Operator CType(ByVal 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`).

ConversionOperatorDeclaration ::=
    [*Attributes*] [*ConversionOperatorModifier+*] `Operator CType (` *Operand* `)`
        [`As` [*Attributes*] *TypeName*] *LineTerminator*
    [*Block*]
    `End Operator` *StatementTerminator*

ConversionOperatorModifier ::= `Widening` | `Narrowing` | *ConversionModifier*

10. Statements

Statements represent executable code.

A method is executed by first initializing all of its parameters to their correct values and initializing all of its local variables to the default value of their types. After parameter and local variable initialization, the method body block is executed. After the method block has been executed, execution returns to the caller of the method.

Statement ::=
  LabelDeclarationStatement |
  LocalDeclarationStatement |
  WithStatement |
  SyncLockStatement |
  EventStatement |
  AssignmentStatement |
  InvocationStatement |
  ConditionalStatement |
  LoopStatement |
  ErrorHandlingStatement |
  BranchStatement |
  ArrayHandlingStatement |
  UsingStatement

10.1 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`.

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(ByVal 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.

Block ::= [*Statements+*]

LabelDeclarationStatement ::= *LabelName* **:**

LabelName ::= *Identifier* | *IntLiteral*

Statements ::=
  [*Statement*] |
  Statements **:** [*Statement*]

10.1.1 Local Variables and Parameters

A method invocation creates a copy, specific to that invocation, of the local variables and parameters of the method. A local variable or parameter comes into existence when control enters the method body that contains the local variable declaration or parameter declaration and ceases to exist when control leaves the method. 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()
        Dim i As Integer
        For i = 0 To 9
            H()
        Next I

        Dim i As Integer
        For i = 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(ByVal 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(ByVal i As Integer) As Integer()
    If i = 0 Then
        F = new Integer(3) { 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.

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 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.

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

```
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 IncompleteInitializationException()
                End If
            Finally
                xInitFlag = 1
                Monitor.Leave(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.

LocalDeclarationStatement ::= *LocalModifier* *VariableDeclarators* *StatementTerminator*

LocalModifier ::= `Static` | `Dim` | `Const`

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 has not been declared in the current method 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. 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.

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. 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.

WithStatement ::=
  `With` *Expression* *StatementTerminator*
  [*Block*]
  `End With` *StatementTerminator*

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. 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
```

SyncLockStatement ::=
  `SyncLock` *Expression* *StatementTerminator*
  [*Block*]
  `End SyncLock` *StatementTerminator*

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. 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 Constructed(ByVal Count As Integer)

    Public Sub New()
        Static CreationCount As Integer = 0
```

```
            CreationCount += 1
            RaiseEvent Constructed(CreationCount)
        End Sub
    End Class

    Module Test
        Private WithEvents x As Raiser

        Private Sub Constructed(ByVal Count As Integer) Handles x.Constructed
            Console.WriteLine("Constructed instance #" & Count)
        End Sub

        Public Sub Main()
            x = New Raiser    ' Causes "Constructed instance #1" to be printed.
            x = New Raiser    ' Causes "Constructed instance #2" to be printed.
            x = New Raiser    ' Causes "Constructed instance #3" to be printed.
        End Sub
    End Module
```

The class `Raiser` above is equivalent to:

```
    Class Raiser
        Public Event Constructed(ByVal Count As Integer)

        Public Sub New()
            Static CreationCount As Integer = 0
            Dim TemporaryDelegate As ConstructedEventHandler

            CreationCount += 1

            ' Use a temporary to avoid a race condition.
            TemporaryDelegate = ConstructedEvent
            If Not TemporaryDelegate Is Nothing Then
                TemporaryDelegate.Invoke(CreationCount)
            End If
        End Sub
    End Class
```

RaiseEventStatement ::= `RaiseEvent` *IdentifierOrKeyword* [`(` [*ArgumentList*] `)`]
StatementTerminator

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.

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
```

AddHandlerStatement ::= `AddHandler` *Expression* `,` *Expression* *StatementTerminator*

RemoveHandlerStatement ::= `RemoveHandler` *Expression* `,` *Expression* *StatementTerminator*

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. 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() ' 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 (ByVal 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 associated instance expression 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 (ByVal 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 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.

Annotation

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()`.

RegularAssignmentStatement ::= *Expression* `=` *Expression* *StatementTerminator*

10.6.2 Compound Assignment Statements

A *compound assignment statement* takes the form *V OP= E* (where *OP* is a valid binary operator). 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
```

The rule for predefined operators is simply that *V OP= E* is permitted if both *V OP E* and *V = E* are permitted. Thus, in the following example, the reason for each error is that a corresponding simple assignment would also have been an error.

```
Option Strict On
Module Test
    Private b As Byte = 0
    Private ch As Char = ControlChars.NullChar
    Private i As Integer = 0
    Sub Main()
        b += 1 ' This is allowed.
        b += 1000 ' Error; b = 1000 is not permitted.
        b += i ' Error, b = i is not permitted.
        b += CByte(i) ' This is allowed.
        ch += 1 ' Error, ch = 1 is not permitted.
        ch += ChrW(1) ' This is allowed.
    End Sub
End Module
```

CompoundAssignmentStatement ::= *Expression CompoundBinaryOperator Expression StatementTerminator*

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

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`. 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.

MidAssignmentStatement ::=
`Mid` [`$`] `(` *Expression* `,` *Expression* [`,` *Expression*] `)` `=` *Expression* *StatementTerminator*

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. The invocation expression must be classified as a value or void. Any value resulting from the evaluation of the invocation expression is discarded.

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. Each expression in an `If...Then...Else` statement must be classified as a value and be implicitly convertible to `Boolean`. 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
```

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
```

10.8.2 Select...Case Statements

A `Select Case` statement executes statements based on the value of an expression. 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.

```
SelectStatement ::=
    Select [ Case ] Expression StatementTerminator
    [ CaseStatement+ ]
    [ CaseElseStatement ]
    End Select StatementTerminator

CaseStatement ::=
    Case CaseClauses StatementTerminator
    [ Block ]

CaseClauses ::=
    CaseClause |
    CaseClauses , CaseClause

CaseClause ::=
    [ Is ] ComparisonOperator Expression |
    Expression [ To Expression ]

ComparisonOperator ::= = | <> | < | > | => | =<

CaseElseStatement ::=
    Case Else StatementTerminator
    [ Block ]
```

10.9 Loop Statements

Loop statements allow repeated execution of statements.

```
LoopStatement ::=
    WhileStatement |
    DoLoopStatement |
    ForStatement |
    ForEachStatement
```

10.9.1 While...End While and Do...Loop Statements

A `While` or `Do` loop statement loops based on a `Boolean` expression. A `While` loop statement loops as long as the `Boolean` expression evaluates to `True`; a `Do` loop statement may contain a more complex condition. In either case, the expressions must be classified as values and must be implicitly convertible to `Boolean`.

An expression may be placed after the `Do` keyword or after the `Loop` keyword, but not after both. 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
Third Loop
Third 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`.

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`

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 `As` clause or an expression. In the case of an identifier, the identifier defines a new local variable of the type specified in the `As` clause, scoped to the entire `For` loop. In the case of an expression, the expression must be classified as a variable.

The loop control variable of a `For` statement must be of a primitive numeric type (`Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single`, `Double`), `Object`, or a type `T` that has the following operators:

```
Public Shared Operator >= (ByVal op1 As T, ByVal op2 As T) As B
Public Shared Operator <= (ByVal op1 As T, ByVal op2 As T) As B
Public Shared Operator - (ByVal op1 As T, ByVal op2 As T) As T
Public Shared Operator + (ByVal op1 As T, ByVal op2 As T) As T
```

Where `B` is a type that can be used in a Boolean expression (i.e. is convertible to `Boolean` or overloads `IsTrue` and `IsFalse`).

The bound and step expressions must be implicitly convertible to the type of the loop control. Enumerated types are treated as their underlying type, which allows use of enumerated types in `For` loops. The bounds and step expressions must be classified as values.

A loop control variable cannot be used by another enclosing `For...Next` statement. A `For` statement must be closed by a matching `Next` statement. A `Next` statement without a variable matches the innermost open `For` statement. 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 control value. If the step value is omitted, it is implicitly the literal `1`. The three expressions are only ever evaluated at the beginning of the loop.

If the `For` loop variable is of type `Object`, then the loop control type is determined at run time as follows:

- If at least one of the expressions at run time is a primitive numeric type or enumerated type, then the loop control type is the widest numeric type among them.
    - If one or more of the types was an enumerated type, all the enumerated types are the same, and the underlying type of the common enumerated type is the widest numeric type, then the loop control type is the enumerated type.
- Otherwise, if all three expressions are typed as `String`, then the loop control type is `Double`.
- Otherwise, if the most encompassing type of the three expressions implements the overloaded operators listed above, then the most encompassing type is the loop control type.

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.

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.

It is not valid to branch into a `For` loop from outside the loop.

ForStatement ::=
 `For` *LoopControlVariable* `=` *Expression* `To` *Expression* [`Step` *Expression*] *StatementTerminator*
 [*Block*]
 `Next` [*NextExpressionList*] *StatementTerminator*

LoopControlVariable ::=
 Identifier [*ArrayNameModifier*] `As` *TypeName* |
 Expression

NextExpressionList ::=
 Expression |
 NextExpressionList `,` *Expression*

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 `As` clause or an expression. In the case of an identifier, the identifier defines a new local variable of the type specified in the `As` clause, scoped to the entire `For Each` loop. In the case of an expression, the expression must be classified as a variable. 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:

- All of the following are true:
  - `C` contains an accessible instance method with the signature `GetEnumerator()` that returns a type `E`.
  - `E` contains an accessible instance method with the signature `MoveNext()` and the return type `Boolean`.
  - `E` contains an accessible instance 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(ByVal 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 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.IEnumerable(Of T)`. If the expression type implements both the generic and non-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.

Annotation

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 loop, `MoveNext` is called on the temporary. If it returns `False`, the loop terminates. If it returns `True`, the `Current` property is retrieved, coerced to the type of the iterator variable (regardless of whether the conversion is implicit or explicit), and assigned to the iterator variable; then the loop block executes.

Annotation

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 most common collection 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 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.

ForEachStatement ::=
    `For Each` *LoopControlVariable* `In` *Expression* *StatementTerminator*
    [*Block*]
    `Next` [*Expression*] *StatementTerminator*

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.

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.

StructuredErrorStatement ::=
  ThrowStatement |
  TryStatement

TryStatement ::=
  `Try` *StatementTerminator*
  [*Block*]
  [*CatchStatement+*]
  [*FinallyStatement*]
  `End Try` *StatementTerminator*

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.

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. 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 `Object` (or `System.Object`), `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`.

Annotation

Allowing exceptions derived from `Object` to be caught is necessary to handle the fact that some languages (such as C++) allow any object to be thrown. Because the more common case is that an exception will derive from `System.Exception`, that is the default case when no type is specified.

A `Catch` clause with a `When` clause will only catch exceptions when the expression evaluates to `True`; the type of the expression must be implicitly convertible to `Boolean`. 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.

CatchStatement ::=
  `Catch` [*Identifier* `As` *NonArrayTypeName*] [`When` *BooleanExpression*] *StatementTerminator*
  [*Block*]

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`. 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 reference 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(ByVal x As Integer)
    Try
        Throw New Exception()
    Catch
        If x = 0 Then
            Throw    ' OK, rethrows exception from above.
        Else
            Try
                If x = 1 Then
                    Throw    ' OK, rethrows exception from above.
                End If
            Finally
                Throw    ' Invalid, inside of a Finally.
            End Try
        End If
    End Try
End Sub
```

ThrowStatement ::= `Throw` [*Expression*] *StatementTerminator*

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. 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`.

UnstructuredErrorStatement ::=
  ErrorStatement |
  OnErrorStatement |
  ResumeStatement

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. 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.
- `On Error Resume Next`, establishes the `Resume Next` behavior as the most recent exception-handler location.

OnErrorStatement ::= `On Error` *ErrorClause* *StatementTerminator*

ErrorClause ::=
  `GoTo - 1` |
  `GoTo 0` |
  GotoStatement |
  `Resume Next`

10.10.2.3 Resume Statement

A `Resume` statement returns execution to the statement that caused the most recent exception. 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 = False
        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).

ResumeStatement ::= `Resume` [*ResumeClause*] *StatementTerminator*

ResumeClause ::= `Next` | *LabelName*

10.11 Branch Statements

Branch statements modify the flow of execution in a method. There are six branch statements:

- A `GoTo` statement causes execution to transfer to the specified label in the method.
- 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, execution is transferred back to the expression that invoked the method. If the `Exit` statement is not contained within the kind of block specified in the statement, a compile-time error occurs.
- 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.
- A `Stop` statement causes a debugger exception to occur.
- 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).
- A `Return` statement returns execution to the expression that invoked the method. If the method is a subroutine, the statement is equivalent to an `Exit Sub` statement and no expression may be supplied. If the method is a function, an expression must be supplied that is classified as a value and whose type is implicitly convertible to the return type of the function. This form is equivalent to assigning to the function return local and then executing an `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*]

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. 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 `System.InvalidCastException` will be thrown. 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) {}

        ' The following statement throws an InvalidCastException.
        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(3, 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 null 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.

RedimStatement ::= `ReDim` [`Preserve`] *RedimClauses* *StatementTerminator*

RedimClauses ::=
  RedimClause |
  RedimClauses `,` *RedimClause*

RedimClause ::= *Expression* *ArraySizeInitializationModifier*

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 |
  EraseExpressions `,` *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("foo")

        ' 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 declared as a part of the statement itself; in the other, the resource is the result of an expression.

If the resource is 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 Foo = New Foo()
    ...
End Using
```

is equivalent to:

```
Dim f As Foo = New Foo()
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
```

UsingStatement ::=
   `Using` *UsingResources* *StatementTerminator*
      [*Block*]
   `End Using` *StatementTerminator*

UsingResources ::= *VariableDeclarators* | *Expression*

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 |
    IndexExpression |
    NewExpression |
    CastExpression |
    OperatorExpression

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 instance expression and an associated type argument list.
- A method pointer, which represents the location of a method. A method pointer may have an associated instance expression and an associated type argument list.
- A property group, which is a set of properties overloaded on the same name. A property group may have an associated instance expression.
- A property access. Every property access has an associated type, namely the type of the property. A property access may have an associated instance expression.
- A late-bound access, which represents a method or property access deferred until run-time. A late-bound access may have an associated instance 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 instance 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.
- Void. This occurs when the expression is an invocation of a subroutine. An expression classified as void is only valid in the context of an invocation statement.

The final result of an expression is never a namespace, type, method group, or property group. Rather, as noted above, these categories of expressions are intermediate constructs 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*. 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 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 actually result in the expression being reclassified as void.
- A method pointer can be reclassified as a value. This reclassification can only occur in assignment statements or as a part of interpreting a parameter list, 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(ByVal i As Integer)

Module Test
    Sub F(ByVal 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 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.

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(ByVal i As Integer)
    End Sub

    ReadOnly Property P() As Integer
        Get
        End Get
    End Sub

    Sub Main()
        F(P)
    End Sub
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. The type of a constant expression can be `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Char`, `Single`, `Double`, `Decimal`, `Boolean`, `String`, 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 not allowed because `String` conversions are always done in the current culture of the execution environment at run time.
- The `+`, `–` and `Not` unary operators.
- The `+`, `–`, `*`, `^`, `Mod`, `/`, `\`, `<<`, `>>`, `&`, `And`, `Or`, `Xor`, `AndAlso`, `OrElse`, `=`, `<`, `>`, `<>`, `<=`, and `=>` binary operators, provided each operand 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

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.

ConstantExpression ::= *Expression*

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 instance 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.

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(ByVal b As Base)
        Console.WriteLine("F(Base)")
    End Sub

    Sub F(ByVal 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.

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 ::= `(` *Expression* `)`

11.4.3 Instance Expressions

An *instance expression* is the keyword `Me`, `MyClass`, or `MyBase`. An instance expression, which may only be used within the body of a non-shared method, constructor, or property accessor, is classified as a value. Instance expressions cannot be used in the arguments to constructor invocations. The three different instance expression types are:

- The keyword `Me` represents the instance of the type containing the method or property accessor being executed.
- The keyword `MyClass` is equivalent to `Me`, but all method invocations on it are treated as if the method being invoked is non-overridable. Thus, the method called will not be affected by the run-time type of the value on which the method is being called. `MyClass` can only be used as the target expression of a member access.
- The keyword `MyBase` represents the instance of the type containing the method or property accessor being executed cast to its direct base type. All method invocations on it are treated as if the method being invoked

is non-overridable. `MyBase` can only be used as the target expression of a member access. A member access using `MyBase` is called a *base access*.

The following example demonstrates how instance expressions can be used:

```
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
```

InstanceExpression ::= `Me`

11.4.4 Simple Name Expressions

A *simple name expression* consists of a single identifier followed by an optional type argument list. The name is resolved and classified as follows:

- 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. 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 with the following exception. If 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.
- 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:
  - If the matching type member is a type parameter, then the result is classified as a type and is the matching type parameter.
  - 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`, where `E` is the identifier.
  - Otherwise, the result is exactly the same as a member access of the form `T.E`, where `T` is the type containing the matching member and `E` is the identifier. In this case, it is an error for the identifier to refer to a non-shared member.
- For each nested namespace, starting from the innermost and going to the outermost namespace, do the following:
  - If the namespace contains an accessible namespace member with the given name, then the identifier refers to that member and, depending on the member, is classified as a namespace or a type.
  - 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`, where `M` is the standard module containing the matching member and `E` is the identifier. If the identifier matches accessible type members in more than one standard module, a compile-time error occurs.
- 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 the source file containing the name reference has one or more imports:

  - If the identifier matches the name of an accessible type or type member in exactly one import, then the identifier refers to that type or type member. If the identifier matches the name of an accessible type or type member in more than one import, a compile-time error occurs.
  - If the identifier matches the name of a namespace in exactly one import, then the identifier refers to that namespace. If the identifier matches the name of a namespace in more than one import, a compile-time error occurs.
  - 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`, where `M` is the standard module containing the matching member and `E` is the identifier. If the identifier matches accessible type members in more than one standard module, a compile-time error occurs.
- 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 the compilation environment defines one or more imports:
  - If the identifier matches the name of an accessible type or type member in exactly one import, then the identifier refers to that type or type member. If the identifier matches the name of an accessible type or type member in more than one import, a compile-time error occurs.
  - If the identifier matches the name of a namespace in exactly one import, then the identifier refers to that namespace. If the identifier matches the name of a namespace in more than one import, a compile-time error occurs.
  - 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`, where `M` is the standard module containing the matching member and `E` is the identifier. If the identifier matches accessible type members in more than one standard module, a compile-time error occurs.
- Otherwise, the name given by the identifier is undefined and a compile-time error occurs.

If a simple name with a type argument list resolves to anything other than a type or method, a compile time error occurs. If a type argument list is supplied, only types with the same arity as the type argument list are considered but type members, including methods with different arities, *are* still considered. This is because type inference can be used to fill in missing type arguments. As a result, names with type arguments may bind differently to types and methods:

```
Class Outer
    Class C1(Of T)
    End Class

    Shared Sub S1(Of T)()
    End Sub

    Class Inner
        Class C1
        End Class

        Sub S1()
```

```
        End Sub

        Sub Test()
            Dim x As C1(Of Integer)    ' Binds to Outer.C1(Of T)
            S1(Of Integer)()           ' Error: Outer.Inner.S1 takes
                                       '        no type arguments
        End Sub
    End Class
End Class
```

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.

SimpleNameExpression ::= *Identifier* [`(` `Of` *TypeArgumentList* `)`]

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. The method group cannot be late-bound and it cannot refer to constructors.

The result is classified as a method pointer and the associated instance expression and type argument list (if any) is the same as the associated instance expression and type argument list of the method group.

AddressOfExpression ::= `AddressOf` *Expression*

11.5 Type Expressions

A *type expression* is a `GetType` expression, a `TypeOf...Is` expression, or an `Is` expression.

TypeExpression ::=
    GetTypeExpression |
    TypeOfIsExpression |
    IsExpression

11.5.1 GetType Expressions

A `GetType` expression consists of the keyword `GetType` and the name of a type. A type expression is classified as a value, and the value of the expression is the reflection (`System.Type)` class that represents that type. If the expression 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 type name in a `GetType` expression is special in two ways:

- The type name is allowed to be `System.Void`, the only place in the language where this type name may be referenced.
- The type name 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()
```

GetTypeExpression ::= `GetType` `(` *GetTypeTypeName* `)`

GetTypeTypeName ::=
    TypeName |
    QualifiedIdentifier `(` `Of` [*TypeArityList*] `)`

TypeArityList ::=
    `,` |
    TypeParameterList `,`

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 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 is derived from or implements the type, `False` otherwise.

TypeOfIsExpression ::= `TypeOf` *Expression* `Is` *TypeName*

11.5.3 Is Expressions

An `Is` or `IsNot` expression is used to do a reference equality comparison. Each expression must be classified as a value and the type of each expression must be a reference type or an unconstrained type parameter type. If the type of one expression is an unconstrained type parameter 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 `False` otherwise. An `IsNot` operation evaluates to `False` if both values refer to the same instance or `True` otherwise.

IsExpression ::=
    Expression `Is` *Expression* |
    Expression `IsNot` *Expression*

11.6 Member Access Expressions

A member access expression is used to access a member of an entity. A member access of the form `E.I`, where `E` is an expression, a built-in type, the keyword `Global`, or omitted and `I` is an identifier with an optional type argument list, is evaluated and classified as follows:

- 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.
- If `E` is a type parameter, then a compile-time error results.
- If `E` is the keyword `Global`, and `I` is the name of an accessible type in the global namespace, then the result is that type.
- If `E` is classified as a namespace and `I` is the name of an accessible member of that namespace, then the result is that member. The result is classified as a namespace or a type depending on the member.
- If `E` is a built-in type or an expression classified as a type, and `I` is the name of an accessible member of `E`, then `E.I` is evaluated and classified as follows:
  - If `I` is the keyword `New`, then a compile-time error occurs.
  - If `I` identifies a type, then the result is that type.
  - If `I` identifies one or more methods, then the result is a method group with the associated type argument list and no associated instance expression.
  - If `I` identifies one or more properties, then the result is a property group with no associated instance expression.
  - If `I` identifies a shared variable, and 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`.
  - If `I` identifies a shared event, the result is an event access with no associated instance expression.
  - If `I` identifies a constant, then the result is the value of that constant.
  - If `I` identifies an enumeration member, then the result is the value of that enumeration member.
  - Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.
- If `E` is classified as a variable or value, the type of which is `T`, and `I` is the name of an accessible member of `E`, then `E.I` is evaluated and classified as follows:
  - If `I` is the keyword `New` and `E` is an instance expression (`Me`, `MyBase`, or `MyClass`), then the result is a method group representing the instance constructors of the type of `E` with an associated instance expression of `E` and no type argument list. Otherwise, a compile-time error occurs.
  - If `I` identifies one or more methods, then the result is a method group with the associated type argument list and an associated instance expression of `E`.
  - If `I` identifies one or more properties, then the result is a property group with an associated instance expression of `E`.
  - If `I` identifies a shared variable or an instance variable, and 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.

- If `I` identifies an event, the result is an event access with an associated instance expression of `E`.
- If `I` identifies a constant, then the result is the value of that constant.
- If `I` identifies an enumeration member, then the result is the value of that enumeration member.
- If `T` is `Object`, then the result is a late-bound member lookup classified as a late-bound access with an associated instance expression of `E`.

- Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.

If the member access expression includes a type argument list, then only types or methods with the same arity as the type argument list are considered.

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`.

MemberAccessExpression ::=
  [[*MemberAccessBase*] `.`] *IdentifierOrKeyword*

MemberAccessBase ::=
  Expression |
  BuiltInTypeName |
  `Global` |
  `MyClass` |
  `MyBase`

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 Class
```

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 prints 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 Class
```

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 Class
```

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 Class
```

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 non-generic classes that are declared as deriving from the group's base type 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.

The signature of the create method must be of the form `Shared Function <Name>(Of T As {New, <Type>})(ByVal 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.MyGroupAttribute("Form", "Create", _
    "Dispose", "My.Forms")> _
Public NotInheritable Class MyForms
    Private Shared Function Create(Of T As {New, Form}) _
        (ByVal Instance Of 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 Of 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.MyGroupAttribute("Form", "Create", _
    "Dispose", "My.Forms")> _
Public NotInheritable Class MyForms
    Private Shared Function Create(Of T As {New, Form}) _
        (ByVal Instance Of 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 Of T)
        Instance.Close()
        Instance = Nothing
    End Sub

    Private m_Form1 As Form1

    Public Property Form1() As Form1
        Get
            Return CreateInstance(m_Form1)
        End Get
        Set (ByVal 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
            DisposeInstance(m_Form1)
        End Set
    End Property
End Class
```

11.7 Dictionary Member Access

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. 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(ByVal 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.

DictionaryAccessExpression ::= [*Expression*] `!` *IdentifierOrKeyword*

11.8 Invocation Expressions

An invocation expression consists of an invocation target and an optional argument list. 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 referring to the `Invoke` member of the delegate type.

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.

Given a method group, overload resolution is applied to pick a single method applicable to the given argument list(s). If the method group only contains one method 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 index expression. If no method is applicable, a compile-time error occurs. If the 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 applicable method is a subroutine, then the result is classified as void.

InvocationExpression ::= *Expression* [`(` [*ArgumentList*] `)`]

ArgumentList ::=
    PositionalArgumentList `,` *NamedArgumentList* |
    PositionalArgumentList |
    NamedArgumentList

PositionalArgumentList ::=
    Expression |
    PositionalArgumentList `,` [*Expression*]

NamedArgumentList ::=
    IdentifierOrKeyword `:=` *Expression* |
    NamedArgumentList `,` *IdentifierOrKeyword* `:=` *Expression*

11.8.1 Overloaded Method Resolution

Given a method group, the applicable method in the group for an argument list is determined as follows:

- Eliminate all inaccessible members from the set.
- Eliminate all members from the set that are not applicable to the argument list. If the set is empty, a compile-time error results. If only one member remains in the set, that is the applicable member.

- Eliminate all members from the set that require narrowing coercions to be applicable to the argument list, except for the case where the argument expression type is `Object`. If the set is empty, a compile-time error results. If only one member remains in the set, that is the applicable member.
- Eliminate all remaining members from the set that require narrowing coercions to be applicable to the argument list. If the set is empty, the type containing the method group is not an interface, and strict semantics are not being used, the invocation target expression is reclassified as a late-bound method access. If strict semantics are being used or the method group is contained in an interface and the set is empty, a compile-time error results. If only one member remains in the set, that is the applicable member.

Annotation

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 because all 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.

- Given any two members of the set, `M` and `N`, if `M` is more applicable than `N` to the argument list, eliminate `N` from the set. If only one member remains in the set, that is the applicable member. If the remaining members do not all have the same signature, a compile-time error results.
- Otherwise, it must be the case that the remaining members have the same signature because of type parameters. Given any two members of the set, `M` and `N`, if `M` is less generic than `N`, eliminate `N` from the set. If only one member remains in the set, that is the applicable member. Otherwise, a compile-time error results.

A member `M` is considered *more applicable* than `N` if their signatures are different and, for each pair of parameters `Mj` and `Nj` that matches an argument `Aj`, one of the following conditions is true:

- `Mj` and `Nj` have identical types, or
- There exists a widening conversion from the type of `Mj` to the type `Nj`, or

Annotation

Note that 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, or

Annotation

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` is `Byte` and `Nj` is `SByte`, or
- `Mj` is `Short` and `Nj` is `UShort`, or

- `Mj` is `Integer` and `Nj` is `UInteger`, or
- `Mj` is `Long` and `Nj` is `ULong`.

Annotation

The rules about the numeric types are necessary because the signed and unsigned numeric types of a particular size only have narrowing conversions between them. Thus, the above rules break 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).

A member `M` is determined to be *less generic* than a member `N` using the following steps:

- If `M` has fewer method type parameters than `N`, then `M` is less generic than `N`.
- Otherwise, if for each pair of matching parameters `Mj` and `Nj`, `Mj` and `Nj` are equally generic with respect to type parameters on the method, or `Mj` is less generic with respect to type parameters on the method, and at least one `Mj` is less generic than `Nj`, then `M` is less generic than `N`.
- Otherwise, if for each pair of matching parameters `Mj` and `Nj`, `Mj` and `Nj` are equally generic with respect to type parameters on the type, or `Mj` is less generic with respect to type parameters on the type, and at least one `Mj` is less generic than `Nj`, then `M` is less generic than `N`.

For example:

```
Class C1(Of T)
    Sub S1(x As T)
    End Sub

    Sub S1(Of U)(x As T)
    End Sub

    Sub S2(x As Integer)
    End Sub

    Sub S2(x As T)
    End Sub

    Sub S2(Of U)(x As U)
    End Sub

    Sub S3(x As T)
    End Sub

    Sub S3(Of U)(x As U)
    End Sub
End Class
```



```
Module Test
    Sub Main()
        Dim x As C1(Of Integer) = New C1(Of Integer)

        x.S1(10)    ' Calls S1(T), as it has fewer type parameters
        x.S2(10)    ' Calls S2(Integer), as it is less generic
        x.S3(10)    ' Calls S3(T), as it is less generic than S3(Of U)(U)
    End Sub
End Module
```

In practice, the rules for determining whether one method is more applicable than another method 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 wider than (or the same as) the parameter types of the other method. If neither method's parameters are wider than the other, then there is no way for to determine which method is closer to the arguments.

Note The context of the method usage is not used in overload resolution (except for determining accessibility).

Annotation

The context of the method call is not generally considered when doing overload resolution to avoid adding even more complexity to an already complex algorithm. The ability for mere mortals to grasp the results of overload resolution is already tenuous one, but adding a myriad of contextual clues would make it impossible. This does mean that in some situations overload resolution will choose the "wrong" overload, such as when it picks an instance method when doing overload resolution off of a type name.

Also note that because of the named parameter syntax, the ordering of the actual and formal parameters may not be the same.

11.8.2 Applicable Methods

A method is *applicable* to a set of type arguments, positional arguments, and named arguments if the method can be invoked using the two argument lists. The argument lists are matched against the parameter lists as follows:

- 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, the method is not applicable.
- 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.
- Next, if parameters that have not been matched are not optional, the method is not applicable. If optional parameters remain, the default value specified in the optional parameter declaration is matched to the parameter. If an `Object` parameter does not specify a default value, then the expression `System.Reflection.Missing.Value` is used. If an optional `Integer` parameter has the `Microsoft.VisualBasic.CompilerServices.OptionCompareAttribute` attribute, then the literal `1` is supplied for text comparisons and the literal `0` otherwise.

- Finally, 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 inferencing 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.

The type of each argument expression must be implicitly convertible to the type of the parameter it matches. If the parameter is a reference parameter, the argument expression must be also implicitly convertible from the type of the parameter. Note that constraints placed on type parameters are not considered when determining applicability.

It is possible for two applicable methods to have the same signature if one or both contains an expanded paramarray parameter. In that case, the member with the fewest number of arguments matching expanded paramarray parameters is considered more applicable.

```
Module Test
    Sub F(ByVal ParamArray a As Object())
        Console.WriteLine("F(Object())")
    End Sub

    Sub F()
        Console.WriteLine("F()")
    End Sub

    Sub F(ByVal a As Object, ByVal b As Object)
        Console.WriteLine("F(Object, Object)")
    End Sub

    Sub F(ByVal a As Object, ByVal b As Object, _
          ByVal ParamArray c As Object())
        Console.WriteLine("F(Object, Object, Object())")
    End Sub

    Sub Main()
        F()
        F(1)
        F(1, 2)
        F(1, 2, 3)
    End Sub
End Module
```

The above example produces the following output:

```
F()
F(Object())
F(Object, Object)
F(Object, Object, Object())
```


In the example, the first and third calls to `F` prefer `F()` and `F(Object, Object)` because they match no arguments to expanded paramarray parameters. The fourth call to `F` prefers `F(Object, Object, Object())` because two arguments match non-expanded paramarray parameters. 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.

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 null 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 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.

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 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)(ByVal b As Boolean, ByVal first As T, _
        ByVal 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, "foo", "bar")
```

Through type inference, the type arguments `Integer` and `String` are determined from the arguments to the method.

Given a set of regular arguments matched to regular parameters, type inference inspects each argument type `A` and its corresponding parameter type `P` to infer type arguments. Each pair `(A, P)` is analyzed as follows:

- Nothing is inferred from the argument (but type inference succeeds) if any of the following are true:
  - `P` does not involve any method type parameters.
  - The argument is the `Nothing` literal.
  - The argument is a method group.
- If `P` is an array type and `A` is an instantiation of `IList(Of T)`, `ICollection(Of T)`, or `IEnumerable(Of T)`, then replace `P` with the element type of `P`, and `A` with the type argument of `A` and restart the process.
- If `P` is an array type and `A` is an array type of the same rank, then replace `A` and `P` respectively with the element types of `A` and `P` and restart this process.
- If `P` is an array type and `A` is not an array type of the same rank, then type inference fails for the generic method.
- If `P` is a method type parameter, then type inference succeeds for this argument, and `A` is the type inferred for that type parameter.
- Otherwise, `P` must be a constructed type. If, for each method type parameter `MX` that occurs in `P`, exactly one type `TX` can be determined such that replacing each `MX` with each `TX` produces a type to which `A` is convertible by a standard implicit conversion, then inference succeeds for this argument, and each `TX` is the type inferred for each `MX`. Method type parameter constraints, if any, are ignored for the purpose of type inference. If, for a given `MX`, no `TX` exists or more than one `TX` exists, then type inference fails for the generic method (a situation where more than one `TX` exists can only occur if `P` is a generic interface type and `A` implements multiple constructed versions of that interface).

If a parameter is a paramarray, then type inference is first performed against the parameter in its normal form. If type inference fails, then type inference is performed against the parameter in its expanded form.

If all of the method arguments have been processed successfully by the above algorithm, all inferences that were produced from the arguments are pooled. This pooled set of inferences must have the following properties:

- Every type parameter of the method must have a matching inferred type argument.
- If a type parameter occurred more than once, all of the inferences about that type parameter must infer the same type argument.

If these two properties hold, then type inference succeeds, else it fails. 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. 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 instance 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 instance expression is either the target expression, if it is a value, or the associated instance 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.

IndexExpression ::= *Expression* `(` [*ArgumentList*] `)`

11.10 New Expressions

The `New` operator is used to create new instances of types. There are three 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 are used to create new instances of delegate types.

A `New` expression is classified as a value and the result is the new instance of the type.

NewExpression ::=
    ObjectCreationExpression |

ArrayCreationExpression |
DelegateCreationExpression

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. 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.

ObjectCreationExpression ::=
`New` *NonArrayTypeName* [`(` [*ArgumentList*] `)`]

11.10.2 Array-Creation Expressions

An array-creation expression is used to create a new instance of an array type. 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 array-element initializer in the expression is not empty, 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 array element initializer.

```
Dim a() As Integer = New Integer(2) {}
Dim b() As Integer = New Integer(2) { 1, 2, 3 }
Dim c() As Integer = New Integer(2, 2) { { 1, 2, 3 } , { 4, 5, 6 } }
```

If an array size initialization modifier is not supplied, then the type name must be an array type and the array element initializer must be empty or the rank of the specified array type must match that of the array element initializer. The individual dimension lengths are inferred from the number of elements in each of the corresponding nesting levels of the array element initializer. If the array-element initializer is empty, the length of each dimension is zero.

```
Dim d() As Integer = New Integer() { 1, 2, 3 }
Dim e() As Integer = New Integer(,) { { 1, 2, 3 } , { 4, 5, 6 } }
```

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. Elements of arrays created by array-creation expressions are always initialized to their default value.

ArrayCreationExpression ::=
`New` *NonArrayTypeName* *ArraySizeInitializationModifier* *ArrayElementInitializer*

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.

One of the methods referenced by the method pointer must exactly match the signature of the delegate type. A method matches the delegate type if the method is not declared `MustOverride` and if both their signatures and return types match. In the following example, the `A.f` variable is initialized with a delegate that refers to the second `Square` method because that method exactly matches the signature and return type of `DoubleFunc`.

```
Delegate Function DoubleFunc(x As Double) As Double
Class A
    Private f As New DoubleFunc(AddressOf Square)
    Overloads Shared Function Square(x As Single) As Single
        Return x * x
    End Function
    Overloads Shared Function Square(x As Double) As Double
        Return x * x
    End Function
End Class
```

Had the second `Square` method not been present, a compile-time error would have occurred.

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(ByVal s As String, ByVal i As Integer) As Integer
Delegate Function E() As Integer

Module Test
    Public Function F(Of T)(ByVal s As String, ByVal 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)(ByVal u1 As U, ByVal 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
```

The result of a delegate-creation expression is a delegate instance that refers to the matching method with the associated instance expression (if any) from the method pointer expression. If the instance 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.

DelegateCreationExpression ::= `New` *NonArrayTypeName* `(` *Expression* `)`

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.

`DirectCast` and `TryCast` have special behaviors. Because of their special behavior, they only support a subset of the conversions supported by the language. Specifically, the conversions supported by `DirectCast` are:

- Conversions from any type to itself.
- Conversions from the literal `Nothing` to any type.
- Conversions from any derived type to one of its base types, and vice versa.
- Conversions from any interface type to any class type, and vice versa.
- Conversions from any interface type to any value type that implements the interface type, and vice versa.
- Conversions from any interface type to any other interface type.
- Conversions from any enumerated type to its underlying type, and vice versa.
- Conversions from an array type `S` with an element type `SE` to a covariant-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.
    - A conversion exists from `SE` to `TE`.
- Conversions 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`.

- Conversions 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`.

The conversions supported by `TryCast` are the same as `DirectCast`, except the type of the expression or the target type cannot be a value type. User-defined conversion operators are not considered when `DirectCast` or `DirectCast` is used.

Annotation

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`.

Annotation

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. For example:

```
Interface ITest
    Sub Test()
End Interface

Module Test
    Sub Convert(ByVal o As Object)
        Dim i As ITest = TryCast(o, ITest)

        If i IsNot Nothing Then
            i.Test()
        End If
    End Sub
End Module
```

Annotation

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`.

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.

CastExpression ::=
  `DirectCast (` *Expression* `,` *TypeName* `)` |
  `TryCast (` *Expression* `,` *TypeName* `)` |
  `CType (` *Expression* `,` *TypeName* `)` |
  CastTarget `(` *Expression* `)`

CastTarget ::=
  `CBool` | `CByte` | `CChar` | `CDate` | `CDec` | `CDbl` | `CInt` | `CLng` | `CObj` | `CSByte` | `CShort` |
  `CSng` | `CStr` | `CUInt` | `CULng` | `CUShort`

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

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
Exponentiation	`^`
Unary negation	`+`, `-`
Multiplicative	`*`, `/`
Integer division	`\`
Modulus	`Mod`
Additive	`+`, `-`
Concatenation	`&`

Shift	`<<`, `>>`
Relational	`=`, `<>`, `<`, `>`, `<=`, `>=`, `Like`, `Is`
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 late-bound method calls made on `Object` values: 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 equality (`=`), inequality (`<>`), `TypeOf...Is`, and `Is` operators.

When given one or more operands of type `Object`, 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:

```
Short = 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:

- 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.
- Then, overload resolution is applied to the operators and operands to select the most specific operator.

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 pre-defined operators will be considered. If no pre-defined operator is defined for the operands, then a compile-time error results.

Each operator lists the pre-defined types it is defined for and the type of the operation performed given the operand types. The result of type of a pre-defined 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*.

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.

ArithmeticOperatorExpression ::=
  UnaryPlusExpression |
  UnaryMinusExpression |
  AdditionOperatorExpression |
  SubtractionOperatorExpression |
  MultiplicationOperatorExpression |
  DivisionOperatorExpression |
  ModuloOperatorExpression |
  ExponentOperatorExpression

11.13.1 Unary Plus Operator

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
SB	SB	By	Sh	US	In	UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob

UnaryPlusExpression ::= `+` *Expression*

11.13.2 Unary Minus Operator

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
SB	SB	Sh	Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob

UnaryMinusExpression ::= `-` *Expression*

11.13.3 Addition Operator

The addition operator computes the sum of the two operands. 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.

  Note 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	SB	SB	Sh	Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
SB		SB	Sh	Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
By			By	Sh	US	In	UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob
Sh				Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
US					US	In	UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob
In						In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
UI							UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob
Lo								Lo	De	De	Si	Do	Err	Err	Do	Ob
UL									UL	De	Si	Do	Err	Err	Do	Ob
De										De	Si	Do	Err	Err	Do	Ob
Si											Si	Do	Err	Err	Do	Ob
Do												Do	Err	Err	Do	Ob
Da													St	Err	St	Ob
Ch														St	St	Ob
St															St	Ob
Ob																Ob

AdditionOperatorExpression ::= *Expression* `+` *Expression*

11.13.4 Subtraction Operator

The subtraction operator subtracts the second operand from the first operand. 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.

  Note 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	SB	SB	Sh	Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
SB		SB	Sh	Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
By			By	Sh	US	In	UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob
Sh				Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
US					US	In	UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob
In						In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
UI							UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob
Lo								Lo	De	De	Si	Do	Err	Err	Do	Ob
UL									UL	De	Si	Do	Err	Err	Do	Ob
De										De	Si	Do	Err	Err	Do	Ob
Si											Si	Do	Err	Err	Do	Ob
Do												Do	Err	Err	Do	Ob
Da													Err	Err	Err	Err
Ch														Err	Err	Err
St															Do	Ob
Ob																Ob

SubtractionOperatorExpression ::= *Expression* `-` *Expression*

11.13.5 Multiplication Operator

The multiplication operator computes the product of two operands. 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	SB	SB	Sh	Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
SB		SB	Sh	Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
By			By	Sh	US	In	UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob
Sh				Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
US					US	In	UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob
In						In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
UI							UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob
Lo								Lo	De	De	Si	Do	Err	Err	Do	Ob
UL									UL	De	Si	Do	Err	Err	Do	Ob
De										De	Si	Do	Err	Err	Do	Ob
Si											Si	Do	Err	Err	Do	Ob
Do												Do	Err	Err	Do	Ob
Da													Err	Err	Err	Err
Ch														Err	Err	Err
St															Do	Ob
Ob																Ob

MultiplicationOperatorExpression ::= *Expression* `*` *Expression*

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.

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.

Annotation

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	SB	SB	Sh	Sh	In	In	Lo	Lo	Lo	Lo	Lo	Lo	Err	Err	Lo	Ob
SB		SB	Sh	Sh	In	In	Lo	Lo	Lo	Lo	Lo	Lo	Err	Err	Lo	Ob
By			By	Sh	US	In	UI	Lo	UL	Lo	Lo	Lo	Err	Err	Lo	Ob
Sh				Sh	In	In	Lo	Lo	Lo	Lo	Lo	Lo	Err	Err	Lo	Ob
US					US	In	UI	Lo	UL	Lo	Lo	Lo	Err	Err	Lo	Ob
In						In	Lo	Lo	Lo	Lo	Lo	Lo	Err	Err	Lo	Ob
UI							UI	Lo	UL	Lo	Lo	Lo	Err	Err	Lo	Ob
Lo								Lo	Lo	Lo	Lo	Lo	Err	Err	Lo	Ob
UL									UL	Lo	Lo	Lo	Err	Err	Lo	Ob
De										Lo	Lo	Lo	Err	Err	Lo	Ob
Si											Lo	Lo	Err	Err	Lo	Ob
Do												Lo	Err	Err	Lo	Ob
Da													Err	Err	Err	Err
Ch														Err	Err	Err
St															Lo	Ob
Ob																Ob

DivisionOperatorExpression ::=
    FPDivisionOperatorExpression |
    IntegerDivisionOperatorExpression

FPDivisionOperatorExpression ::= *Expression* `/` *Expression*

IntegerDivisionOperatorExpression ::= *Expression* `\` *Expression*

11.13.7 Mod Operator

The `Mod` (modulo) operator computes the remainder of the division between two operands. 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	SB	SB	Sh	Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
SB		SB	Sh	Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
By			By	Sh	US	In	UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob
Sh				Sh	In	In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
US					US	In	UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob
In						In	Lo	Lo	De	De	Si	Do	Err	Err	Do	Ob
UI							UI	Lo	UL	De	Si	Do	Err	Err	Do	Ob
Lo								Lo	De	De	Si	Do	Err	Err	Do	Ob
UL									UL	De	Si	Do	Err	Err	Do	Ob
De										De	Si	Do	Err	Err	Do	Ob
Si											Si	Do	Err	Err	Do	Ob
Do												Do	Err	Err	Do	Ob
Da													Err	Err	Err	Err
Ch														Err	Err	Err
St															Do	Ob
Ob																Ob

ModuloOperatorExpression ::= *Expression* `Mod` *Expression*

11.13.8 Exponentiation Operator

The exponentiation operator computes the first operand raised to the power of the second operand. 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

ExponentOperatorExpression ::= *Expression* `^` *Expression*

11.14 Relational Operators

The *relational operators* compare values to one other. The comparison operators are `=`, `<>`, `<`, `>`, `<=`, and `>=`. 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 reference 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

```
RelationalOperatorExpression ::=
    Expression = Expression |
    Expression <> Expression |
    Expression < Expression |
    Expression > Expression |
    Expression <= Expression |
    Expression >= Expression
```

11.15 Like Operator

The `Like` operator is defined for the `String` type and determines whether a string matches a given pattern. 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.

> **Note** To match the special characters left bracket (`[`), question mark (`?`), number sign (`#`), and asterisk (`*`), brackets must enclose them. The right bracket (`]`) can not 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 `""`.

Also 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 a predefined 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

LikeOperatorExpression ::= *Expression* `Like` *Expression*

11.16 Concatenation Operator

The *concatenation operator* is defined for the `String` type. To make string concatenation simpler, for the purposes of operator resolution, all conversions to `String` are considered to be widening, regardless of whether strict semantics are used. A concatenation operation results in a string that is the concatenation of the two operands in order from left to right.

In a concatenation expression where both operands are `Nothing` or one operand has a predefined conversion to `String` and the other operand is `Nothing`, `Nothing` is treated as if it were the empty string literal `""`. Also, if only one operand is typed as `System.DBNull`, that operand is treated as if it was the literal `Nothing`.

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

ConcatenationOperatorExpression ::= *Expression* `&` *Expression*

11.17 Logical Operators

The `And`, `Not`, `Or`, and `Xor` operators, which are called 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).

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

LogicalOperatorExpression ::=
  `Not` *Expression* |
  Expression `And` *Expression* |
  Expression `Or` *Expression* |
  Expression `Xor` *Expression*

11.17.1 Short-circuiting Logical Operators

The `AndAlso` and `OrElse` operators are the short-circuiting versions of the `And` and `Or` logical operators. 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`, the expression returns `False`. 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`, the expression returns `True`. 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(ByVal op As T) As Boolean
Public Shared Operator IsFalse(ByVal op As T) As Boolean
```

as well as overloading the corresponding `And` or `Or` operator:

```
Public Shared Operator And(ByVal op1 As T, ByVal op2 As T) As T
Public Shared Operator Or(ByVal op1 As T, ByVal 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.WriteLine("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
```

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

ShortCircuitLogicalOperatorExpression ::=
    Expression `AndAlso` *Expression* |
    Expression `OrElse` *Expression*

11.18 Shift Operators

The binary operators `<<` and `>>` perform bit shifting operations. 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
SB	SB	By	Sh	US	In	UI	Lo	UL	Lo	Lo	Lo	Err	Err	Lo	Ob

ShiftOperatorExpression ::=
    Expression `<<` *Expression* |
    Expression `>>` *Expression*

11.19 Boolean Operators

The Visual Basic language defines two pseudo operators, `IsTrue` and `IsFalse`. These two operators are used to determine the result of the `Boolean` expression in an `If`, `While` or `Do` statement, the `When` clause of a `Catch` statement, or the result of the `AndAlso` and `OrElse` operators if the result type of the expression does not have a widening conversion to `Boolean`.

Annotation

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, it's use in the `If` statement causes a call to the `IsTrue` operator.

```
Class MyBool
    Public Shared Widening Operator CType(ByVal b As Boolean) As MyBool
        ...
    End Operator

    Public Shared Narrowing Operator CType(ByVal b As MyBool) As Boolean
        ...
    End Operator

    Public Shared Operator IsTrue(ByVal b As MyBool) As Boolean
        ...
    End Operator

    Public Shared Operator IsFalse(ByVal 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
```

BooleanExpression ::= *Expression*

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. 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(ByVal x As Integer, ByVal 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(ByVal 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 MyClass
    ''' <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(ByVal x As Integer, ByVal 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(ByVal x As Integer, ByVal 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(ByVal x As Integer, ByVal 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(ByVal x As Integer, ByVal 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(ByVal 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 or 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(ByVal 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 Widget()
```

```
        End Sub

        Public Sub New()
        End Sub

        Public Sub New(ByVal 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(ByVal i As Integer)
        End Sub
    End Structure

    Class Widget
        Public Class NestedClass
            Public Sub M(ByVal i As Integer)
            End Sub
        End Class

        Public Shared Sub M0()
        End Sub

        Public Sub M1(ByVal c As Char, ByRef f As Float, _
            ByRef v As ValueType)
        End Sub

        Public Sub M2(ByVal x1() As Short, ByVal x2(,) As Integer, _
            ByVal x3()() As Long)
        End Sub

        Public Sub M3(ByVal x3()() As Long, ByVal x4()(,,) As Widget)
        End Sub
```

```
        Public Sub M4(ByVal Optional i As Integer = 1, _
            ByVal 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,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(ByVal i As Integer) As Integer
            Get
            End Get
            Set (Value As Integer)
            End Set
        End Property

        Public Default Property Item(ByVal s As String, _
            ByVal 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 Del
        Public Event AnotherEvent()
    End Class
End Namespace

"E:Acme.Widget.AnEvent"
"E:Acme.Widget.AnotherEvent"
```

- Operators.

```
Namespace Acme
    Class Widget
        Public Shared Operator +(ByVal x As Widget) As Widget
        End Operator

        Public Shared Operator +(ByVal x1 As Widget, ByVal x2 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(ByVal x As Widget) As _
            Integer
        End Operator

        Public Shared Widening Operator CType(ByVal 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(ByVal 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(ByVal 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(ByVal x As Integer, ByVal 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(ByVal x As Integer, ByVal 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(ByVal x As Integer, ByVal 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 ="/>
''' <seealso cref="Operator <>"/>
Public Overrides Function Equals(ByVal 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 =(ByVal p1 As Point, _
          ByVal 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 <>(ByVal p1 As Point, _
          ByVal 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>
```

13. Grammar Summary

This section summarizes the Visual Basic language grammar. For information on how to read the grammar, see Grammar Notation.

13.1 Lexical Grammar

Start ::= [*LogicalLine+*]

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

LogicalLineElement ::= *WhiteSpace* | *LineContinuation* | *Token*

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

13.1.1 Characters and Lines

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

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

LineContinuation ::= *WhiteSpace* `_` [*WhiteSpace+*] *LineTerminator*

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

Comment ::= *CommentMarker* [*Character+*]

CommentMarker ::= *SingleQuoteCharacter* | `REM`

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

13.1.2 Identifiers

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

NonEscapedIdentifier ::= < *IdentifierName* but not *Keyword* >

EscapedIdentifier ::= `[` *IdentifierName* `]`

IdentifierName ::= *IdentifierStart* [*IdentifierCharacter+*]

IdentifierStart ::=
    AlphaCharacter |
    UnderscoreCharacter IdentifierCharacter

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

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

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

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

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

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

IdentifierOrKeyword ::= *Identifier* | *Keyword*

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

IntegerTypeCharacter ::= `%`

LongTypeCharacter ::= `&`

DecimalTypeCharacter ::= `@`

SingleTypeCharacter ::= `!`

DoubleTypeCharacter ::= `#`

StringTypeCharacter ::= `$`

13.1.3 Keywords

Keyword ::= < member of keyword table in 2.3 >

13.1.4 Literals

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

BooleanLiteral ::= `True` | `False`

IntegerLiteral ::= *IntegralLiteralValue* [*IntegralTypeCharacter*]

IntegralLiteralValue ::= *IntLiteral* | *HexLiteral* | *OctalLiteral*

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

ShortCharacter ::= `S`

UnsignedShortCharacter ::= `US`

IntegerCharacter ::= `I`

UnsignedIntegerCharacter ::= `UI`

LongCharacter ::= `L`

UnsignedLongCharacter ::= `UL`

IntLiteral ::= *Digit+*

HexLiteral ::= `&` `H` *HexDigit+*

OctalLiteral ::= `&` `O` *OctalDigit+*

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

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

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

FloatingPointLiteral ::=
FloatingPointLiteralValue [*FloatingPointTypeCharacter*] |
IntLiteral *FloatingPointTypeCharacter*

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

SingleCharacter ::= `F`

DoubleCharacter ::= `R`

DecimalCharacter ::= `D`

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

Exponent ::= `E` [*Sign*] *IntLiteral*

Sign ::= + | -

StringLiteral ::=
    DoubleQuoteCharacter [*StringCharacter+*] *DoubleQuoteCharacter*

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

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

CharacterLiteral ::= *DoubleQuoteCharacter* *StringCharacter* *DoubleQuoteCharacter* C

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

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

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

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

MonthValue ::= *IntLiteral*

DayValue ::= *IntLiteral*

YearValue ::= *IntLiteral*

HourValue ::= *IntLiteral*

MinuteValue ::= *IntLiteral*

SecondValue ::= *IntLiteral*

AMPM ::= AM | PM

Nothing ::= Nothing

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

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

13.2 Preprocessing Directives

13.2.1 Conditional Compilation

Start ::= [*CCStatement+*]

CCStatement ::=
    CCConstantDeclaration |

```
    CCIfGroup |
    LogicalLine

CCExpression ::=
    LiteralExpression |
    CCParenthesizedExpression |
    SimpleNameExpression |
    CCCastExpression |
    CCOperatorExpression

CCParenthesizedExpression ::= ( CCExpression )

CCCastExpression ::= CastTarget ( CCExpression )

CCOperatorExpression ::=
    CCUnaryOperator CCExpression
    CCExpression CCBinaryOperator CCExpression

CCUnaryOperator ::= + | - | Not

CCBinaryOperator ::= + | - | * | / | \ | Mod | ^ | = | <> | < | > |

    <= | >= | & | And | Or | Xor | AndAlso | OrElse | << | >>

CCConstantDeclaration ::= # Const Identifier = CCExpression LineTerminator

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

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

CCElseGroup ::=
    # Else LineTerminator
    [ CCStatement+ ]
```

13.2.2 External Source Directives

```
Start ::= [ ExternalSourceStatement+ ]

ExternalSourceStatement ::= ExternalSourceGroup | LogicalLine

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

13.2.3 Region Directives

```
Start ::= [ RegionStatement+ ]

RegionStatement ::= RegionGroup | LogicalLine

RegionGroup ::=
    # Region StringLiteral LineTerminator
```

```
    [ LogicalLine+ ]
    # End Region LineTerminator
```

13.2.4 External Checksum Directives

```
Start ::= [ ExternalChecksumStatement+ ]

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

13.3 Syntactic Grammar

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

QualifiedIdentifier ::=
    Identifier |
    Global . IdentifierOrKeyword |
    QualifiedIdentifier . IdentifierOrKeyword

TypeParameterList ::=
    ( Of TypeParameters )

TypeParameters ::=
    TypeParameter |
    TypeParameters , TypeParameter

TypeParameter ::=
    Identifier [ TypeParameterConstraints ]

TypeParameterConstraints ::=
    As Constraint |
    As { ConstraintList }

ConstraintList ::=
    ConstraintList , Constraint |
    Constraint

Constraint ::= TypeName | New
```

13.3.1 Attributes

```
Attributes ::=
    AttributeBlock |
    Attributes AttributeBlock

AttributeBlock ::= < AttributeList >

AttributeList ::=
    Attribute |
    AttributeList , Attribute

Attribute ::=
    [ AttributeModifier : ] SimpleTypeName [ ( [ AttributeArguments ] ) ]

AttributeModifier ::= Assembly | Module

AttributeArguments ::=
    AttributePositionalArgumentList |
```

AttributePositionalArgumentList `,` *VariablePropertyInitializerList* |
VariablePropertyInitializerList

AttributePositionalArgumentList ::=
AttributeArgumentExpression |
AttributePositionalArgumentList `,` *AttributeArgumentExpression*

VariablePropertyInitializerList ::=
VariablePropertyInitializer |
VariablePropertyInitializerList `,` *VariablePropertyInitializer*

VariablePropertyInitializer ::=
IdentifierOrKeyword `:=` *AttributeArgumentExpression*

AttributeArgumentExpression ::=
ConstantExpression |
GetTypeExpression |
ArrayCreationExpression

13.3.2 Source Files and Namespaces

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

StatementTerminator ::= *LineTerminator* | `:`

AttributesStatement ::= *Attributes* *StatementTerminator*

OptionStatement ::=
OptionExplicitStatement |
OptionStrictStatement |
OptionCompareStatement

OptionExplicitStatement ::= `Option Explicit` [*OnOff*] *StatementTerminator*

OnOff ::= `On` | `Off`

OptionStrictStatement ::= `Option Strict` [*OnOff*] *StatementTerminator*

OptionCompareStatement ::= `Option Compare` *CompareOption* *StatementTerminator*

CompareOption ::= `Binary` | `Text`

ImportsStatement ::= `Imports` *ImportsClauses* *StatementTerminator*

ImportsClauses ::=
ImportsClause |
ImportsClauses `,` *ImportsClause*

ImportsClause ::= *ImportsAliasClause* | *ImportsNamespaceClause*

ImportsAliasClause ::=
Identifier `=` *QualifiedIdentifier* |
Identifier `=` *ConstructedTypeName*

```
ImportsNamespaceClause ::=
    QualifiedIdentifier |
    ConstructedTypeName

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

NamespaceMemberDeclaration ::=
    NamespaceDeclaration |
    TypeDeclaration

TypeDeclaration ::=
    ModuleDeclaration |
    NonModuleDeclaration

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

13.3.3 Types

```
TypeName ::=
    ArrayTypeName |
    NonArrayTypeName

NonArrayTypeName ::=
    SimpleTypeName |
    ConstructedTypeName

SimpleTypeName ::=
    QualifiedIdentifier |
    BuiltInTypeName

BuiltInTypeName ::= Object | PrimitiveTypeName

TypeModifier ::= AccessModifier | Shadows

TypeImplementsClause ::= Implements Implements StatementTerminator

Implements ::=
    NonArrayTypeName |
    Implements , NonArrayTypeName

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

NumericTypeName ::= IntegralTypeName | FloatingPointTypeName | Decimal

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

FloatingPointTypeName ::= Single | Double

EnumDeclaration ::=
    [ Attributes ] [ TypeModifier+ ] Enum Identifier [ As QualifiedName ] StatementTerminator
```

```
    EnumMemberDeclaration+
    End Enum StatementTerminator

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

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

ClassModifier ::= TypeModifier | MustInherit | NotInheritable | Partial

ClassBase ::= Inherits NonArrayTypeName StatementTerminator

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

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

StructureModifier ::= TypeModifier | Partial

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

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

ModuleMemberDeclaration ::=
    NonModuleDeclaration |
    VariableMemberDeclaration |
    ConstantMemberDeclaration |
    EventMemberDeclaration |
    MethodMemberDeclaration |
```

PropertyMemberDeclaration |
ConstructorMemberDeclaration

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

InterfaceBase ::= `Inherits` *InterfaceBases* *StatementTerminator*

InterfaceBases ::=
NonArrayTypeName |
InterfaceBases `,` *NonArrayTypeName*

InterfaceMemberDeclaration ::=
NonModuleDeclaration |
InterfaceEventMemberDeclaration |
InterfaceMethodMemberDeclaration |
InterfacePropertyMemberDeclaration

ArrayTypeName ::= *NonArrayTypeName* *ArrayTypeModifiers*

ArrayTypeModifiers ::= *ArrayTypeModifier+*

ArrayTypeModifier ::= `(` [*RankList*] `)`

RankList ::=
`,` |
RankList `,`

ArrayNameModifier ::=
ArrayTypeModifiers |
ArraySizeInitializationModifier

DelegateDeclaration ::=
[*Attributes*] [*TypeModifier+*] `Delegate` *MethodSignature* *StatementTerminator*

MethodSignature ::= *SubSignature* | *FunctionSignature*

ConstructedTypeName ::=
QualifiedIdentifier `( Of` *TypeArgumentList* `)`

TypeArgumentList ::=
TypeName |
TypeArgumentList `,` *TypeName*

13.3.4 Type Members

ImplementsClause ::= [`Implements` *ImplementsList*]

ImplementsList ::=
InterfaceMemberSpecifier |
ImplementsList `,` *InterfaceMemberSpecifier*

InterfaceMemberSpecifier ::= *NonArrayTypeName* `.` *IdentifierOrKeyword*

MethodMemberDeclaration ::= *MethodDeclaration* | *ExternalMethodDeclaration*

InterfaceMethodMemberDeclaration ::= *InterfaceMethodDeclaration*

```
MethodDeclaration ::=
    SubDeclaration |
    MustOverrideSubDeclaration |
    FunctionDeclaration |
    MustOverrideFunctionDeclaration

InterfaceMethodDeclaration ::=
    InterfaceSubDeclaration |
    InterfaceFunctionDeclaration

SubSignature ::= Identifier [ TypeParameterList ] [ ( [ ParameterList ] ) ]

FunctionSignature ::= SubSignature [ As [ Attributes ] TypeName ]

SubDeclaration ::=
    [ Attributes ] [ ProcedureModifier+ ] Sub SubSignature [ HandlesOrImplements ] LineTerminator
    Block
    End Sub StatementTerminator

MustOverrideSubDeclaration ::=
    [ Attributes ] [ MustOverrideProcedureModifier+ ] Sub SubSignature [ HandlesOrImplements ]
        StatementTerminator

InterfaceSubDeclaration ::=
    [ Attributes ] [ InterfaceProcedureModifier+ ] Sub SubSignature StatementTerminator

FunctionDeclaration ::=
    [ Attributes ] [ ProcedureModifier+ ] Function FunctionSignature [ HandlesOrImplements ]
        LineTerminator
    Block
    End Function StatementTerminator

MustOverrideFunctionDeclaration ::=
    [ Attributes ] [ MustOverrideProcedureModifier+ ] Function FunctionSignature
        [ HandlesOrImplements ] StatementTerminator

InterfaceFunctionDeclaration ::=
    [ Attributes ] [ InterfaceProcedureModifier+ ] Function FunctionSignature StatementTerminator

ProcedureModifier ::=
    AccessModifier |
    Shadows |
    Shared |
    Overridable |
    NotOverridable |
    Overrides |
    Overloads

MustOverrideProcedureModifier ::= ProcedureModifier | MustOverride

InterfaceProcedureModifier ::= Shadows | Overloads

HandlesOrImplements ::= HandlesClause | ImplementsClause

ExternalMethodDeclaration ::=
    ExternalSubDeclaration |
    ExternalFunctionDeclaration
```

```
ExternalSubDeclaration ::=
    [ Attributes ] [ ExternalMethodModifier+ ] Declare [ CharsetModifier ] Sub Identifier
        LibraryClause [ AliasClause ] [ ( [ ParameterList ] ) ] StatementTerminator

ExternalFunctionDeclaration ::=
    [ Attributes ] [ ExternalMethodModifier+ ] Declare [ CharsetModifier ] Function Identifier
        LibraryClause [ AliasClause ] [ ( [ ParameterList ] ) ] [ As [ Attributes ] TypeName ]
        StatementTerminator

ExternalMethodModifier ::= AccessModifier | Shadows | Overloads

CharsetModifier ::= Ansi | Unicode | Auto

LibraryClause ::= Lib StringLiteral

AliasClause ::= Alias StringLiteral

ParameterList ::=
    Parameter |
    ParameterList , Parameter

Parameter ::=
    [ Attributes ] ParameterModifier+ ParameterIdentifier [ As TypeName ] [ = ConstantExpression ]

ParameterModifier ::= ByVal | ByRef | Optional | ParamArray

ParameterIdentifier ::= Identifier [ ArrayNameModifier ]

HandlesClause ::= [ Handles EventHandlesList ]

EventHandlesList ::=
    EventMemberSpecifier |
    EventHandlesList , EventMemberSpecifier

EventMemberSpecifier ::=
    QualifiedIdentifier . IdentifierOrKeyword |
    MyBase . IdentifierOrKeyword |
    Me . IdentifierOrKeyword

ConstructorMemberDeclaration ::=
    [ Attributes ] [ ConstructorModifier+ ] Sub New [ ( [ ParameterList ] ) ] LineTerminator
    [ Block ]
    End Sub StatementTerminator

ConstructorModifier ::= AccessModifier | Shared

EventMemberDeclaration ::=
    RegularEventMemberDeclaration |
    CustomEventMemberDeclaration

RegularEventMemberDeclaration ::=
    [ Attributes ] [ EventModifiers+ ] Event Identifier ParametersOrType [ ImplementsClause ]
        StatementTerminator

InterfaceEventMemberDeclaration ::=
    [ Attributes ] [ InterfaceEventModifiers+ ] Event Identifier ParametersOrType StatementTerminator

ParametersOrType ::=
    [ ( [ ParameterList ] ) ] |
    As NonArrayTypeName
```

```
EventModifiers ::= AccessModifier | Shadows | Shared

InterfaceEventModifiers ::= Shadows

CustomEventMemberDeclaration ::=
    [ Attributes ] [ EventModifiers+ ] Custom Event Identifier As TypeName [ ImplementsClause ]
        StatementTerminator
        EventAccessorDeclaration+
    End Event StatementTerminator

EventAccessorDeclaration ::=
    AddHandlerDeclaration |
    RemoveHandlerDeclaration |
    RaiseEventDeclaration

AddHandlerDeclaration ::=
    [ Attributes ] AddHandler ( ParameterList ) LineTerminator
    [ Block ]
    End AddHandler StatementTerminator

RemoveHandlerDeclaration ::=
    [ Attributes ] RemoveHandler ( ParameterList ) LineTerminator
    [ Block ]
    End RemoveHandler StatementTerminator

RaiseEventDeclaration ::=
    [ Attributes ] RaiseEvent ( ParameterList ) LineTerminator
    [ Block ]
    End RaiseEvent StatementTerminator

ConstantMemberDeclaration ::=
    [ Attributes ] [ ConstantModifier+ ] Const ConstantDeclarators StatementTerminator

ConstantModifier ::= AccessModifier | Shadows

ConstantDeclarators ::=
    ConstantDeclarator |
    ConstantDeclarators , ConstantDeclarator

ConstantDeclarator ::= Identifier [ As TypeName ] = ConstantExpression StatementTerminator

VariableMemberDeclaration ::=
    [ Attributes ] VariableModifier+ VariableDeclarators StatementTerminator

VariableModifier ::=
    AccessModifier |
    Shadows |
    Shared |
    ReadOnly |
    WithEvents |
    Dim

VariableDeclarators ::=
    VariableDeclarator |
    VariableDeclarators , VariableDeclarator
```

VariableDeclarator ::=
    VariableIdentifiers [`As` [`New`] *TypeName* [`(` *ArgumentList* `)`]] |
    VariableIdentifier [`As` *TypeName*] [`=` *VariableInitializer*]

VariableIdentifiers ::=
    VariableIdentifier |
    VariableIdentifiers `,` *VariableIdentifier*

VariableIdentifier ::= *Identifier* [*ArrayNameModifier*]

VariableInitializer ::= *RegularInitializer* | *ArrayElementInitializer*

RegularInitializer ::= *Expression*

ArraySizeInitializationModifier ::=
    `(` *BoundList* `)` [*ArrayTypeModifiers*]

BoundList::=
    Expression |
    `0 To` Expression |
    UpperBoundList `,` *Expression*

ArrayElementInitializer ::= `{` [*VariableInitializerList*] `}`

VariableInitializerList ::=
    VariableInitializer |
    VariableInitializerList `,` *VariableInitializer*

VariableInitializer ::= *Expression* | *ArrayElementInitializer*

PropertyMemberDeclaration ::=
    RegularPropertyMemberDeclaration |
    MustOverridePropertyMemberDeclaration

RegularPropertyMemberDeclaration ::=
    [*Attributes*] [*PropertyModifier+*] `Property` *FunctionSignature* [*ImplementsClause*]
        LineTerminator
    PropertyAccessorDeclaration+
    `End Property` *StatementTerminator*

MustOverridePropertyMemberDeclaration ::=
    [*Attributes*] [*MustOverridePropertyModifier+*] `Property` *FunctionSignature* [*ImplementsClause*]
        StatementTerminator

InterfacePropertyMemberDeclaration ::=
    [*Attributes*] [*InterfacePropertyModifier+*] `Property` *FunctionSignature* *StatementTerminator*

PropertyModifier ::= *ProcedureModifier* | `Default` | `ReadOnly` | `WriteOnly`

MustOverridePropertyModifier ::= *PropertyModifier* | `MustOverride`

InterfacePropertyModifier ::=
    `Shadows` |
    `Overloads` |
    `Default` |
    `ReadOnly` |
    `WriteOnly`

PropertyAccessorDeclaration ::= *PropertyGetDeclaration* | *PropertySetDeclaration*

```
PropertyGetDeclaration ::=
    [ Attributes ] [ AccessModifier ] Get LineTerminator
    [ Block ]
    End Get StatementTerminator

PropertySetDeclaration ::=
    [ Attributes ] [ AccessModifier ] Set [ ( ParameterList ) ] LineTerminator
    [ Block ]
    End Set StatementTerminator

OperatorDeclaration ::=
    UnaryOperatorDeclaration |
    BinaryOperatorDeclaration |
    ConversionOperatorDeclaration

OperatorModifier ::= Public | Shared | Overloads | Shadows

Operand ::= [ ByVal ] Identifier [ As TypeName ]

UnaryOperatorDeclaration ::=
    [ Attributes ] [ OperatorModifier+ ] Operator OverloadableUnaryOperator ( Operand )
        [ As [ Attributes ] TypeName ] LineTerminator
    [ Block ]
    End Operator StatementTerminator

OverloadableUnaryOperator ::= + | - | Not | IsTrue | IsFalse

BinaryOperatorDeclaration ::=
    [ Attributes ] [ OperatorModifier+ ] Operator OverloadableBinaryOperator
        ( Operand , Operand ) [ As [ Attributes ] TypeName ] LineTerminator
    [ Block ]
    End Operator StatementTerminator

OverloadableBinaryOperator ::=
    + | - | * | / | \ | & | Like | Mod | And | Or | Xor |
    ^ | << | >> | = | <> | > | < | >= | <=

ConversionOperatorDeclaration ::=
    [ Attributes ] [ ConversionOperatorModifier+ ] Operator CType ( Operand )
        [ As [ Attributes ] TypeName ] LineTerminator
    [ Block ]
    End Operator StatementTerminator

ConversionOperatorModifier ::= Widening | Narrowing | ConversionModifier
```

13.3.5 Statements

```
Statement ::=
    LabelDeclarationStatement |
    LocalDeclarationStatement |
    WithStatement |
    SyncLockStatement |
    EventStatement |
    AssignmentStatement |
    InvocationStatement |
    ConditionalStatement |
    LoopStatement |
```

ErrorHandlingStatement |
BranchStatement |
ArrayHandlingStatement |
UsingStatement

Block ::= [*Statements+*]

LabelDeclarationStatement ::= *LabelName* `:`

LabelName ::= *Identifier* | *IntLiteral*

Statements ::=
[*Statement*] |
Statements `:` [*Statement*]

LocalDeclarationStatement ::= *LocalModifier VariableDeclarators StatementTerminator*

LocalModifier ::= `Static` | `Dim` | `Const`

WithStatement ::=
`With` *Expression StatementTerminator*
[*Block*]
`End With` *StatementTerminator*

SyncLockStatement ::=
`SyncLock` *Expression StatementTerminator*
[*Block*]
`End SyncLock` *StatementTerminator*

EventStatement ::=
RaiseEventStatement |
AddHandlerStatement |
RemoveHandlerStatement

RaiseEventStatement ::= `RaiseEvent` *IdentifierOrKeyword* [`(` [*ArgumentList*] `)`]
StatementTerminator

AddHandlerStatement ::= `AddHandler` *Expression* `,` *Expression StatementTerminator*

RemoveHandlerStatement ::= `RemoveHandler` *Expression* `,` *Expression StatementTerminator*

AssignmentStatement ::=
RegularAssignmentStatement |
CompoundAssignmentStatement |
MidAssignmentStatement

RegularAssignmentStatement ::= *Expression* `=` *Expression StatementTerminator*

CompoundAssignmentStatement ::= *Expression CompoundBinaryOperator Expression StatementTerminator*

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

MidAssignmentStatement ::=
`Mid` [`$`] `(` *Expression* `,` *Expression* [`,` *Expression*] `)` `=` *Expression StatementTerminator*

InvocationStatement ::= [`Call`] *InvocationExpression StatementTerminator*

ConditionalStatement ::= *IfStatement* | *SelectStatement*

IfStatement ::= *BlockIfStatement* | *LineIfThenStatement*

```
BlockIfStatement ::=
    If BooleanExpression [ Then ] StatementTerminator
    [ Block ]
    [ ElseIfStatement+ ]
    [ ElseStatement ]
    End If StatementTerminator

ElseIfStatement ::=
    ElseIf BooleanExpression [ Then ] StatementTerminator
    [ Block ]

ElseStatement ::=
    Else StatementTerminator
    [ Block ]

LineIfThenStatement ::=
    If BooleanExpression Then Statements [ Else Statements ] StatementTerminator

SelectStatement ::=
    Select [ Case ] Expression StatementTerminator
    [ CaseStatement+ ]
    [ CaseElseStatement ]
    End Select StatementTerminator

CaseStatement ::=
    Case CaseClauses StatementTerminator
    [ Block ]

CaseClauses ::=
    CaseClause |
    CaseClauses , CaseClause

CaseClause ::=
    [ Is ] ComparisonOperator Expression |
    Expression [ To Expression ]

ComparisonOperator ::= = | <> | < | > | => | =<

CaseElseStatement ::=
    Case Else StatementTerminator
    [ Block ]

LoopStatement ::=
    WhileStatement |
    DoLoopStatement |
    ForStatement |
    ForEachStatement

WhileStatement ::=
    While BooleanExpression StatementTerminator
    [ Block ]
    End While StatementTerminator

DoLoopStatement ::= DoTopLoopStatement | DoBottomLoopStatement

DoTopLoopStatement ::=
    Do [ WhileOrUntil BooleanExpression ] StatementTerminator
```

```
    [ Block ]
    Loop StatementTerminator

DoBottomLoopStatement ::=
    Do StatementTerminator
    [ Block ]
    Loop WhileOrUntil BooleanExpression StatementTerminator

WhileOrUntil ::= While | Until

ForStatement ::=
    For LoopControlVariable = Expression To Expression [ Step Expression ] StatementTerminator
    [ Block ]
    Next [ NextExpressionList ] StatementTerminator

LoopControlVariable ::=
    Identifier [ ArrayNameModifier ] As TypeName |
    Expression

NextExpressionList ::=
    Expression |
    NextExpressionList , Expression

ForEachStatement ::=
    For Each LoopControlVariable In Expression StatementTerminator
    [ Block ]
    Next [Expression ] StatementTerminator

ErrorHandlingStatement ::=
    StructuredErrorStatement |
    UnstructuredErrorStatement

StructuredErrorStatement ::=
    ThrowStatement |
    TryStatement

TryStatement ::=
    Try StatementTerminator
    [ Block ]
    [ CatchStatement+ ]
    [ FinallyStatement ]
    End Try StatementTerminator

FinallyStatement ::=
    Finally StatementTerminator
    [ Block ]

CatchStatement ::=
    Catch [ Identifier As NonArrayTypeName ] [ When BooleanExpression ] StatementTerminator
    [ Block ]

ThrowStatement ::= Throw [ Expression ] StatementTerminator

UnstructuredErrorStatement ::=
    ErrorStatement |
    OnErrorStatement |
    ResumeStatement
```

ErrorStatement ::= `Error` *Expression* *StatementTerminator*

OnErrorStatement ::= `On Error` *ErrorClause* *StatementTerminator*

ErrorClause ::=
    `GoTo - 1` |
    `GoTo 0` |
    GotoStatement |
    `Resume Next`

ResumeStatement ::= `Resume` [*ResumeClause*] *StatementTerminator*

ResumeClause ::= `Next` | *LabelName*

BranchStatement ::=
    GotoStatement |
    ExitStatement |
    ContinueStatement |
    StopStatement |
    EndStatement |
    ReturnStatement

GotoStatement ::= `GoTo` *LabelName* *StatementTerminator*

ExitStatement ::= `Exit` *ExitKind* *StatementTerminator*

ExitKind ::= `Do` | `For` | `While` | `Select` | `Sub` | `Function` | `Property` | `Try`

ContinueStatement ::= `Continue` *ContinueKind* *StatementTerminator*

ContinueKind ::= `Do` | `For` | `While`

StopStatement ::= `Stop` *StatementTerminator*

EndStatement ::= `End` *StatementTerminator*

ReturnStatement ::= `Return` [*Expression*]

ArrayHandlingStatement ::=
    RedimStatement |
    EraseStatement

RedimStatement ::= `ReDim` [`Preserve`] *RedimClauses* *StatementTerminator*

RedimClauses ::=
    RedimClause |
    RedimClauses `,` *RedimClause*

RedimClause ::= *Expression* *ArraySizeInitializationModifier*

EraseStatement ::= `Erase` *EraseExpressions* *StatementTerminator*

EraseExpressions ::=
    Expression |
    EraseExpressions `,` *Expression*

UsingStatement ::=
    `Using` *UsingResources* *StatementTerminator*
        [*Block*]
    `End Using` *StatementTerminator*

UsingResources ::= *VariableDeclarators* | *Expression*

13.3.6 Expressions

Expression ::=
  SimpleExpression |
  TypeExpression |
  MemberAccessExpression |
  DictionaryAccessExpression |
  IndexExpression |
  NewExpression |
  CastExpression |
  OperatorExpression

ConstantExpression ::= *Expression*

SimpleExpression ::=
  LiteralExpression |
  ParenthesizedExpression |
  InstanceExpression |
  SimpleNameExpression |
  AddressOfExpression

LiteralExpression ::= *Literal*

ParenthesizedExpression ::= `(` *Expression* `)`

InstanceExpression ::= `Me`

SimpleNameExpression ::= *Identifier* [`(` `Of` *TypeArgumentList* `)`]

AddressOfExpression ::= `AddressOf` *Expression*

TypeExpression ::=
  GetTypeExpression |
  TypeOfIsExpression |
  IsExpression

GetTypeExpression ::= `GetType` `(` *GetTypeTypeName* `)`

GetTypeTypeName ::=
  TypeName |
  QualifiedIdentifier `(` `Of` [*TypeArityList*] `)`

TypeArityList ::=
  `,` |
  TypeParameterList `,`

TypeOfIsExpression ::= `TypeOf` *Expression* `Is` *TypeName*

IsExpression ::=
  Expression `Is` *Expression* |
  Expression `IsNot` *Expression*

MemberAccessExpression ::=
  [[*MemberAccessBase*] `.`] *IdentifierOrKeyword*

MemberAccessBase ::=
  Expression |
  BuiltInTypeName |
  `Global` |

```
    MyClass |
    MyBase
```

DictionaryAccessExpression ::= [*Expression*] `!` *IdentifierOrKeyword*

InvocationExpression ::= *Expression* [`(` [*ArgumentList*] `)`]

ArgumentList ::=
    PositionalArgumentList `,` *NamedArgumentList* |
    PositionalArgumentList |
    NamedArgumentList

PositionalArgumentList ::=
    Expression |
    PositionalArgumentList `,` [*Expression*]

NamedArgumentList ::=
    IdentifierOrKeyword `:=` *Expression* |
    NamedArgumentList `,` *IdentifierOrKeyword* `:=` *Expression*

IndexExpression ::= *Expression* `(` [*ArgumentList*] `)`

NewExpression ::=
    ObjectCreationExpression |
    ArrayCreationExpression |
    DelegateCreationExpression

ObjectCreationExpression ::=
    `New` *NonArrayTypeName* [`(` [*ArgumentList*] `)`]

ArrayCreationExpression ::=
    `New` *NonArrayTypeName* *ArraySizeInitializationModifier* *ArrayElementInitializer*

DelegateCreationExpression ::= `New` *NonArrayTypeName* `(` *Expression* `)`

CastExpression ::=
    `DirectCast` `(` *Expression* `,` *TypeName* `)` |
    `TryCast` `(` *Expression* `,` *TypeName* `)` |
    `CType` `(` *Expression* `,` *TypeName* `)` |
    CastTarget `(` *Expression* `)`

CastTarget ::=
    `CBool` | `CByte` | `CChar` | `CDate` | `CDec` | `CDbl` | `CInt` | `CLng` | `CObj` | `CSByte` | `CShort` |
    `CSng` | `CStr` | `CUInt` | `CULng` | `CUShort`

OperatorExpression ::=
    ArithmeticOperatorExpression |
    RelationalOperatorExpression |
    LikeOperatorExpression |
    ConcatenationOperatorExpression |
    ShortCircuitLogicalOperatorExpression |
    LogicalOperatorExpression |
    ShiftOperatorExpression

ArithmeticOperatorExpression ::=
    UnaryPlusExpression |
    UnaryMinusExpression |
    AdditionOperatorExpression |

```
    SubtractionOperatorExpression  |
    MultiplicationOperatorExpression  |
    DivisionOperatorExpression  |
    ModuloOperatorExpression  |
    ExponentOperatorExpression

UnaryPlusExpression  ::=  +  Expression

UnaryMinusExpression  ::=  -  Expression

AdditionOperatorExpression  ::=  Expression  +  Expression

SubtractionOperatorExpression  ::=  Expression  -  Expression

MultiplicationOperatorExpression  ::=  Expression  *  Expression

DivisionOperatorExpression  ::=
    FPDivisionOperatorExpression  |
    IntegerDivisionOperatorExpression

FPDivisionOperatorExpression  ::=  Expression  /  Expression

IntegerDivisionOperatorExpression  ::=  Expression  \  Expression

ModuloOperatorExpression  ::=  Expression  Mod  Expression

ExponentOperatorExpression  ::=  Expression  ^  Expression

RelationalOperatorExpression  ::=
    Expression  =  Expression  |
    Expression  <>  Expression  |
    Expression  <  Expression  |
    Expression  >  Expression  |
    Expression  <=  Expression  |
    Expression  >=  Expression

LikeOperatorExpression  ::=  Expression  Like  Expression

ConcatenationOperatorExpression  ::=  Expression  &  Expression

LogicalOperatorExpression  ::=
    Not  Expression  |
    Expression  And  Expression  |
    Expression  Or  Expression  |
    Expression  Xor  Expression

ShortCircuitLogicalOperatorExpression  ::=
    Expression  AndAlso  Expression  |
    Expression  OrElse  Expression

ShiftOperatorExpression  ::=
    Expression  <<  Expression  |
    Expression  >>  Expression

BooleanExpression  ::=  Expression
```

14. Change List

The following is a list of changes made to the specification between the previous major version and this version. The sections affected are listed after each change.

14.1 Major changes

- Multiple attribute blocks are now allowed before a declaration (i.e. `<a> <b>` instead of just `<a,b>`). [5.2, 6]
- Added the `Continue` statement. [2.3, 10.11]
- Added the `Using` statement. [2.3, 10, 10.13]
- Added the `IsNot` operator. [2.3, 11.5.3]
- Added the `Global` qualifier which allows binding in the global namespace. [2.3, 4.7, 11.6]
- Added XML Documentation comments. [12]
- Derived classes are allowed to re-implement interfaces implemented by their base class. [4.4, 4.4.1]
- Added the `TryCast` operator. [2.3, 11.11]
- Attributes can have arguments typed as `Object` or one-dimensional arrays. [5.1, 5.2.2]
- Added a section on language compatibility [1.2, 1.2.1, 1.2.2, 1.2.3]
- Added operator overloading. [2.3, 4.1.1, 7.5.2, 7.6.1, 9.8, 9.8.1, 9.8.2, 9.8.3, 10.8.2, 10.9.2, 11.17.1, 11.11, 8.11, 8.11.1, 8.11.2, 11.12.3]
- Added pseudo operators `IsTrue` and `IsFalse`. [11.19, 10.8.1, 10.9.1, 10.10.1.2]
- Added unsigned integer types. [2.2.1, 2.3, 2.4.2, 7.3, 11.11, 7.4, 8.2, 8.3, 8.7, 8.8, 8.9, 10.9.2, 11.2, 11.12.3, 11.13.1, 11.13.2, 11.13.3, 11.13.4, 11.13.5, 11.13.6, 11.13.7, 11.13.8, 11.14, 11.15, 11.16, 11.17, 11.17.1, 11.18, 11.8.1]
- Added custom event declarations. [9.4.1]
- Property accessors can specify a more restrictive access level than their containing property. [9.7, 9.7.1, 9.7.2, 9.7.3]
- Added partial types. [2.3, 7.5, 7.6, 7.11]
- Added default instances. [11.6.2, 11.6.2.1, 11.6.2.2]
- Added generic types and methods. [2.3, 2.4.7, 4.1.1, 4.4.1, 4.5.1, 4.6, 4.7.1, 4.7.2, 4.9, 4.9.1, 4.9.2, 5.1, 5.2.2, 6.1, 6.3.1, 6.3.2, 7, 7.2, 7.5, 7.5.1, 7.6, 7.8, 7.8.1, 7.9, 7.11, 7.12, 7.12.1, 8.6, 8.10, 9.1, 9.2.1, 9.2.2, 9.3.2, 9.4, 9.6, 9.8.1, 9.8.2, 9.8.3, 10.2, 10.9.3, 10.10.1.2, 11.1, 11.4.4, 11.4.5, 11.5.1, 11.5.2, 11.5.3, 11.6, 11.6.2, 11.8, 11.8.2, 11.8.5, 11.10.1, 11.10.2, 11.10.3, 12.2.16, 12.3]

14.2 Minor changes

- Binary string comparisons are always used for conditional compilation. Otherwise, string comparisons would not work because text string comparisons depend on the run-time culture. [3.1.2]

- Types cannot inherit from a type that is directly or indirectly contained within it. Also clarified the examples. [4.3]
- Changed the grammar and spec so that enumerated types can use the `System` equivalents of the fundamental types as an underlying type. [7.4]
- We now allow a conversion between an array of an enumerated type and an array of the underlying type of the enumeration. [8.5, 8.8. 8.9]
- When overriding a method, you can override it with a `MustOverride` method, causing it to become abstract. [4.5.1]
- A type member can handle an event in its own class using `Handles`. [9.2.6]
- Methods and properties that are declared `Overrides` now assume `Overloads`, which is more logical than assuming `Shadows`. [4.3.3]
- Fields and local variables are allowed to initialize multiple variables as once using the `As New` syntax. [9.6]
- Removed the restriction that an inner `Catch` block cannot branch into an outer `Try` block. [10.10.1.2]
- Classes cannot inherit from `System.MulticastDelegate`. [7.5.1]
- Shared variables in structures can have initializers. [9.6.3]
- Added a rule that numeric types are preferred over enumerated types when doing overload resolution against the literal 0. [11.8.1]
- Array size initializers can explicitly state a lower bound of zero. [9.6.3.3]
- Added an external checksum directive. [3.4]
- Array-size initializers on fields are allowed to be non-constant expressions. [9.6.3.3]
- Keywords with type characters are now treated as identifiers. [2.2]
- Constants, fields, properties, locals and parameters that have the same name as their type can be interpreted either as the member or the type for the purposes of member lookup. [11.6.1]
- Added a section talking about types restricted by the .NET Framework and moved discussion of `System.Void` to that section. [7, 7.13]
- When dealing with a project and a referenced assembly that define the same fully-qualified name, the project's type is preferred, otherwise the name is ambiguous. [4.7.2, 11.4.4]
- `On Error` statements do not extend over the call to `New` at the beginning of a constructor. [10.10.2]
- `Catch` statements can now specify `Object` as the exception type in a catch. [10.10.1.2]
- Resolving an overloaded call to a late bound call is disallowed if the containing type is an interface. [11.8.1]
- `Overloads` and `Shadows` are not allowed in a standard module. [4.3.3]
- When looking up in interfaces, a name shadowed in one path through the hierarchy is shadowed in all paths through the hierarchy. Previously, we would give an ambiguity error. [4.3.2]
- When binding through imports, types and type members are given preference over namespaces. [4.7.2, 11.6]
- Changing the default property of a type no longer requires `Shadows`. [9.7.3]
- Unreserved the contextual keywords `Assembly`, `Ansi`, `Auto`, `Preserve`, `Unicode` and `Until`. [2.3]

14.3 Clarifications/Errata

- Added a note that full-width/half-width equivalence only works on a whole-token basis (i.e. you can't mix it within a token.) [1.1]
- There are places in the language that allow regular type names but do not allow array type names. Added clarification to the grammar to call these out. [5.2, 7, 7.2, 7.5.1, 7.8.1, 7.9, 9.1, 9.2.5, 10.10.1.2, 11.10.1, 11.10.2, 11.10.3]
- The spec incorrectly states that colons can only be used as separators at the statement level. In fact, they can be used almost anywhere. Clarified the grammar and spec. [6, 6.2.1, 6.2.2, 6.2.3, 6.3, 6.4.1, 7.2, 7.4, 7.4.1, 7.5, 7.5.1, 7.6, 7.7, 7.8, 7.8.1, 9.2.1, 9.2.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.7.1, 9.7.2, 10]
- As a part of the previous bullet point, made labels a statement unto themselves and fixed up the grammar. [10, 10.1]
- Narrowed the scope of the grammar for delegate declarations to be more accurate. [7.10]
- Array covariance also includes interface implementation in addition to inheritance. [8.5]
- When implementing an interface or handling an event, identifiers after a period can match keywords. [9.1, 9.2.6]
- Split `MustOverride` declarations out from regular method and property declarations. [9.2.1, 9.7]
- The grammar was wrong for variable declarations – `Dim` is a regular modifier, and not a separate element. [9.6]
- Spelled out how attributes on `WithEvents` fields are transmitted to the underlying synthetic members. [9.6.2]
- Invoking an event won't throw an exception if there are no listeners. [10.5.1]
- Expanded the `RaiseEvent` section to cover the avoidance of race conditions on raising an event. [10.5.1]
- Clarified that `RaiseEvent` takes an identifier, not an expression. [10.5.1]
- Covered a corner case with `Mid` assignment. [10.6.3]
- Statements after a line `If` are not optional. [10.8.1]
- Expanded the `Do…Loop` grammar to make it more explicit. [10.9.1]
- Loop control variables in `For` loops can be of an array type. [10.9.2]
- The size modifier is not optional in an array creation expression. [11.10.2]
- Property return types can have attributes [9.7]
- Split out the interface versions of event, property and method declarations for grammatical clarity. [7.8.2, 9.2, 9.2.1, 9.4, 9.7]
- `MyClass` and `MyBase` cannot stand alone and are moved to the qualified expression production. [11.4.3, 11.6]
- `TypeOf…Is` is part of the primary category of operator precedence. [11.12.1]
- A `Handles` clause can have more than one identifier, but only two identifiers are legal. [9.2.6]
- Conditional compilation only supports a subset of constant expressions. [3.1]
- Corrected references to `System.Monitor` to `System.Threading.Monitor`. [10.4]

- Clarified that a compiler can put declarations into a particular namespace by default. [6.4]
- Re-throwing an exception (`Throw` with no argument) cannot occur inside of a `Finally` block. [10.10.1.3]
- A conversion from a type to itself is considered widening. [8.8]
- Tried to clarify the exact behavior of `DirectCast` a bit better than before. [11.11]
- The element type of a collection in a `For Each` statement does not have to have an implicit conversion to the loop control variable/expression: the conversion can be of any kind. [10.9.3]
- Clarified decimal division behavior. [11.13.6]
- Clarified that overloaded operators are not considered when converting Nothing to an empty string for the `&` operator. Also clarified the same behavior applies to the `Like` operator. [11.15, 11.16]
- Clarified that operations that have `Object` parameters might result in something other than `Integer`. [11.12.2]
- Added explicit operator type tables. [11.12.3, 11.13.1, 11.13.2, 11.13.3, 11.13.4, 11.13.5, 11.13.6, 11.13.7, 11.13.8, 11.14, 11.15, 11.16, 11.17, 11.17.1, 11.18]
- Tried to make the "most specific" rules more clear in intent. [11.8.1]
- Clarified that shadowing a `ParamArray` method by name and signature hides only that signature, even if the shadowing method matches the unexpanded signature of the `ParamArray` method. [4.3.3]
- Moved the rule about preferring fewer paramarray matches over more during overload resolution earlier in the process to match compiler (and desired) behavior. [11.8.1, 11.8.2]
- Array-size initializers and array-element initializers cannot be combined. [9.6.3.3]
- When specifying bounds on an array creation expression and supplying an array-element initializer, the bounds must be specified using constant expressions. [9.6.3.4]
- Added a discussion of boxing and unboxing, as well as limitations to our anti-aliasing of boxed types design. [8.6]
- Tried to clarify an obscure rule about enumerated types and `For` loops with `Object` loop control variables. [10.9.2]
- Clarified that mucking with the loop control variable during an `Object` loop doesn't change the type of the loop. [10.9.2]
- Noted that delegates and external methods can use duplicate parameter names. [9.2.5]
- Interfaces have a widening conversion to `Object`. [8.4, 8.8]
- Interfaces do not inherit from `Object`. [7.8.1]
- `Object` is reference type. It is not a type that is "neither a reference type nor a value type." [7]
- Noted the position of `System.ValueType` and `System.Enum` in the value type hierarchy. [7.1]
- Called out the primitive type conversions we allow when boxed as `Object`. [8.6]
- Expanded the explanation of a delegate's members. [7.10]
- Expanded discussion of implicit locals. [10.1.1, 10.2.1]
- Clarified how static initializers work. [10.2]
- Noted which synthetic names are ignored by name binding. [9.4, 9.4.1, 9.7.1, 9.7.2]

- Exceptions caught in `Try…Catch` blocks store their exceptions in the Err object. [10.10.1.2]
- Called out the presence of the identity conversion. [8]
- Clarified how late-bound accesses are treated in expression contexts. [11.1, 11.1.1, 11.3, 11.6, 11.8.1, 11.9]
- Corrected rules on when shared constructors execute. [9.3.2]

14.4 Miscellaneous

- Changed references to the name of the language from "Microsoft Visual Basic .NET" to "Microsoft Visual Basic." The official name of the language itself is simply "Visual Basic."
- Moved multi-token punctuators and operators such as `<=` or `>>=` into the lexical grammar for clarity. [2.5, 2.6, 5.2.2, 10.6.2, 10.8.2, 11.14]
- Removed the `NamespaceOrTypeName` production because it wasn't really needed. [4.7, 6.3.1, 6.3.2, 7]
- Removed the local variable productions because they were superfluous. [10.2]
- Consolidated all the productions that just included access modifiers and `Shadows` into a single production. [7, 7.4, 7.5, 7.6, 7.7, 7.8, 7.10]
- With the advent of a default response file for the command-line, all projects will import `System` by default, so I removed it from all examples. [Too many to count.]
- Changed the suffix for preprocessing statement productions from `Element` to `Statement` and the prefix for conditional compilation statements from `Conditional` to `CC`. [3.1, 3.1.1, 3.1.2, 3.2, 3.3]
- Corrected the example in the `Resume` statement section. [10.10.2.3]
- Added `Protected Friend` to the modifier production. [4.6]
- Added some examples to array creation expressions. [11.10.2]