Language Independence and Language-Independent Components

.NET Framework (current version)

The .NET Framework is language independent. This means that, as a developer, you can develop in one of the many languages that target the .NET Framework, such as C#, C++/CLI, Eiffel, F#, IronPython, IronRuby, PowerBuilder, Visual Basic, Visual COBOL, and Windows PowerShell. You can access the types and members of class libraries developed for the .NET Framework without having to know the language in which they were originally written and without having to follow any of the original language's conventions. If you are a component developer, your component can be accessed by any .NET Framework app regardless of its language.

Note

This first part of this article discusses creating language-independent components—that is, components that can be consumed by apps that are written in any language. You can also create a single component or app from source code written in multiple languages; see Cross-Language Interoperability in the second part of this article.

To fully interact with other objects written in any language, objects must expose to callers only those features that are common to all languages. This common set of features is defined by the Common Language Specification (CLS), which is a set of rules that apply to generated assemblies. The Common Language Specification is defined in Partition I, Clauses 7 through 11 of the ECMA-335 Standard: Common Language Infrastructure.

If your component conforms to the Common Language Specification, it is guaranteed to be CLS-compliant and can be accessed from code in assemblies written in any programming language that supports the CLS. You can determine whether your component conforms to the Common Language Specification at compile time by applying the CLSCompliantAttribute attribute to your source code. For more information, see The CLSCompliantAttribute attribute.

In this article:

CLS compliance rules

This section discusses the rules for creating a CLS-compliant component. For a complete list of rules, see Partition I, Clause 11 of the ECMA-335 Standard: Common Language Infrastructure.

Note

The Common Language Specification discusses each rule for CLS compliance as it applies to consumers (developers who are programmatically accessing a component that is CLS-compliant), frameworks (developers who are using a language compiler to create CLS-compliant libraries), and extenders (developers who are creating a tool such as a language compiler or a code parser that creates CLS-compliant components). This article focuses on the rules as they apply to frameworks. Note, though, that some of the rules that apply to extenders may also apply to assemblies that are created using Reflection.Emit.

To design a component that is language independent, you only need to apply the rules for CLS compliance to your component's public interface. Your private implementation does not have to conform to the specification.

Important

The rules for CLS compliance apply only to a component's public interface, not to its private implementation.

For example, unsigned integers other than Byte are not CLS-compliant. Because the `Person` class in the following example exposes an `Age` property of type UInt16, the following code displays a compiler warning.

VB

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private personAge As UInt16

   Public ReadOnly Property Age As UInt16
      Get
         Return personAge
      End Get
   End Property
End Class
' The attempt to compile the example displays the following compiler warning:
'    Public1.vb(9) : warning BC40027: Return type of function 'Age' is not
CLS-compliant.
'
'       Public ReadOnly Property Age As UInt16
'                                ~~~
```

You can make the `Person` class CLS-compliant by changing the type of `Age` property from UInt16 to Int16, which is a CLS-compliant, 16-bit signed integer. You do not have to change the type of the private `personAge` field.

VB

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private personAge As UInt16

   Public ReadOnly Property Age As Int16
      Get
         Return CType(personAge, Int16)
      End Get
   End Property
End Class
```

A library's public interface consists of the following:

- Definitions of public classes.
- Definitions of the public members of public classes, and definitions of members accessible to derived classes (that is, protected members).
- Parameters and return types of public methods of public classes, and parameters and return types of methods accessible to derived classes.

The rules for CLS compliance are listed in the following table. The text of the rules is taken verbatim from the ECMA-335 Standard: Common Language Infrastructure, which is Copyright 2012 by Ecma International. More detailed information about these rules is found in the following sections.

Category	See	Rule	Rule number
Accessibility	Member accessibility	Accessibility shall not be changed when overriding inherited methods, except when overriding a method inherited from a different assembly with accessibility **family-or-assembly**. In this case, the override shall have accessibility **family**.	10
Accessibility	Member accessibility	The visibility and accessibility of types and members shall be such that types in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, a public method that is visible outside its assembly shall not have an argument whose type is visible only within the assembly. The visibility and accessibility of types composing an instantiated generic type used in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, an instantiated generic type present in the signature of a member that is visible outside its assembly shall not have a generic argument whose type is visible only within the assembly.	12
Arrays	Arrays	Arrays shall have elements with a CLS-compliant type, and all dimensions of the array shall have lower bounds of zero. Only the fact that an item is an array and the element type of the array shall be required to distinguish between overloads. When overloading is based on two or more array types the element types shall be named types.	16
Attributes	Attributes	Attributes shall be of type System.Attribute, or a type inheriting from it.	41
Attributes	Attributes	The CLS only allows a subset of the encodings of custom attributes. The only types that shall appear in these encodings are (see Partition IV): System.Type, System.String, System.Char, System.Boolean, System.Byte, System.Int16, System.Int32, System.Int64, System.Single, System.Double, and any enumeration type based on a CLS-compliant base integer type.	34
Attributes	Attributes	The CLS does not allow publicly visible required modifiers (**modreq**, see Partition II), but does allow optional modifiers (**modopt**, see Partition II) it does not understand.	35
Constructors	Constructors	An object constructor shall call some instance constructor of its base class before any access occurs to inherited instance data. (This does not apply to value types, which need not have constructors.)	21
Constructors	Constructors	An object constructor shall not be called except as part of the creation of an object, and an object shall not be initialized twice.	22

Enumerations	Enumerations	The underlying type of an enum shall be a built-in CLS integer type, the name of the field shall be "value__", and that field shall be marked **RTSpecialName**.	7
Enumerations	Enumerations	There are two distinct kinds of enums, indicated by the presence or absence of the System.FlagsAttribute (see Partition IV Library) custom attribute. One represents named integer values; the other represents named bit flags that can be combined to generate an unnamed value. The value of an **enum** is not limited to the specified values.	8
Enumerations	Enumerations	Literal static fields of an enum shall have the type of the enum itself.	9
Events	Events	The methods that implement an event shall be marked **SpecialName** in themetadata.	29
Events	Events	The accessibility of an event and of its accessors shall be identical.	30
Events	Events	The **add** and **remove** methods for an event shall both either be present or absent.	31
Events	Events	The **add** and **remove** methods for an event shall each take one parameter whose type defines the type of the event and that shall be derived from System.Delegate.	32
Events	Events	Events shall adhere to a specific naming pattern. The **SpecialName** attribute referred to in CLS rule 29 shall be ignored in appropriate name comparisons and shall adhere to identifier rules.	33
Exceptions	Exceptions	Objects that are thrown shall be of type System.Exception or a type inheriting from it. Nonetheless, CLS-compliant methods are not required to block the propagation of other types of exceptions.	40
General	CLS compliance: the Rules	CLS rules apply only to those parts of a type that are accessible or visible outsideof the defining assembly.	1
General	CLS compliance: the Rules	Members of non-CLS compliant types shall not be marked CLS-compliant.	2
Generics	Generic types and members	Nested types shall have at least as many generic parameters as the enclosing type. Generic parameters in a nested type correspond by position to the generic parameters in its enclosing type.	42
Generics	Generic types and members	The name of a generic type shall encode the number of type parameters declared on the non-nested type, or newly introduced	43

		to the type if nested, according to the rules defined above.	
Generics	Generic types and members	A generic type shall redeclare sufficient constraints to guarantee that any constraints on the base type, or interfaces would be satisfied by the generic type constraints.	4444
Generics	Generic types and members	Types used as constraints on generic parameters shall themselves be CLS-compliant.	45
Generics	Generic types and members	The visibility and accessibility of members (including nested types) in an instantiated generic type shall be considered to be scoped to the specific instantiation rather than the generic type declaration as a whole. Assuming this, the visibility and accessibility rules of CLS rule 12 still apply.	46
Generics	Generic types and members	For each abstract or virtual generic method, there shall be a default concrete (nonabstract) implementation.	47
Interfaces	Interfaces	CLS-compliant interfaces shall not require the definition of non-CLS compliantmethods in order to implement them.	18
Interfaces	Interfaces	CLS-compliant interfaces shall not define static methods, nor shall they define fields.	19
Members	Type members in general	Global static fields and methods are not CLS-compliant.	36
Members	--	The value of a literal static is specified through the use of field initialization metadata. A CLS-compliant literal must have a value specified in field initialization metadata that is of exactly the same type as the literal (or of the underlying type, if that literal is an **enum**).	13
Members	Type members in general	The vararg constraint is not part of the CLS, and the only calling convention supported by the CLS is the standard managed calling convention.	15
Naming conventions	Naming conventions	Assemblies shall follow Annex 7 of Technical Report 15 of the Unicode Standard3.0 governing the set of characters permitted to start and be included in identifiers, available onlineat http://www.unicode.org/unicode/reports/tr15/tr15-18.html. Identifiers shall be in thecanonical format defined by Unicode Normalization Form C. For CLS purposes, two identifiersare the same if their lowercase mappings (as specified by the Unicode locale-insensitive, one-toonelowercase mappings) are the same. That is, for two identifiers to be considered differentunder the CLS they shall differ in more than simply their case. However, in order to override aninherited definition the CLI requires the precise encoding of the original declaration be used.	4

Overloading	Naming conventions	All names introduced in a CLS-compliant scope shall be distinct independent ofkind, except where the names are identical and resolved via overloading. That is, while the CTSallows a single type to use the same name for a method and a field, the CLS does not.	5
Overloading	Naming conventions	Fields and nested types shall be distinct by identifier comparison alone, eventhough the CTS allows distinct signatures to be distinguished. Methods, properties, and eventsthat have the same name (by identifier comparison) shall differ by more than just the return type,except as specified in CLS Rule 39.	6
Overloading	Overloads	Only properties and methods can be overloaded.	37
Overloading	Overloads	Properties and methods can be overloaded based only on the number and types of their parameters, except the conversion operators named **op_Implicit** and **op_Explicit**, which can also be overloaded based on their return type.	38
Overloading	--	If two or more CLS-compliant methods declared in a type have the same nameand, for a specific set of type instantiations, they have the same parameter and return types, thenall these methods shall be semantically equivalent at those type instantiations.	48
Types	Type and type member signatures	System.Object is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS-compliant class.	23
Properties	Properties	The methods that implement the getter and setter methods of a property shallbe marked **SpecialName** in the metadata.	24
Properties	Properties	A property's accessors shall all be static, all be virtual, or all be instance.	26
Properties	Properties	The type of a property shall be the return type of the getter and the type of the last argument of the setter. The types of the parameters of the property shall be the types of the parameters to the getter and the types of all but the final parameter of the setter. All of these types shall be CLS-compliant, and shall not be managed pointers (i.e., shall not be passed by reference).	27
Properties	Properties	Properties shall adhere to a specific naming pattern. The **SpecialName** attribute referred to in CLS rule 24 shall be ignored in appropriate name comparisons and shall adhere to identifier rules. A property shall have a getter method, a setter method, or both.	28
Type conversion	Type conversion	If either **op_Implicit** or **op_Explicit** is provided, an alternate means of providing the coercion shall be provided.	39

Types	Type and type member signatures	Boxed value types are not CLS-compliant.	3
Types	Type and type member signatures	All types appearing in a signature shall be CLS-compliant. All types composing an instantiated generic type shall be CLS-compliant.	11
Types	Type and type member signatures	Typed references are not CLS-compliant.	14
Types	Type and type member signatures	Unmanaged pointer types are not CLS-compliant.	17
Types	Type and type member signatures	CLS-compliant classes, value types, and interfaces shall not require the implementation of non-CLS-compliant members.	20

Types and type member signatures

The System.Object type is CLS-compliant and is the base type of all object types in the .NET Framework type system. Inheritance in the .NET Framework is either implicit (for example, the String class implicitly inherits from the Object class) or explicit (for example, the CultureNotFoundException class explicitly inherits from the ArgumentException class, which explicitly inherits from the SystemException class, which explicitly inherits from the Exception class). For a derived type to be CLS compliant, its base type must also be CLS-compliant.

The following example shows a derived type whose base type is not CLS-compliant. It defines a base `Counter` class that uses an unsigned 32-bit integer as a counter. Because the class provides counter functionality by wrapping an unsigned integer, the class is marked as non-CLS-compliant. As a result, a derived class, `NonZeroCounter`, is also not CLS-compliant.

VB

```
<Assembly: CLSCompliant(True)>

<CLSCompliant(False)> _
Public Class Counter
   Dim ctr As UInt32

   Public Sub New
      ctr = 0
   End Sub

   Protected Sub New(ctr As UInt32)
      ctr = ctr
   End Sub
```

```
    Public Overrides Function ToString() As String
       Return String.Format("{0}). ", ctr)
    End Function

    Public ReadOnly Property Value As UInt32
       Get
          Return ctr
       End Get
    End Property

    Public Sub Increment()
       ctr += CType(1, UInt32)
    End Sub
End Class

Public Class NonZeroCounter : Inherits Counter
    Public Sub New(startIndex As Integer)
       MyClass.New(CType(startIndex, UInt32))
    End Sub

    Private Sub New(startIndex As UInt32)
       MyBase.New(CType(startIndex, UInt32))
    End Sub
End Class
' Compilation produces a compiler warning like the following:
'    Type3.vb(34) : warning BC40026: 'NonZeroCounter' is not CLS-compliant
'    because it derives from 'Counter', which is not CLS-compliant.
'
'    Public Class NonZeroCounter : Inherits Counter
'                 ~~~~~~~~~~~~~~
```

All types that appear in member signatures, including a method's return type or a property type, must be CLS-compliant. In addition, for generic types:

- All types that compose an instantiated generic type must be CLS-compliant.
- All types used as constraints on generic parameters must be CLS-compliant.

The .NET Framework common type system includes a number of built-in types that are supported directly by the common language runtime and are specially encoded in an assembly's metadata. Of these intrinsic types, the types listed in the following table are CLS-compliant.

CLS-compliant type	Description
Byte	8-bit unsigned integer
Int16	16-bit signed integer
Int32	32-bit signed integer

Int64	64-bit signed integer
Single	Single-precision floating-point value
Double	Double-precision floating-point value
Boolean	**true** or **false** value type
Char	UTF-16 encoded code unit
Decimal	Non-floating-point decimal number
IntPtr	Pointer or handle of a platform-defined size
String	Collection of zero, one, or more Char objects

The intrinsic types listed in the following table are not CLS-Compliant.

Non-compliant type	Description	CLS-compliant alternative
SByte	8-bit signed integer data type	Int16
TypedReference	Pointer to an object and its runtime type	None
UInt16	16-bit unsigned integer	Int32
UInt32	32-bit unsigned integer	Int64
UInt64	64-bit unsigned integer	Int64 (may overflow), BigInteger, or Double
UIntPtr	Unsigned pointer or handle	IntPtr

The .NET Framework Class Library or any other class library may include other types that aren't CLS-compliant; for example:

- Boxed value types. The following C# example creates a class that has a public property of type **int*** named `Value`. Because an **int*** is a boxed value type, the compiler flags it as non-CLS-compliant.

C#

```
using System;

[assembly:CLSCompliant(true)]
```

```
public unsafe class TestClass
{
   private int* val;

   public TestClass(int number)
   {
      val = (int*) number;
   }

   public int* Value {
      get { return val; }
   }
}
// The compiler generates the following output when compiling this example:
//        warning CS3003: Type of 'TestClass.Value' is not CLS-compliant
```

- Typed references, which are special constructs that contain a reference to an object and a reference to a type. Typed references are represented in the .NET Framework by the TypedReference class.

If a type is not CLS-compliant, you should apply the CLSCompliantAttribute attribute with an *isCompliant* value of **false** to it. For more information, see the The CLSCompliantAttribute attribute section.

The following example illustrates the problem of CLS compliance in a method signature and in generic type instantiation. It defines an `InvoiceItem` class with a property of type UInt32, a property of type **Nullable(Of UInt32)**, and a constructor with parameters of type UInt32 and **Nullable(Of UInt32)**. You get four compiler warnings when you try to compile this example.

VB

```
<Assembly: CLSCompliant(True)>

Public Class InvoiceItem

   Private invId As UInteger = 0
   Private itemId As UInteger = 0
   Private qty AS Nullable(Of UInteger)

   Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
      itemId = sku
      qty = quantity
   End Sub

   Public Property Quantity As Nullable(Of UInteger)
      Get
         Return qty
      End Get
      Set
         qty = value
      End Set
   End Property

   Public Property InvoiceId As UInteger
```

```
      Get
         Return invId
      End Get
      Set
         invId = value
      End Set
   End Property
End Class
' The attempt to compile the example displays output similar to the following:
'    Type1.vb(13) : warning BC40028: Type of parameter 'sku' is not CLS-compliant.
'
'       Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
'                      ~~~
'    Type1.vb(13) : warning BC40041: Type 'UInteger' is not CLS-compliant.
'
'       Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
'                                                           ~~~~~~~~
'    Type1.vb(18) : warning BC40041: Type 'UInteger' is not CLS-compliant.
'
'       Public Property Quantity As Nullable(Of UInteger)
'                                               ~~~~~~~~
'    Type1.vb(27) : warning BC40027: Return type of function 'InvoiceId' is not
CLS-compliant.
'
'       Public Property InvoiceId As UInteger
'                       ~~~~~~~~~
```

To eliminate the compiler warnings, replace the non-CLS-compliant types in the `InvoiceItem` public interface with compliant types:

VB

```
<Assembly: CLSCompliant(True)>

Public Class InvoiceItem

   Private invId As UInteger = 0
   Private itemId As UInteger = 0
   Private qty AS Nullable(Of Integer)

   Public Sub New(sku As Integer, quantity As Nullable(Of Integer))
      If sku <= 0 Then
         Throw New ArgumentOutOfRangeException("The item number is zero or negative.")
      End If
      itemId = CUInt(sku)
      qty = quantity
   End Sub

   Public Property Quantity As Nullable(Of Integer)
      Get
         Return qty
      End Get
      Set
```

```
            qty = value
        End Set
    End Property

    Public Property InvoiceId As Integer
        Get
            Return CInt(invId)
        End Get
        Set
            invId = CUInt(value)
        End Set
    End Property
End Class
```

In addition to the specific types listed, some categories of types are not CLS compliant. These include unmanaged pointer types and function pointer types. The following example generates a compiler warning because it uses a pointer to an integer to create an array of integers.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class ArrayHelper
{
   unsafe public static Array CreateInstance(Type type, int* ptr, int items)
   {
      Array arr = Array.CreateInstance(type, items);
      int* addr = ptr;
      for (int ctr = 0; ctr < items; ctr++) {
          int value = *addr;
          arr.SetValue(value, ctr);
          addr++;
      }
      return arr;
   }
}
// The attempt to compile this example displays the following output:
//    UnmanagedPtr1.cs(8,57): warning CS3001: Argument type 'int*' is not
CLS-compliant
```

For CLS-compliant abstract classes (that is, classes marked as **abstract** in C# or as **MustInherit** in Visual Basic), all members of the class must also be CLS-compliant.

Naming conventions

Because some programming languages are case-insensitive, identifiers (such as the names of namespaces, types, and members) must differ by more than case. Two identifiers are considered equivalent if their lowercase mappings are the same. The following C# example defines two public classes, `Person` and `person`. Because they differ only by case, the

C# compiler flags them as not CLS-compliant.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Person : person
{

}

public class person
{

}
// Compilation produces a compiler warning like the following:
//    Naming1.cs(11,14): warning CS3005: Identifier 'person' differing
//                       only in case is not CLS-compliant
//    Naming1.cs(6,14): (Location of symbol related to previous warning)
```

Programming language identifiers, such as the names of namespaces, types, and members, must conform to the Unicode Standard 3.0, Technical Report 15, Annex 7. This means that:

- The first character of an identifier can be any Unicode uppercase letter, lowercase letter, title case letter, modifier letter, other letter, or letter number. For information on Unicode character categories, see the System.Globalization.UnicodeCategory enumeration.
- Subsequent characters can be from any of the categories as the first character, and can also include non-spacing marks, spacing combining marks, decimal numbers, connector punctuations, and formatting codes.

Before you compare identifiers, you should filter out formatting codes and convert the identifiers to Unicode Normalization Form C, because a single character can be represented by multiple UTF-16-encoded code units. Character sequences that produce the same code units in Unicode Normalization Form C are not CLS-compliant. The following example defines a property named **Å**, which consists of the character ANGSTROM SIGN (U+212B), and a second property named **Å**, which consists of the character LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5). Both the C# and Visual Basic compilers flag the source code as non-CLS-compliant.

VB

```
<Assembly: CLSCompliant(True)>
Public Class Size
   Private a1 As Double
   Private a2 As Double

   Public Property Å As Double
       Get
          Return a1
       End Get
       Set
          a1 = value
```

```
        End Set
    End Property

    Public Property Å As Double
        Get
           Return a2
        End Get
        Set
           a2 = value
        End Set
    End Property
End Class
' Compilation produces a compiler warning like the following:
'    Naming1.vb(9) : error BC30269: 'Public Property Å As Double' has multiple
definitions
'     with identical signatures.
'
'       Public Property Å As Double
'                       ~
```

Member names within a particular scope (such as the namespaces within an assembly, the types within a namespace, or the members within a type) must be unique except for names that are resolved through overloading. This requirement is more stringent than that of the common type system, which allows multiple members within a scope to have identical names as long as they are different kinds of members (for example, one is a method and one is a field). In particular, for type members:

- Fields and nested types are distinguished by name alone.
- Methods, properties, and events that have the same name must differ by more than just return type.

The following example illustrates the requirement that member names must be unique within their scope. It defines a class named `Converter` that includes four members named `Conversion`. Three are methods, and one is a property. The method that includes an Int64 parameter is uniquely named, but the two methods with an Int32 parameter are not, because return value is not considered a part of a member's signature. The `Conversion` property also violates this requirement, because properties cannot have the same name as overloaded methods.

VB

```
<Assembly: CLSCompliant(True)>

Public Class Converter
    Public Function Conversion(number As Integer) As Double
        Return CDbl(number)
    End Function

    Public Function Conversion(number As Integer) As Single
        Return CSng(number)
    End Function

    Public Function Conversion(number As Long) As Double
        Return CDbl(number)
    End Function
```

```
   Public ReadOnly Property Conversion As Boolean
      Get
         Return True
      End Get
   End Property
End Class
' Compilation produces a compiler error like the following:
'    Naming3.vb(8) : error BC30301: 'Public Function Conversion(number As Integer) As
Double'
'                    and 'Public Function Conversion(number As Integer) As Single'
cannot
'                    overload each other because they differ only by return types.
'
'       Public Function Conversion(number As Integer) As Double
'                       ~~~~~~~~~~
'    Naming3.vb(20) : error BC30260: 'Conversion' is already declared as 'Public
Function
'                     Conversion(number As Integer) As Single' in this class.
'
'       Public ReadOnly Property Conversion As Boolean
'                                ~~~~~~~~~~
```

Individual languages include unique keywords, so languages that target the common language runtime must also provide some mechanism for referencing identifiers (such as type names) that coincide with keywords. For example, **case** is a keyword in both C# and Visual Basic. However, the following Visual Basic example is able to disambiguate a class named `case` from the **case** keyword by using opening and closing braces. Otherwise, the example would produce the error message, "Keyword is not valid as an identifier," and fail to compile.

VB

```
Public Class [case]
   Private _id As Guid
   Private name As String

   Public Sub New(name As String)
      _id = Guid.NewGuid()
      Me.name = name
   End Sub

   Public ReadOnly Property ClientName As String
      Get
         Return name
      End Get
   End Property
End Class
```

The following C# example is able to instantiate the `case` class by using the **@** symbol to disambiguate the identifier from the language keyword. Without it, the C# compiler would display two error messages, "Type expected" and "Invalid expression term 'case'."

C#

```
using System;

public class Example
{
   public static void Main()
   {
      @case c = new @case("John");
      Console.WriteLine(c.ClientName);
   }
}
```

Type conversion

The Common Language Specification defines two conversion operators:

- **op_Implicit**, which is used for widening conversions that do not result in loss of data or precision. For example, the Decimal structure includes an overloaded **op_Implicit** operator to convert values of integral types and Char values to Decimal values.
- **op_Explicit**, which is used for narrowing conversions that can result in loss of magnitude (a value is converted to a value that has a smaller range) or precision. For example, the Decimal structure includes an overloaded **op_Explicit** operator to convert Double and Single values to Decimal and to convert Decimal values to integral values, Double, Single, and Char.

However, not all languages support operator overloading or the definition of custom operators. If you choose to implement these conversion operators, you should also provide an alternate way to perform the conversion. We recommend that you provide **From***Xxx* and **To***Xxx* methods.

The following example defines CLS-compliant implicit and explicit conversions. It creates a `UDouble` class that represents an signed double-precision, floating-point number. It provides for implicit conversions from `UDouble` to Double and for explicit conversions from `UDouble` to Single, Double to `UDouble`, and Single to `UDouble`. It also defines a `ToDouble` method as an alternative to the implicit conversion operator and the `ToSingle`, `FromDouble`, and `FromSingle` methods as alternatives to the explicit conversion operators.

VB

```
Public Structure UDouble
   Private number As Double

   Public Sub New(value As Double)
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a
UDouble.")
      End If
      number = value
   End Sub

   Public Sub New(value As Single)
      If value < 0 Then
```

```
         Throw New InvalidCastException("A negative value cannot be converted to a
UDouble.")
      End If
      number = value
   End Sub

   Public Shared ReadOnly MinValue As UDouble = CType(0.0, UDouble)
   Public Shared ReadOnly MaxValue As UDouble = Double.MaxValue

   Public Shared Widening Operator CType(value As UDouble) As Double
      Return value.number
   End Operator

   Public Shared Narrowing Operator CType(value As UDouble) As Single
      If value.number > CDbl(Single.MaxValue) Then
         Throw New InvalidCastException("A UDouble value is out of range of the
Single type.")
      End If
      Return CSng(value.number)
   End Operator

   Public Shared Narrowing Operator CType(value As Double) As UDouble
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a
UDouble.")
      End If
      Return New UDouble(value)
   End Operator

   Public Shared Narrowing Operator CType(value As Single) As UDouble
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a
UDouble.")
      End If
      Return New UDouble(value)
   End Operator

   Public Shared Function ToDouble(value As UDouble) As Double
      Return CType(value, Double)
   End Function

   Public Shared Function ToSingle(value As UDouble) As Single
      Return CType(value, Single)
   End Function

   Public Shared Function FromDouble(value As Double) As UDouble
      Return New UDouble(value)
   End Function

   Public Shared Function FromSingle(value As Single) As UDouble
      Return New UDouble(value)
   End Function
End Structure
```

Arrays

CLS-compliant arrays conform to the following rules:

- All dimensions of an array must have a lower bound of zero. The following example creates a non-CLS-compliant array with a lower bound of one. Note that, despite the presence of the CLSCompliantAttribute attribute, the compiler does not detect that the array returned by the `Numbers.GetTenPrimes` method is not CLS-compliant.

VB

```
<Assembly: CLSCompliant(True)>

Public Class Numbers
   Public Shared Function GetTenPrimes() As Array
      Dim arr As Array = Array.CreateInstance(GetType(Int32), {10}, {1})
      arr.SetValue(1, 1)
      arr.SetValue(2, 2)
      arr.SetValue(3, 3)
      arr.SetValue(5, 4)
      arr.SetValue(7, 5)
      arr.SetValue(11, 6)
      arr.SetValue(13, 7)
      arr.SetValue(17, 8)
      arr.SetValue(19, 9)
      arr.SetValue(23, 10)

      Return arr
   End Function
End Class
```

- All array elements must consist of CLS-compliant types. The following example defines two methods that return non-CLS-compliant arrays. The first returns an array of UInt32 values. The second returns an Object array that includes Int32 and UInt32 values. Although the compiler identifies the first array as non-compliant because of its UInt32 type, it fails to recognize that the second array includes non-CLS-compliant elements.

VB

```
<Assembly: CLSCompliant(True)>

Public Class Numbers
   Public Shared Function GetTenPrimes() As UInt32()
      Return { 1ui, 2ui, 3ui, 5ui, 7ui, 11ui, 13ui, 17ui, 19ui }
   End Function

   Public Shared Function GetFivePrimes() As Object()
      Dim arr() As Object = { 1, 2, 3, 5ui, 7ui }
      Return arr
   End Function
End Class
' Compilation produces a compiler warning like the following:
'    warning BC40027: Return type of function 'GetTenPrimes' is not
```

```
CLS-compliant.
'
'       Public Shared Function GetTenPrimes() As UInt32()
'                              ~~~~~~~~~~~~
```

- Overload resolution for methods that have array parameters is based on the fact that they are arrays and on their element type. For this reason, the following definition of an overloaded `GetSquares` method is CLS-compliant.

VB

```
Imports System.Numerics

<Assembly: CLSCompliant(True)>

Public Module Numbers
   Public Function GetSquares(numbers As Byte()) As Byte()
      Dim numbersOut(numbers.Length - 1) As Byte
      For ctr As Integer = 0 To numbers.Length - 1
         Dim square As Integer = (CInt(numbers(ctr)) * CInt(numbers(ctr)))
         If square <= Byte.MaxValue Then
            numbersOut(ctr) = CByte(square)
         ' If there's an overflow, assign MaxValue to the corresponding
         ' element.
         Else
            numbersOut(ctr) = Byte.MaxValue
         End If
      Next
      Return numbersOut
   End Function

   Public Function GetSquares(numbers As BigInteger()) As BigInteger()
      Dim numbersOut(numbers.Length - 1) As BigInteger
      For ctr As Integer = 0 To numbers.Length - 1
         numbersOut(ctr) = numbers(ctr) * numbers(ctr)
      Next
      Return numbersOut
   End Function
End Module
```

Interfaces

CLS-compliant interfaces can define properties, events, and virtual methods (methods with no implementation). A CLS-compliant interface cannot have any of the following:

- Static methods or static fields. Both the C# and Visual Basic compilers generate compiler errors if you define a static member in an interface.
- Fields. Both the C# and Visual Basic compilers generate compiler errors if you define a field in an interface.
- Methods that are not CLS-compliant. For example, the following interface definition includes a method,

`INumber.GetUnsigned`, that is marked as non-CLS-compliant. This example generates a compiler warning.

VB

```
<Assembly: CLSCompliant(True)>

Public Interface INumber
   Function Length As Integer

   <CLSCompliant(False)> Function GetUnsigned As ULong
End Interface
' Attempting to compile the example displays output like the following:
'    Interface2.vb(9) : warning BC40033: Non CLS-compliant 'function' is not
allowed in a
'    CLS-compliant interface.
'
'       <CLSCompliant(False)> Function GetUnsigned As ULong
'                                      ~~~~~~~~~~~
```

Because of this rule, CLS-compliant types are not required to implement non-CLS-compliant members. If a CLS-compliant framework does expose a class that implements a non-CLS compliant interface, it should also provide concrete implementations of all non-CLS-compliant members.

CLS-compliant language compilers must also allow a class to provide separate implementations of members that have the same name and signature in multiple interfaces. Both C# and Visual Basic support explicit interface implementations to provide different implementations of identically named methods. Visual Basic also supports the **Implements** keyword, which enables you to explicitly designate which interface and member a particular member implements. The following example illustrates this scenario by defining a `Temperature` class that implements the `ICelsius` and `IFahrenheit` interfaces as explicit interface implementations.

VB

```
<Assembly: CLSCompliant(True)>

Public Interface IFahrenheit
   Function GetTemperature() As Decimal
End Interface

Public Interface ICelsius
   Function GetTemperature() As Decimal
End Interface

Public Class Temperature : Implements ICelsius, IFahrenheit
   Private _value As Decimal

   Public Sub New(value As Decimal)
      ' We assume that this is the Celsius value.
      _value = value
   End Sub

   Public Function GetFahrenheit() As Decimal _
          Implements IFahrenheit.GetTemperature
```

```
        Return _value * 9 / 5 + 32
    End Function

    Public Function GetCelsius() As Decimal _
           Implements ICelsius.GetTemperature
        Return _value
    End Function
End Class

Module Example
    Public Sub Main()
        Dim temp As New Temperature(100.0d)
        Console.WriteLine("Temperature in Celsius: {0} degrees",
                          temp.GetCelsius())
        Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                          temp.GetFahrenheit())
    End Sub
End Module
' The example displays the following output:
'       Temperature in Celsius: 100.0 degrees
'       Temperature in Fahrenheit: 212.0 degrees
```

Enumerations

CLS-compliant enumerations must follow these rules:

- The underlying type of the enumeration must be an intrinsic CLS-compliant integer (Byte, Int16, Int32, or Int64). For example, the following code tries to define an enumeration whose underlying type is UInt32 and generates a compiler warning.

VB

```
<Assembly: CLSCompliant(True)>

Public Enum Size As UInt32
   Unspecified = 0
   XSmall = 1
   Small = 2
   Medium = 3
   Large = 4
   XLarge = 5
End Enum

Public Class Clothing
   Public Name As String
   Public Type As String
   Public Size As Size
End Class
' The attempt to compile the example displays a compiler warning like the
following:
'    Enum3.vb(6) : warning BC40032: Underlying type 'UInt32' of Enum is not
```

```
CLS-compliant.
'
'    Public Enum Size As UInt32
'                ~~~~
```

- An enumeration type must have a single instance field named **Value__** that is marked with the FieldAttributes.RTSpecialName attribute. This enables you to reference the field value implicitly.
- An enumeration includes literal static fields whose types match the type of the enumeration itself. For example, if you define a `State` enumeration with values of `State.On` and `State.Off`, `State.On` and `State.Off` are both literal static fields whose type is `State`.
- There are two kinds of enumerations:
  - An enumeration that represents a set of mutually exclusive, named integer values. This type of enumeration is indicated by the absence of the System.FlagsAttribute custom attribute.
  - An enumeration that represents a set of bit flags that can combine to generate an unnamed value. This type of enumeration is indicated by the presence of the System.FlagsAttribute custom attribute.

  For more information, see the documentation for the Enum structure.
- The value of an enumeration is not limited to the range of its specified values. In other words, the range of values in an enumeration is the range of its underlying value. You can use the Enum.IsDefined method to determine whether a specified value is a member of an enumeration.

Type members in general

The Common Language Specification requires all fields and methods to be accessed as members of a particular class. Therefore, global static fields and methods (that is, static fields or methods that are defined apart from a type) are not CLS-compliant. If you try to include a global field or method in your source code, both the C# and Visual Basic compilers generate a compiler error.

The Common Language Specification supports only the standard managed calling convention. It doesn't support unmanaged calling conventions and methods with variable argument lists marked with the **varargs** keyword. For variable argument lists that are compatible with the standard managed calling convention, use the ParamArrayAttribute attribute or the individual language's implementation, such as the **params** keyword in C# and the **ParamArray** keyword in Visual Basic.

Member accessibility

Overriding an inherited member cannot change the accessibility of that member. For example, a public method in a base class cannot be overridden by a private method in a derived class. There is one exception: a **protected internal** (in C#) or **Protected Friend** (in Visual Basic) member in one assembly that is overridden by a type in a different assembly. In that case, the accessibility of the override is **Protected**.

The following example illustrates the error that is generated when the CLSCompliantAttribute attribute is set to **true**, and `Person`, which is a class derived from `Animal`, tries to change the accessibility of the `Species` property from public to private. The example compiles successfully if its accessibility is changed to public.

VB

```
<Assembly: CLSCompliant(True)>

Public Class Animal
   Private _species As String

   Public Sub New(species As String)
      _species = species
   End Sub

   Public Overridable ReadOnly Property Species As String
      Get
         Return _species
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return _species
   End Function
End Class

Public Class Human : Inherits Animal
   Private _name As String

   Public Sub New(name As String)
      MyBase.New("Homo Sapiens")
      _name = name
   End Sub

   Public ReadOnly Property Name As String
      Get
         Return _name
      End Get
   End Property

   Private Overrides ReadOnly Property Species As String
      Get
         Return MyBase.Species
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return _name
   End Function
End Class

Public Module Example
   Public Sub Main()
      Dim p As New Human("John")
      Console.WriteLine(p.Species)
      Console.WriteLine(p.ToString())
   End Sub
End Module
```

```
' The example displays the following output:
'     'Private Overrides ReadOnly Property Species As String' cannot override
'     'Public Overridable ReadOnly Property Species As String' because
'      they have different access levels.
'
'         Private Overrides ReadOnly Property Species As String
```

Types in the signature of a member must be accessible whenever that member is accessible. For example, this means that a public member cannot include a parameter whose type is private, protected, or internal. The following example illustrates the compiler error that results when a `StringWrapper` class constructor exposes an internal `StringOperationType` enumeration value that determines how a string value should be wrapped.

VB

```
Imports System.Text

<Assembly:CLSCompliant(True)>

Public Class StringWrapper

   Dim internalString As String
   Dim internalSB As StringBuilder = Nothing
   Dim useSB As Boolean = False

   Public Sub New(type As StringOperationType)
      If type = StringOperationType.Normal Then
         useSB = False
      Else
         internalSB = New StringBuilder()
         useSB = True
      End If
   End Sub

   ' The remaining source code...
End Class

Friend Enum StringOperationType As Integer
   Normal = 0
   Dynamic = 1
End Enum
' The attempt to compile the example displays the following output:
'    error BC30909: 'type' cannot expose type 'StringOperationType'
'     outside the project through class 'StringWrapper'.
'
'       Public Sub New(type As StringOperationType)
'                                ~~~~~~~~~~~~~~~~~~
```

Generic types and members

Nested types always have at least as many generic parameters as their enclosing type. These correspond by position to

the generic parameters in the enclosing type. The generic type can also include new generic parameters.

The relationship between the generic type parameters of a containing type and its nested types may be hidden by the syntax of individual languages. In the following example, a generic type `Outer<T>` contains two nested classes, `Inner1A` and `Inner1B<U>`. The calls to the **ToString** method, which each class inherits from Object.ToString, show that each nested class includes the type parameters of its containing class.

VB

```
<Assembly:CLSCompliant(True)>

Public Class Outer(Of T)
   Dim value As T

   Public Sub New(value As T)
      Me.value = value
   End Sub

   Public Class Inner1A : Inherits Outer(Of T)
      Public Sub New(value As T)
         MyBase.New(value)
      End Sub
   End Class

   Public Class Inner1B(Of U) : Inherits Outer(Of T)
      Dim value2 As U

      Public Sub New(value1 As T, value2 As U)
         MyBase.New(value1)
         Me.value2 = value2
      End Sub
   End Class
End Class

Public Module Example
   Public Sub Main()
      Dim inst1 As New Outer(Of String)("This")
      Console.WriteLine(inst1)

      Dim inst2 As New Outer(Of String).Inner1A("Another")
      Console.WriteLine(inst2)

      Dim inst3 As New Outer(Of String).Inner1B(Of Integer)("That", 2)
      Console.WriteLine(inst3)
   End Sub
End Module
' The example displays the following output:
'       Outer`1[System.String]
'       Outer`1+Inner1A[System.String]
'       Outer`1+Inner1B`1[System.String,System.Int32]
```

Generic type names are encoded in the form *name*`*n*, where *name* is the type name, ` is a character literal, and *n* is the

number of parameters declared on the type, or, for nested generic types, the number of newly introduced type parameters. This encoding of generic type names is primarily of interest to developers who use reflection to access CLS-complaint generic types in a library.

If constraints are applied to a generic type, any types used as constraints must also be CLS-compliant. The following example defines a class named `BaseClass` that is not CLS-compliant and a generic class named `BaseCollection` whose type parameter must derive from `BaseClass`. But because `BaseClass` is not CLS-compliant, the compiler emits a warning.

VB

```
<Assembly: CLSCompliant(True)>

<CLSCompliant(False)> Public Class BaseClass
End Class


Public Class BaseCollection(Of T As BaseClass)
End Class
' Attempting to compile the example displays the following output:
'    warning BC40040: Generic parameter constraint type 'BaseClass' is not
'    CLS-compliant.
'
'    Public Class BaseCollection(Of T As BaseClass)
'                                       ~~~~~~~~~
```

If a generic type is derived from a generic base type, it must redeclare any constraints so that it can guarantee that constraints on the base type are also satisfied. The following example defines a `Number<T>` that can represent any numeric type. It also defines a `FloatingPoint<T>` class that represents a floating point value. However, the source code fails to compile, because it does not apply the constraint on `Number<T>` (that T must be a value type) to `FloatingPoint<T>`.

VB

```
<Assembly:CLSCompliant(True)>

Public Class Number(Of T As Structure)
   ' Use Double as the underlying type, since its range is a superset of
   ' the ranges of all numeric types except BigInteger.
   Protected number As Double

   Public Sub New(value As T)
      Try
         Me.number = Convert.ToDouble(value)
      Catch e As OverflowException
         Throw New ArgumentException("value is too large.", e)
      Catch e As InvalidCastException
         Throw New ArgumentException("The value parameter is not numeric.", e)
      End Try
   End Sub

   Public Function Add(value As T) As T
```

```
      Return CType(Convert.ChangeType(number + Convert.ToDouble(value), GetType(T)),
T)
   End Function

   Public Function Subtract(value As T) As T
      Return CType(Convert.ChangeType(number - Convert.ToDouble(value), GetType(T)),
T)
   End Function
End Class

Public Class FloatingPoint(Of T) : Inherits Number(Of T)
   Public Sub New(number As T)
      MyBase.New(number)
      If TypeOf number Is Single Or
               TypeOf number Is Double Or
               TypeOf number Is Decimal Then
         Me.number = Convert.ToDouble(number)
      Else
         throw new ArgumentException("The number parameter is not a floating-point
number.")
      End If
   End Sub
End Class
' The attempt to comple the example displays the following output:
'    error BC32105: Type argument 'T' does not satisfy the 'Structure'
'    constraint for type parameter 'T'.
'
'    Public Class FloatingPoint(Of T) : Inherits Number(Of T)
'                                                        ~
```

The example compiles successfully if the constraint is added to the `FloatingPoint<T>` class.

VB

```
<Assembly:CLSCompliant(True)>

Public Class Number(Of T As Structure)
   ' Use Double as the underlying type, since its range is a superset of
   ' the ranges of all numeric types except BigInteger.
   Protected number As Double

   Public Sub New(value As T)
      Try
         Me.number = Convert.ToDouble(value)
      Catch e As OverflowException
         Throw New ArgumentException("value is too large.", e)
      Catch e As InvalidCastException
         Throw New ArgumentException("The value parameter is not numeric.", e)
      End Try
   End Sub

   Public Function Add(value As T) As T
      Return CType(Convert.ChangeType(number + Convert.ToDouble(value), GetType(T)),
```

```
T)
   End Function

   Public Function Subtract(value As T) As T
      Return CType(Convert.ChangeType(number - Convert.ToDouble(value), GetType(T)),
T)
   End Function
End Class

Public Class FloatingPoint(Of T As Structure) : Inherits Number(Of T)
   Public Sub New(number As T)
      MyBase.New(number)
      If TypeOf number Is Single Or
               TypeOf number Is Double Or
               TypeOf number Is Decimal Then
         Me.number = Convert.ToDouble(number)
      Else
         throw new ArgumentException("The number parameter is not a floating-point
number.")
      End If
   End Sub
End Class
```

The Common Language Specification imposes a conservative per-instantiation model for nested types and protected members. Open generic types cannot expose fields or members with signatures that contain a specific instantiation of a nested, protected generic type. Non-generic types that extend a specific instantiation of a generic base class or interface cannot expose fields or members with signatures that contain a different instantiation of a nested, protected generic type.

The following example defines a generic type, `C1<T>` (or `C1(Of T)` in Visual Basic), and a protected class, `C1<T>.N` (or `C1(Of T).N` in Visual Basic). `C1<T>` has two methods, `M1` and `M2`. However, `M1` is not CLS-compliant because it tries to return a `C1<int>.N` (or `C1(Of Integer).N`) object from C1<T> (or `C1(Of T)`). A second class, `C2`, is derived from `C1<long>` (or `C1(Of Long)`). It has two methods, `M3` and `M4`. `M3` is not CLS-compliant because it tries to return a `C1<int>.N` (or `C1(Of Integer).N`) object from a subclass of `C1<long>`. Note that language compilers can be even more restrictive. In this example, Visual Basic displays an error when it tries to compile `M4`.

VB

```
<Assembly:CLSCompliant(True)>

Public Class C1(Of T)
   Protected Class N
   End Class

   Protected Sub M1(n As C1(Of Integer).N)    ' Not CLS-compliant - C1<int>.N not
                                              ' accessible from within C1(Of T) in all
   End Sub                                    ' languages


   Protected Sub M2(n As C1(Of T).N)      ' CLS-compliant – C1(Of T).N accessible
   End Sub                                ' inside C1(Of T)
End Class
```

```
Public Class C2 : Inherits C1(Of Long)
   Protected Sub M3(n As C1(Of Integer).N)    ' Not CLS-compliant – C1(Of Integer).N
is not
   End Sub                                    ' accessible in C2 (extends C1(Of Long))

   Protected Sub M4(n As C1(Of Long).N)
   End Sub
End Class
' Attempting to compile the example displays output like the following:
'    error BC30508: 'n' cannot expose type 'C1(Of Integer).N' in namespace
'    '<Default>' through class 'C1'.
'
'       Protected Sub M1(n As C1(Of Integer).N)    ' Not CLS-compliant - C1<int>.N not
'                                 ~~~~~~~~~~~~~~~~
'    error BC30389: 'C1(Of T).N' is not accessible in this context because
'    it is 'Protected'.
'
'       Protected Sub M3(n As C1(Of Integer).N)    ' Not CLS-compliant - C1(Of
Integer).N is not
'
'                                 ~~~~~~~~~~~~~~~~
'
'    error BC30389: 'C1(Of T).N' is not accessible in this context because it is
'Protected'.
'
'       Protected Sub M4(n As C1(Of Long).N)
'                                 ~~~~~~~~~~~~
```

Constructors

Constructors in CLS-compliant classes and structures must follow these rules:

- A constructor of a derived class must call the instance constructor of its base class before it accesses inherited instance data. This requirement is due to the fact that base class constructors are not inherited by their derived classes. This rule does not apply to structures, which do not support direct inheritance.

  Typically, compilers enforce this rule independently of CLS compliance, as the following example shows. It creates a `Doctor` class that is derived from a `Person` class, but the `Doctor` class fails to call the `Person` class constructor to initialize inherited instance fields.

  VB

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private fName, lName, _id As String

   Public Sub New(firstName As String, lastName As String, id As String)
      If String.IsNullOrEmpty(firstName + lastName) Then
         Throw New ArgumentNullException("Either a first name or a last name
must be provided.")
```

```
        End If

        fName = firstName
        lName = lastName
        _id = id
    End Sub

    Public ReadOnly Property FirstName As String
        Get
            Return fName
        End Get
    End Property

    Public ReadOnly Property LastName As String
        Get
            Return lName
        End Get
    End Property

    Public ReadOnly Property Id As String
        Get
            Return _id
        End Get
    End Property

    Public Overrides Function ToString() As String
        Return String.Format("{0}{1}{2}", fName,
                             If(String.IsNullOrEmpty(fName), "", " "),
                             lName)
    End Function
End Class

Public Class Doctor : Inherits Person
    Public Sub New(firstName As String, lastName As String, id As String)
    End Sub

    Public Overrides Function ToString() As String
        Return "Dr. " + MyBase.ToString()
    End Function
End Class
' Attempting to compile the example displays output like the following:
'    Ctor1.vb(46) : error BC30148: First statement of this 'Sub New' must be a
call
'    to 'MyBase.New' or 'MyClass.New' because base class 'Person' of 'Doctor'
does
'    not have an accessible 'Sub New' that can be called with no arguments.
'
'       Public Sub New()
'                  ~~~
```

- An object constructor cannot be called except to create an object. In addition, an object cannot be initialized twice. For example, this means that Object.MemberwiseClone and deserialization methods such as BinaryFormatter.Deserialize must not call constructors.

Properties

Properties in CLS-compliant types must follow these rules:

- A property must have a setter, a getter, or both. In an assembly, these are implemented as special methods, which means that they will appear as separate methods (the getter is named **get**_*propertyname* and the setter is **set**_*propertyname*) marked as `SpecialName` in the assembly's metadata. The C# and Visual Basic compilers enforce this rule automatically without the need to apply the CLSCompliantAttribute attribute.
- A property's type is the return type of the property getter and the last argument of the setter. These types must be CLS compliant, and arguments cannot be assigned to the property by reference (that is, they cannot be managed pointers).
- If a property has both a getter and a setter, they must both be virtual, both static, or both instance. The C# and Visual Basic compilers automatically enforce this rule through their property definition syntax.

Events

An event is defined by its name and its type. The event type is a delegate that is used to indicate the event. For example, the AppDomain.AssemblyResolve event is of type ResolveEventHandler. In addition to the event itself, three methods with names based on the event name provide the event's implementation and are marked as `SpecialName` in the assembly's metadata:

- A method for adding an event handler, named **add**_*EventName*. For example, the event subscription method for the AppDomain.AssemblyResolve event is named **add_AssemblyResolve**.
- A method for removing an event handler, named **remove**_*EventName*. For example, the removal method for the AppDomain.AssemblyResolve event is named **remove_AssemblyResolve**.
- A method for indicating that the event has occurred, named **raise**_*EventName*.

> **Note**
>
> Most of the Common Language Specification's rules regarding events are implemented by language compilers and are transparent to component developers.

The methods for adding, removing, and raising the event must have the same accessibility. They must also all be static, instance, or virtual. The methods for adding and removing an event have one parameter whose type is the event delegate type. The add and remove methods must both be present or both be absent.

The following example defines a CLS-compliant class named `Temperature` that raises a `TemperatureChanged` event if the change in temperature between two readings equals or exceeds a threshold value. The `Temperature` class explicitly defines a `raise_TemperatureChanged` method so that it can selectively execute event handlers.

VB

```
Imports System.Collections
```

```
Imports System.Collections.Generic

<Assembly: CLSCompliant(True)>

Public Class TemperatureChangedEventArgs   : Inherits EventArgs
   Private originalTemp As Decimal
   Private newTemp As Decimal
   Private [when] As DateTimeOffset

   Public Sub New(original As Decimal, [new] As Decimal, [time] As DateTimeOffset)
      originalTemp = original
      newTemp = [new]
      [when] = [time]
   End Sub

   Public ReadOnly Property OldTemperature As Decimal
      Get
         Return originalTemp
      End Get
   End Property

   Public ReadOnly Property CurrentTemperature As Decimal
      Get
         Return newTemp
      End Get
   End Property

   Public ReadOnly Property [Time] As DateTimeOffset
      Get
         Return [when]
      End Get
   End Property
End Class

Public Delegate Sub TemperatureChanged(sender As Object, e As
TemperatureChangedEventArgs)

Public Class Temperature
   Private Structure TemperatureInfo
      Dim Temperature As Decimal
      Dim Recorded As DateTimeOffset
   End Structure

   Public Event TemperatureChanged As TemperatureChanged

   Private previous As Decimal
   Private current As Decimal
   Private tolerance As Decimal
   Private tis As New List(Of TemperatureInfo)

   Public Sub New(temperature As Decimal, tolerance As Decimal)
      current = temperature
      Dim ti As New TemperatureInfo()
      ti.Temperature = temperature
```

```
      ti.Recorded = DateTimeOffset.UtcNow
      tis.Add(ti)
      Me.tolerance = tolerance
   End Sub

   Public Property CurrentTemperature As Decimal
      Get
         Return current
      End Get
      Set
         Dim ti As New TemperatureInfo
         ti.Temperature = value
         ti.Recorded = DateTimeOffset.UtcNow
         previous = current
         current = value
         If Math.Abs(current - previous) >= tolerance Then
            raise_TemperatureChanged(New TemperatureChangedEventArgs(previous,
current, ti.Recorded))
         End If
      End Set
   End Property

   Public Sub raise_TemperatureChanged(eventArgs As TemperatureChangedEventArgs)
      If TemperatureChangedEvent Is Nothing Then Exit Sub

      Dim ListenerList() As System.Delegate =
TemperatureChangedEvent.GetInvocationList()
      For Each d As TemperatureChanged In TemperatureChangedEvent.GetInvocationList()
         If d.Method.Name.Contains("Duplicate") Then
            Console.WriteLine("Duplicate event handler; event handler not executed.")
         Else
            d.Invoke(Me, eventArgs)
         End If
      Next
   End Sub
End Class

Public Class Example
   Public WithEvents temp As Temperature

   Public Shared Sub Main()
      Dim ex As New Example()
   End Sub

   Public Sub New()
      temp = New Temperature(65, 3)
      RecordTemperatures()
      Dim ex As New Example(temp)
      ex.RecordTemperatures()
   End Sub

   Public Sub New(t As Temperature)
      temp = t
      RecordTemperatures()
```

```
    End Sub

    Public Sub RecordTemperatures()
        temp.CurrentTemperature = 66
        temp.CurrentTemperature = 63

    End Sub

    Friend Shared Sub TemperatureNotification(sender As Object, e As
TemperatureChangedEventArgs) _
            Handles temp.TemperatureChanged
        Console.WriteLine("Notification 1: The temperature changed from {0} to {1}",
e.OldTemperature, e.CurrentTemperature)
    End Sub

    Friend Shared Sub DuplicateTemperatureNotification(sender As Object, e As
TemperatureChangedEventArgs) _
            Handles temp.TemperatureChanged
        Console.WriteLine("Notification 2: The temperature changed from {0} to {1}",
e.OldTemperature, e.CurrentTemperature)
    End Sub
End Class
```

Overloads

The Common Language Specification imposes the following requirements on overloaded members:

- Members can be overloaded based on the number of parameters and the type of any parameter. Calling convention, return type, custom modifiers applied to the method or its parameter, and whether parameters are passed by value or by reference are not considered when differentiating between overloads. For an example, see the code for the requirement that names must be unique within a scope in the Naming conventions section.
- Only properties and methods can be overloaded. Fields and events cannot be overloaded.
- Generic methods can be overloaded based on the number of their generic parameters.

Note

The **op_Explicit** and **op_Implicit** operators are exceptions to the rule that return value is not considered part of a method signature for overload resolution. These two operators can be overloaded based on both their parameters and their return value.

Exceptions

Exception objects must derive from System.Exception or from another type derived from System.Exception. The following example illustrates the compiler error that results when a custom class named `ErrorClass` is used for

exception handling.

VB

```
Imports System.Runtime.CompilerServices

<Assembly: CLSCompliant(True)>

Public Class ErrorClass
   Dim msg As String

   Public Sub New(errorMessage As String)
      msg = errorMessage
   End Sub

   Public ReadOnly Property Message As String
      Get
         Return msg
      End Get
   End Property
End Class

Public Module StringUtilities
   <Extension()> Public Function SplitString(value As String, index As Integer) As
String()
      If index < 0 Or index > value.Length Then
         Dim BadIndex As New ErrorClass("The index is not within the string.")
         Throw BadIndex
      End If
      Dim retVal() As String = { value.Substring(0, index - 1),
                                 value.Substring(index) }
      Return retVal
   End Function
End Module
' Compilation produces a compiler error like the following:
'    Exceptions1.vb(27) : error BC30665: 'Throw' operand must derive from
'System.Exception'.
'
'             Throw BadIndex
'             ~~~~~~~~~~~~~~
```

To correct this error, the `ErrorClass` class must inherit from System.Exception. In addition, the `Message` property must be overridden. The following example corrects these errors to define an `ErrorClass` class that is CLS-compliant.

VB

```
Imports System.Runtime.CompilerServices

<Assembly: CLSCompliant(True)>

Public Class ErrorClass : Inherits Exception
   Dim msg As String
```

```
   Public Sub New(errorMessage As String)
      msg = errorMessage
   End Sub

   Public Overrides ReadOnly Property Message As String
      Get
         Return msg
      End Get
   End Property
End Class

Public Module StringUtilities
   <Extension()> Public Function SplitString(value As String, index As Integer) As
String()
      If index < 0 Or index > value.Length Then
         Dim BadIndex As New ErrorClass("The index is not within the string.")
         Throw BadIndex
      End If
      Dim retVal() As String = { value.Substring(0, index - 1),
                                 value.Substring(index) }
      Return retVal
   End Function
End Module
```

Attributes

In.NET Framework assemblies, custom attributes provide an extensible mechanism for storing custom attributes and retrieving metadata about programming objects, such as assemblies, types, members, and method parameters. Custom attributes must derive from System.Attribute or from a type derived from System.Attribute.

The following example violates this rule. It defines a `NumericAttribute` class that does not derive from System.Attribute. Note that a compiler error results only when the non-CLS-compliant attribute is applied, not when the class is defined.

VB

```
<Assembly: CLSCompliant(True)>

<AttributeUsageAttribute(AttributeTargets.Class Or AttributeTargets.Struct)> _
Public Class NumericAttribute
   Private _isNumeric As Boolean

   Public Sub New(isNumeric As Boolean)
      _isNumeric = isNumeric
   End Sub

   Public ReadOnly Property IsNumeric As Boolean
      Get
         Return _isNumeric
      End Get
   End Property
```

```
End Class

<Numeric(True)> Public Structure UDouble
   Dim Value As Double
End Structure
' Compilation produces a compiler error like the following:
'    error BC31504: 'NumericAttribute' cannot be used as an attribute because it
'    does not inherit from 'System.Attribute'.
'
'    <Numeric(True)> Public Structure UDouble
'     ~~~~~~~~~~~~~~
```

The constructor or the properties of a CLS-compliant attribute can expose only the following types:

- Boolean
- Byte
- Char
- Double
- Int16
- Int32
- Int64
- Single
- String
- Type
- Any enumeration type whose underlying type is Byte, Int16, Int32, or Int64.

The following example defines a `DescriptionAttribute` class that derives from Attribute. The class constructor has a parameter of type `Descriptor`, so the class is not CLS-compliant. Note that the C# compiler emits a warning but compiles successfully, whereas the Visual Basic compiler emits neither a warning nor an error.

VB

```
<Assembly:CLSCompliantAttribute(True)>

Public Enum DescriptorType As Integer
   Type = 0
   Member = 1
End Enum

Public Class Descriptor
   Public Type As DescriptorType
   Public Description As String
End Class
```

```
<AttributeUsage(AttributeTargets.All)> _
Public Class DescriptionAttribute : Inherits Attribute
   Private desc As Descriptor

   Public Sub New(d As Descriptor)
      desc = d
   End Sub

   Public ReadOnly Property Descriptor As Descriptor
      Get
         Return desc
      End Get
   End Property
End Class
```

The CLSCompliantAttribute attribute

The CLSCompliantAttribute attribute is used to indicate whether a program element complies with the Common Language Specification. The CLSCompliantAttribute.CLSCompliantAttribute(Boolean) constructor includes a single required parameter, *isCompliant*, that indicates whether the program element is CLS-compliant.

At compile time, the compiler detects non-compliant elements that are presumed to be CLS-compliant and emits a warning. The compiler does not emit warnings for types or members that are explicitly declared to be non-compliant.

Component developers can use the CLSCompliantAttribute attribute in two ways:

- To define the parts of the public interface exposed by a component that are CLS-compliant and the parts that are not CLS-compliant. When the attribute is used to mark particular program elements as CLS-compliant, its use guarantees that those elements are accessible from all languages and tools that target the .NET Framework.
- To ensure that the component library's public interface exposes only program elements that are CLS-compliant. If elements are not CLS-compliant, compilers will generally issue a warning.

Warning

In some cases, language compilers enforce CLS-compliant rules regardless of whether the CLSCompliantAttribute attribute is used. For example, defining a static member in an interface violates a CLS rule. However, if you define a **static** (in C#) or **Shared** (in Visual Basic) member in an interface, both the C# and Visual Basic compilers display an error message and fail to compile the app.

The CLSCompliantAttribute attribute is marked with an AttributeUsageAttribute attribute that has a value of AttributeTargets.All. This value allows you to apply the CLSCompliantAttribute attribute to any program element, including assemblies, modules, types (classes, structures, enumerations, interfaces, and delegates), type members

(constructors, methods, properties, fields, and events), parameters, generic parameters, and return values. However, in practice, you should apply the attribute only to assemblies, types, and type members. Otherwise, compilers ignore the attribute and continue to generate compiler warnings whenever they encounter a non-compliant parameter, generic parameter, or return value in your library's public interface.

The value of the CLSCompliantAttribute attribute is inherited by contained program elements. For example, if an assembly is marked as CLS-compliant, its types are also CLS-compliant. If a type is marked as CLS-compliant, its nested types and members are also CLS-compliant.

You can explicitly override the inherited compliance by applying the CLSCompliantAttribute attribute to a contained program element. For example, you can use the CLSCompliantAttribute attribute with an *isCompliant* value of **false** to define a non-compliant type in a compliant assembly, and you can use the attribute with an *isCompliant* value of **true** to define a compliant type in a non-compliant assembly. You can also define non-compliant members in a compliant type. However, a non-compliant type cannot have compliant members, so you cannot use the attribute with an *isCompliant* value of **true** to override inheritance from a non-compliant type.

When you are developing components, you should always use the CLSCompliantAttribute attribute to indicate whether your assembly, its types, and its members are CLS-compliant.

To create CLS-compliant components:

1. Use the CLSCompliantAttribute to mark you assembly as CLS-compliant.
2. Mark any publicly exposed types in the assembly that are not CLS-compliant as non-compliant.
3. Mark any publicly exposed members in CLS-compliant types as non-compliant.
4. Provide a CLS-compliant alternative for non-CLS-compliant members.

If you've successfully marked all your non-compliant types and members, your compiler should not emit any non-compliance warnings. However, you should indicate which members are not CLS-compliant and list their CLS-compliant alternatives in your product documentation.

The following example uses the CLSCompliantAttribute attribute to define a CLS-compliant assembly and a type, `CharacterUtilities`, that has two non-CLS-compliant members. Because both members are tagged with the **CLSCompliant(false)** attribute, the compiler produces no warnings. The class also provides a CLS-compliant alternative for both methods. Ordinarily, we would just add two overloads to the `ToUTF16` method to provide CLS-compliant alternatives. However, because methods cannot be overloaded based on return value, the names of the CLS-compliant methods are different from the names of the non-compliant methods.

VB

```
Imports System.Text

<Assembly:CLSCompliant(True)>

Public Class CharacterUtilities
   <CLSCompliant(False)> Public Shared Function ToUTF16(s As String) As UShort
      s = s.Normalize(NormalizationForm.FormC)
      Return Convert.ToUInt16(s(0))
   End Function

   <CLSCompliant(False)> Public Shared Function ToUTF16(ch As Char) As UShort
```

```
        Return Convert.ToUInt16(ch)
    End Function

    ' CLS-compliant alternative for ToUTF16(String).
    Public Shared Function ToUTF16CodeUnit(s As String) As Integer
        s = s.Normalize(NormalizationForm.FormC)
        Return CInt(Convert.ToInt16(s(0)))
    End Function

    ' CLS-compliant alternative for ToUTF16(Char).
    Public Shared Function ToUTF16CodeUnit(ch As Char) As Integer
        Return Convert.ToInt32(ch)
    End Function

    Public Function HasMultipleRepresentations(s As String) As Boolean
        Dim s1 As String = s.Normalize(NormalizationForm.FormC)
        Return s.Equals(s1)
    End Function

    Public Function GetUnicodeCodePoint(ch As Char) As Integer
        If Char.IsSurrogate(ch) Then
            Throw New ArgumentException("ch cannot be a high or low surrogate.")
        End If
        Return Char.ConvertToUtf32(ch.ToString(), 0)
    End Function

    Public Function GetUnicodeCodePoint(chars() As Char) As Integer
        If chars.Length > 2 Then
            Throw New ArgumentException("The array has too many characters.")
        End If
        If chars.Length = 2 Then
            If Not Char.IsSurrogatePair(chars(0), chars(1)) Then
                Throw New ArgumentException("The array must contain a low and a high
surrogate.")
            Else
                Return Char.ConvertToUtf32(chars(0), chars(1))
            End If
        Else
            Return Char.ConvertToUtf32(chars.ToString(), 0)
        End If
    End Function
End Class
```

If you are developing an app rather than a library (that is, if you aren't exposing types or members that can be consumed by other app developers), the CLS compliance of the program elements that your app consumes are of interest only if your language does not support them. In that case, your language compiler will generate an error when you try to use a non-CLS-compliant element.

Cross-Language Interoperability

Language independence has a number of possible meanings. One meaning, which is discussed in the article Language

Independence and Language-Independent Components, involves seamlessly consuming types written in one language from an app written in another language. A second meaning, which is the focus of this article, involves combining code written in multiple languages into a single .NET Framework assembly.

The following example illustrates cross-language interoperability by creating a class library named Utilities.dll that includes two classes, `NumericLib` and `StringLib`. The `NumericLib` class is written in C#, and the `StringLib` class is written in Visual Basic. Here's the source code for StringUtil.vb, which includes a single member, `ToTitleCase`, in its `StringLib` class.

VB

```
Imports System.Collections.Generic
Imports System.Runtime.CompilerServices

Public Module StringLib
   Private exclusions As List(Of String)

   Sub New()
      Dim words() As String = { "a", "an", "and", "of", "the" }
      exclusions = New List(Of String)
      exclusions.AddRange(words)
   End Sub

   <Extension()> _
   Public Function ToTitleCase(title As String) As String
      Dim words() As String = title.Split()
      Dim result As String = String.Empty

      For ctr As Integer = 0 To words.Length - 1
         Dim word As String = words(ctr)
         If ctr = 0 OrElse Not exclusions.Contains(word.ToLower()) Then
            result += word.Substring(0, 1).ToUpper() + _
                      word.Substring(1).ToLower()
         Else
            result += word.ToLower()
         End If
         If ctr <= words.Length - 1 Then
            result += " "
         End If
      Next
      Return result
   End Function
End Module
```

Here's the source code for NumberUtil.cs, which defines a `NumericLib` class that has two members, `IsEven` and `NearZero`.

C#

```
using System;

public static class NumericLib
{
```

```
    public static bool IsEven(this IConvertible number)
    {
       if (number is Byte ||
           number is SByte ||
           number is Int16 ||
           number is UInt16 ||
           number is Int32 ||
           number is UInt32 ||
           number is Int64)
          return ((long) number) % 2 == 0;
       else if (number is UInt64)
          return ((ulong) number) %2 == 0;
       else
          throw new NotSupportedException("IsEven called for a non-integer value.");
    }

    public static bool NearZero(double number)
    {
       return number < .00001;
    }
}
```

To package the two classes in a single assembly, you must compile them into modules. To compile the Visual Basic source code file into a module, use this command:

```
vbc /t:module StringUtil.vb
```

For more information about the command-line syntax of the Visual Basic compiler, see Building from the Command Line (Visual Basic).

To compile the C# source code file into a module, use this command:

```
csc /t:module NumberUtil.cs
```

For more information about the command-line syntax of the C# compiler, see Command-line Building With csc.exe.

You then use the Link tool (Link.exe) to compile the two modules into an assembly:

```
link numberutil.netmodule stringutil.netmodule /out:UtilityLib.dll /dll
```

The following example then calls the `NumericLib.NearZero` and `StringLib.ToTitleCase` methods. Note that both the Visual Basic code and the C# code are able to access the methods in both classes.

VB

```
Module Example
```

```
    Public Sub Main()
       Dim dbl As Double = 0.0 - Double.Epsilon
       Console.WriteLine(NumericLib.NearZero(dbl))

       Dim s As String = "war and peace"
       Console.WriteLine(s.ToTitleCase())
    End Sub
End Module
' The example displays the following output:
'       True
'       War and Peace
```

To compile the Visual Basic code, use this command:

```
vbc example.vb /r:UtilityLib.dll
```

To compile with C#, change the name of the compiler from **vbc** to **csc**, and change the file extension from .vb to .cs:

```
csc example.cs /r:UtilityLib.dll
```

See Also

CLSCompliantAttribute