**Cross-Language Interoperability**

This page is specific to

**Microsoft Visual Studio 2008/.NET Framework 3.5**

Other versions are also available for the following:

[Microsoft Visual Studio 2003/.NET Framework 1.1](http://msdn.microsoft.com/en-us/library/730f1wy3(VS.71).aspx)

[Microsoft Visual Studio 2005/.NET Framework 2.0](http://msdn.microsoft.com/en-us/library/730f1wy3(VS.80).aspx)

[.NET Framework 3.0](http://msdn.microsoft.com/en-us/library/730f1wy3(VS.85).aspx)

[Microsoft Visual Studio 2010/.NET Framework 4](http://msdn.microsoft.com/en-us/library/730f1wy3(VS.100).aspx)

The common language runtime provides built-in support for language interoperability. However, this support does not guarantee that code you write can be used by developers using another programming language. To ensure that you can develop managed code that can be fully used by developers using any programming language, a set of language features and rules for using them, called the [Common Language Specification](http://msdn.microsoft.com/en-us/library/12a7a7h3.aspx) (CLS), has been defined. Components that follow these rules and expose only CLS features are considered CLS-compliant.

This section describes the common language runtime's built-in support for language interoperability and explains the role that the CLS plays in enabling guaranteed cross-language interoperability. CLS features and rules are identified, and CLS compliance is discussed.

In This Section

[Language Interoperability Overview](http://msdn.microsoft.com/en-us/library/a2c7tshk.aspx)

Describes built-in support for cross-language interoperability and introduces the Common Language Specification.

[Common Language Specification](http://msdn.microsoft.com/en-us/library/12a7a7h3.aspx)

Explains the need for a set of features common to all languages and identifies CLS rules and features.

[Writing CLS-Compliant Code](http://msdn.microsoft.com/en-us/library/bhc3fa7f.aspx)

Discusses the meaning of CLS compliance for components and identifies levels of CLS compliance for tools.

**Language Interoperability Overview**

Language interoperability is the ability of code to interact with code that is written using a different programming language. Language interoperability can help maximize code reuse and, therefore, improve the efficiency of the development process.

Because developers use a wide variety of tools and technologies, each of which might support different features and types, it has historically been difficult to ensure language interoperability. However, language compilers and tools that target the common language runtime benefit from the runtime's built-in support for language interoperability.

The common language runtime provides the necessary foundation for language interoperability by specifying and enforcing a common type system and by providing metadata. Because all languages targeting the runtime follow the [common type system](http://msdn.microsoft.com/en-us/library/zcx1eb1e.aspx) rules for defining and using types, the usage of types is consistent across languages. [Metadata](http://msdn.microsoft.com/en-us/library/xcd8txaw.aspx) enables language interoperability by defining a uniform mechanism for storing and retrieving information about types. Compilers store type information as metadata, and the common language runtime uses this information to provide services during execution; the runtime can manage the execution of multilanguage applications because all type information is stored and retrieved in the same way, regardless of the language the code was written in.

Managed code benefits from the runtime's support for language interoperability in the following ways:

* Types can inherit implementation from other types, pass objects to another type's methods, and call methods defined on other types, regardless of the language the types are implemented in.
* Debuggers, profilers, or other tools are required to understand only one environment—the Microsoft intermediate language (MSIL) and metadata for the common language runtime—and they can support any programming language that targets the runtime.
* Exception handling is consistent across languages. Your code can throw an exception in one language and that exception can be caught and understood by an object written in another language.

Even though the runtime provides all managed code with support for executing in a multilanguage environment, there is no guarantee that the functionality of the types you create can be fully used by the programming languages that other developers use. This is primarily because each language compiler targeting the runtime uses the type system and metadata to support its own unique set of language features. In cases where you do not know what language the calling code will be written in, you are unlikely to know whether the features your component exposes are accessible to the caller. For example, if your language of choice provides support for unsigned integers, you might design a method with a parameter of type **UInt32**; but from a language that has no notion of unsigned integers, that method would be unusable.

To ensure that your managed code is accessible to developers using any programming language, the .NET Framework provides the [Common Language Specification](http://msdn.microsoft.com/en-us/library/12a7a7h3.aspx) (CLS), which describes a fundamental set of language features and defines rules for how those features are used. For more information about CLS compliance in components and tools, see [Writing CLS-Compliant Code](http://msdn.microsoft.com/en-us/library/bhc3fa7f.aspx).

**Common Language Specification**

To fully interact with other objects regardless of the language they were implemented in, objects must expose to callers only those features that are common to all the languages they must interoperate with. For this reason, the Common Language Specification (CLS), which is a set of basic language features needed by many applications, has been defined. The CLS rules define a subset of the [Common Type System](http://msdn.microsoft.com/en-us/library/zcx1eb1e.aspx); that is, all the rules that apply to the common type system apply to the CLS, except where stricter rules are defined in the CLS. The CLS helps enhance and ensure language interoperability by defining a set of features that developers can rely on to be available in a wide variety of languages. The CLS also establishes requirements for CLS compliance; these help you determine whether your managed code conforms to the CLS and to what extent a given tool supports the development of managed code that uses CLS features.

If your component uses only CLS features in the API that it exposes to other code (including derived classes), the component is guaranteed to be accessible from any programming language that supports the CLS. Components that adhere to the CLS rules and use only the features included in the CLS are said to be CLS-compliant components.

Most of the members defined by types in the [.NET Framework Class Library](http://msdn.microsoft.com/en-us/library/ms229335.aspx) are CLS-compliant. However, some types in the class library have one or more members that are not CLS-compliant. These members enable support for language features that are not in the CLS. The types and members that are not CLS-compliant are identified as such in the reference documentation, and in all cases a CLS-compliant alternative is available. For more information about the types in the .NET Framework class library, see the [.NET Framework Class Library](http://msdn.microsoft.com/en-us/library/ms229335.aspx).

The CLS was designed to be large enough to include the language constructs that are commonly needed by developers, yet small enough that most languages are able to support it. In addition, any language construct that makes it impossible to rapidly verify the type safety of code was excluded from the CLS so that all CLS-compliant languages can produce verifiable code if they choose to do so. For more information about verification of type safety, see [Compiling MSIL to Native Code](http://msdn.microsoft.com/en-us/library/ht8ecch6.aspx).

The following table summarizes the features that are in the CLS and indicates whether the feature applies to both developers and compilers (All) or only compilers. It is intended to be informative, but not comprehensive. For details, see the specification for the Common Language Infrastructure, Partition I, which is available on the [Microsoft Developer Network (MSDN)](http://go.microsoft.com/fwlink/?LinkId=99212) Web site.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Applies to** | **Description** |
| **General** |  |  |
| Visibility | All | CLS rules apply only to those parts of a type that are exposed outside the defining assembly. |
| Global members | All | Global **static** fields and methods are not CLS-compliant. |
| **Naming** |  |  |
| Characters and casing | All | CLS-compliant language compilers must follow the rules of Annex 7 of Technical Report 15 of the Unicode Standard 3.0, which governs the set of characters that can start and be included in identifiers. This standard is available from the Web site of the [Unicode Consortium](http://go.microsoft.com/fwlink/?LinkId=99211).  For two identifiers to be considered distinct, they must differ by more than just their case. |
| Keywords | Compilers | CLS-compliant language compilers supply a mechanism for referencing identifiers that coincide with keywords. CLS-compliant language compilers provide a mechanism for defining and overriding virtual methods with names that are keywords in the language. |
| Uniqueness | All | All names within a CLS-compliant scope must be distinct, even if the names are for two different kinds of members, except where the names are identical and resolved through overloading. For example, the CLS does not allow a single type to use the same name for a method and a field. |
| Signatures | All | All return and parameter types appearing in a type or member signature must be CLS-compliant. |
| **Types** |  |  |
| Primitive types | All | The .NET Framework class library includes types that correspond to the primitive data types that compilers use. Of these types, the following are CLS-compliant: [Byte](http://msdn.microsoft.com/en-us/library/system.byte.aspx), [Int16](http://msdn.microsoft.com/en-us/library/system.int16.aspx), [Int32](http://msdn.microsoft.com/en-us/library/system.int32.aspx), [Int64](http://msdn.microsoft.com/en-us/library/system.int64.aspx), [Single](http://msdn.microsoft.com/en-us/library/system.single.aspx), [Double](http://msdn.microsoft.com/en-us/library/system.double.aspx), [Boolean](http://msdn.microsoft.com/en-us/library/system.boolean.aspx), [Char](http://msdn.microsoft.com/en-us/library/system.char.aspx), [Decimal](http://msdn.microsoft.com/en-us/library/system.decimal.aspx), [IntPtr](http://msdn.microsoft.com/en-us/library/system.intptr.aspx), and [String](http://msdn.microsoft.com/en-us/library/system.string.aspx). For more information about these types, see the table of types in [.NET Framework Class Library Overview](http://msdn.microsoft.com/en-us/library/hfa3fa08.aspx). |
| Boxed types | All | Boxed value types (value types that have been converted to objects) are not part of the CLS. Instead, use [System..::.Object](http://msdn.microsoft.com/en-us/library/system.object.aspx), [System..::.ValueType](http://msdn.microsoft.com/en-us/library/system.valuetype.aspx), or [System..::.Enum](http://msdn.microsoft.com/en-us/library/system.enum.aspx), as appropriate. |
| Visibility | All | Type and member declarations must not contain types that are less visible or accessible than the type or member being declared. |
| Interface methods | Compilers | CLS-compliant language compilers must have syntax for the situation where a single type implements two interfaces and each of those interfaces requires the definition of a method with the same name and signature. Such methods must be considered distinct and need not have the same implementation. |
| Closure | All | The individual members of CLS-compliant interfaces and abstract classes must be defined to be CLS-compliant. |
| Constructor invocation | All | Before it accesses any inherited instance data, a constructor must call the base class's constructor. |
| Typed references | All | Typed references are not CLS-compliant. (A typed reference is a special construct that contains a reference to an object and a reference to a type. Typed references enable the common language runtime to provide C++-style support for methods that have a variable number of arguments.) |
| **Type Members** |  |  |
| Overloading | All | Indexed properties, methods, and constructors are allowed to be overloaded; fields and events must not be overloaded.  Properties must not be overloaded by type (that is, by the return type of their getter method), but they are allowed to be overloaded with different numbers or types of indexes.  Methods are allowed to be overloaded only based on the number and types of their parameters, and in the case of generic methods, the number of their generic parameters.  Operator overloading is not in the CLS. However, the CLS provides guidelines about providing useful names (such as Add()) and setting a bit in metadata. Compilers that choose to support operator overloading should follow these guidelines but are not required to do so. |
| Uniqueness of overloaded members | All | Fields and nested types must be distinct by identifier comparison alone. Methods, properties, and events that have the same name (by identifier comparison) must differ by more than just the return type. |
| Conversion operators | All | If either **op\_Implicit** or **op\_Explicit** is overloaded on its return type, an alternate means of providing the conversion must be provided. |
| **Methods** |  |  |
| Accessibility of overridden methods | All | Accessibility must not be changed when overriding inherited methods, except when overriding a method inherited from a different assembly with **FamilyOrAssembly** accessibility. In this case, the override must have **Family** accessibility. |
| Argument lists | All | The only calling convention supported by the CLS is the standard managed calling convention; variable length argument lists are not allowed. (Use the **ParamArray** keyword in Microsoft Visual Basic and the **params** keyword in C# for variable number of arguments support.) |
| **Properties** |  |  |
| Accessor metadata | Compilers | The getter and setter methods that implement the methods of a property are marked with the **mdSpecialName** identifier in the metadata. |
| Modifiers | All | The property and its accessors must all be **static**, all be **virtual**, or all be **instance**. |
| Accessor names | All | Properties must follow specific naming patterns. For a property called **Name**, the getter method, if defined, will be called **get\_Name** and the setter method, if defined, will be called **set\_Name**. |
| Return type and arguments | All | The type of the property is the return type of the getter and the type of the last argument of the setter. The types of the parameters of the property are the types of the parameters to the getter and the types of all but the final parameter of the setter. All these types must be CLS-compliant and cannot be managed pointers; they must not be passed by reference. |
| **Events** |  |  |
| Event methods | All | The methods for adding and removing an event must both be present or absent. |
| Event method metadata | Compilers | The methods that implement an event must be marked with the **mdSpecialName** identifier in the metadata. |
| Accessor accessibility | All | The accessibility of the methods for adding, removing, and raising an event must be identical. |
| Modifiers | All | The methods for adding, removing, and raising an event must all be **static**, all be **virtual**, or all be **instance**. |
| Event method names | All | Events must follow specific naming patterns. For an event named **MyEvent**, the add method, if defined, will be named **add\_MyEvent**, the remove method, if defined, will be named **remove\_MyEvent**, and the raise method will be named **raise\_MyEvent**. |
| Arguments | All | The methods for adding and removing an event must each take one parameter whose type defines the type of the event, and that type must be derived from [System..::.Delegate](http://msdn.microsoft.com/en-us/library/system.delegate.aspx). |
| **Pointer Types** |  |  |
| Pointers | All | Pointer types and function pointer types are not CLS-compliant. |
| **Interfaces** |  |  |
| Member signatures | All | CLS-compliant interfaces must not require the definition of non-CLS-compliant methods in order to implement them. |
| Member modifiers | All | CLS-compliant interfaces cannot define static methods, nor can they define fields. They are allowed to define properties, events, and virtual methods. |
| **Reference Types** |  |  |
| Constructor invocation | All | For reference types, object constructors are only called as part of the creation of an object, and objects are only initialized once. |
| **Class Types** |  |  |
| Inheritance | All | A CLS-compliant class must inherit from a CLS-compliant class ([System..::.Object](http://msdn.microsoft.com/en-us/library/system.object.aspx) is CLS-compliant). |
| **Arrays**1 |  |  |
| Element types | All | Array elements must be CLS-compliant types. |
| Dimensions | All | Arrays must have a fixed number of dimensions, greater than zero. |
| Bounds | All | All dimensions of an array must have a zero lower bound. |
| **Enumerations** |  |  |
| Underlying type | All | The underlying type of an enumeration must be a built-in CLS integer type ([Byte](http://msdn.microsoft.com/en-us/library/system.byte.aspx), [Int16](http://msdn.microsoft.com/en-us/library/system.int16.aspx), [Int32](http://msdn.microsoft.com/en-us/library/system.int32.aspx), or [Int64](http://msdn.microsoft.com/en-us/library/system.int64.aspx)). |
| **FlagsAttribute** | Compilers | The presence of the [System..::.FlagsAttribute](http://msdn.microsoft.com/en-us/library/system.flagsattribute.aspx) custom attribute on the definition of an enumeration indicates that the enumeration should be treated as a set of bit fields (flags), and the absence of this attribute indicates the type should be viewed as a group of enumerated constants. It is recommended that languages use either the **FlagsAttribute** or language-specific syntax for distinguishing between these two types of enumerations. |
| Field members | All | Literal **static** fields of an enumeration must be the same type as the type of the enumeration itself. |
| **Exceptions** |  |  |
| Inheritance | All | Objects that are thrown must be of type [System..::.Exception](http://msdn.microsoft.com/en-us/library/system.exception.aspx) or inherit from **System.Exception**. |
| **Custom Attributes** |  |  |
| Value encodings | Compilers | CLS-compliant compilers are required to deal with only a subset of the encodings of custom attributes (the representation of custom attributes in metadata). The only types that are permitted to appear in these encodings are: [System..::.Type](http://msdn.microsoft.com/en-us/library/system.type.aspx), [System..::.String](http://msdn.microsoft.com/en-us/library/system.string.aspx), [System..::.Char](http://msdn.microsoft.com/en-us/library/system.char.aspx), [System..::.Boolean](http://msdn.microsoft.com/en-us/library/system.boolean.aspx), [System..::.Byte](http://msdn.microsoft.com/en-us/library/system.byte.aspx), [System..::.Int16](http://msdn.microsoft.com/en-us/library/system.int16.aspx), [System..::.Int32](http://msdn.microsoft.com/en-us/library/system.int32.aspx), [System..::.Int64](http://msdn.microsoft.com/en-us/library/system.int64.aspx), [System..::.Single](http://msdn.microsoft.com/en-us/library/system.single.aspx), [System..::.Double](http://msdn.microsoft.com/en-us/library/system.double.aspx), and any enumeration type based on a CLS-compliant base integer type. |
| **Metadata** |  |  |
| CLS compliance | All | Types whose CLS compliance differs from that of the assembly in which they are defined must be so marked with the [System..::.CLSCompliantAttribute](http://msdn.microsoft.com/en-us/library/system.clscompliantattribute.aspx). Similarly, members whose CLS compliance differs from that of their type must also be marked. If a member or type is marked as not CLS-compliant, a CLS-compliant alternative must be provided. |
| **Generics** |  |  |
| Type names | Compilers | The name of a generic type must encode the number of type parameters declared on the type. The name of a nested generic type must encode the number of type parameters newly introduced to the type. |
| Nested types | Compilers | Nested types must 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. |
| Constraints | All | A generic type must declare sufficient constraints to guarantee that any constraints on the base type or interfaces are satisfied by the generic type constraints. |
| Constraint types | All | Types used as constraints on generic parameters must themselves be CLS-compliant. |
| Member signatures | All | The visibility and accessibility of members (including nested types) in an instantiated generic type is considered to be scoped to the specific instantiation rather than the generic type declaration as a whole. |
| Generic methods | All | For each abstract or virtual generic method, there must be a default concrete (non-abstract) implementation |

1. Jagged arrays — that is, arrays of arrays — are CLS-compliant. In the .NET Framework version 1.0, the C# compiler mistakenly reports that they are not.

**Writing CLS-Compliant Code**

Common Language Specification (CLS) compliance generally refers to the claim that CLS rules and restrictions are being followed. However, the concept has a more specific meaning depending on whether you are describing CLS-compliant code or CLS-compliant development tools, such as a compiler. CLS-compliant tools can help you write CLS-compliant code.

 CLS-Compliant Code

If you want your code to be CLS-compliant, you must expose functionality in a way that is CLS-compliant in the following places:

* Definitions of your public classes.
* Definitions of the public members of public classes, and of members accessible to derived classes (**family** access).
* Parameters and return types of public methods of public classes, and of methods accessible to derived classes.

The features you use in the definitions of your private classes, in the definitions of private methods on public classes, and in local variables do not have to follow the CLS rules. You can also use any language features you want in the code that implements your class and still have a CLS-compliant component.

|  |
| --- |
| **NoteNote:** |
| Jagged arrays — that is, arrays of arrays — are CLS-compliant. In the .NET Framework version 1.0, the C# compiler mistakenly reports that they are not. |

You can mark assemblies, modules, types, and members as either CLS-compliant or not CLS compliant using the [CLSCompliantAttribute](http://msdn.microsoft.com/en-us/library/system.clscompliantattribute.aspx). All assemblies that are intended to be CLS-compliant should be marked as such. An assembly that is not marked as CLS-compliant is considered to be not CLS compliant. If no CLS attribute is applied to a type, that type is assumed to have the same CLS compliance as the assembly in which the type is defined. Similarly, if no CLS attribute is applied to a member, the member is considered to have the same CLS compliance as the type that defines it. You cannot mark a program element as CLS-compliant if its enclosing element is not marked as CLS-compliant. The example at the end of this topic illustrates the use of the [CLSCompliantAttribute](http://msdn.microsoft.com/en-us/library/system.clscompliantattribute.aspx).

Assemblies, modules, and types can be CLS-compliant even if some parts of the assembly, module, or type are not CLS-compliant, as long as two conditions are met:

* If the element is marked as CLS-compliant, the parts that are not CLS-compliant must be marked using the [CLSCompliantAttribute](http://msdn.microsoft.com/en-us/library/system.clscompliantattribute.aspx) with its argument set to **false**.
* A comparable CLS-compliant alternative member must be supplied for each member that is not CLS-compliant.

If you design a CLS–compliant class library, your library will have a guarantee of interoperability with a wide range of programming languages; therefore, your library is likely to have a wider customer base than a version that is not CLS-compliant.

The .NET Framework provides a CLS-compliant class library. For more information about this class library, see [.NET Framework Class Library](http://msdn.microsoft.com/en-us/library/ms229335.aspx).

 CLS-Compliant Tools

Languages that target the runtime have agreed to support CLS features and follow the CLS rules directed to compilers. These language compilers simplify CLS compliance by making the CLS data types and features available for creating components. The levels of CLS compliance among compilers and other tools are described as follows:

* CLS-compliant consumer tools.

Consumer tools are languages that enable developers to access all the features supplied by CLS-compliant libraries. Developers using these languages might not be able to extend CLS-compliant libraries by creating new types, but they can use any type defined by a compliant library. This level of compliance can be useful when you want to access a .NET Framework class library, but do not need to author new objects for consumption by others, such as when you are using Web Forms on an ASP.NET page or creating a Windows Forms user interface.

* CLS-compliant extender tools.

Extender tools are languages that allow developers to both use and extend types defined in CLS-compliant libraries. Developers can use existing types as well as define new types. Extender tools must follow all the rules that consumer tools must follow, as well as some additional rules, which are described in the specification for the Common Language Infrastructure, Partition I - Architecture, available from the [Microsoft Developer Network (MSDN)](http://go.microsoft.com/fwlink/?LinkId=99212) Web site.

When you design your own CLS-compliant components, it is helpful to use a CLS-compliant tool. Writing CLS-compliant components without this support is more difficult because otherwise you might not have access to all the CLS features you want to use.

Some CLS-compliant language compilers, such as the C# or Visual Basic compilers, enable you to specify that you intend your code to be CLS-compliant. These compilers can check for CLS compliance and let you know when your code uses functionality that is not supported by the CLS. The C# and Visual Basic compilers allow you to mark a program element as CLS-compliant, which will cause the compiler to generate a compile-time error if the code is not CLS-compliant. For example, the following code generates a compiler warning.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl19_ctl00_ctl01_code');" \o "Copy Code)

<Assembly: CLSCompliant(True)>

<CLSCompliant(True)> Public Class MyCompliantClass

Public Sub ChangeValue(value As UInt32)

End Sub

Public Shared Sub Main()

Dim i As Integer = 2

Console.WriteLine(i)

End Sub

End Class

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl19_ctl00_ctl02_code');" \o "Copy Code)

using System;

// Assembly marked as compliant.

[assembly: CLSCompliant(true)]

// Class marked as compliant.

[CLSCompliant(true)]

public class MyCompliantClass {

// ChangeValue exposes UInt32, which is not in CLS.

// A compile-time warning results.

public void ChangeValue(UInt32 value){ }

public static void Main( ) {

int i = 2;

Console.WriteLine(i);

}

}

This code generates the following C# warning:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl19_ctl00_ctl03_code');" \o "Copy Code)

warning CS3001: Argument type 'uint' is not CLS-compliant

or the following Visual Basic warning:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl19_ctl00_ctl04_code');" \o "Copy Code)

warning BC40028: Type of parameter 'value' is not CLS-compliant.

To remove the warning, you can indicate that ChangeValue is not compliant, as shown in the following example.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl19_ctl00_ctl05_code');" \o "Copy Code)

' Assembly marked as compliant.

<Assembly: CLSCompliant(True)>

' Class marked as compliant.

<CLSCompliant(True)> Public Class MyCompliantClass

' Method marked as not compliant.

<CLSCompliant(False)> Public Sub ChangeValue(value As UInt32)

End Sub

Public Shared Sub Main()

Dim i As Integer = 2

Console.WriteLine(i)

End Sub

End Class

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl19_ctl00_ctl06_code');" \o "Copy Code)

using System;

// Assembly marked as compliant.

[assembly: CLSCompliantAttribute(true)]

// Class marked as compliant.

[CLSCompliantAttribute(true)]

public class MyCompliantClass {

// Method marked as not compliant.

[CLSCompliantAttribute(false)]

public void ChangeValue(UInt32 value){ }

public static void Main( ) {

int i = 2;

Console.WriteLine(i);

}

}

This code produces no compiler warnings. The output is 2.

For more information about how to specify the CLS-compliance of your code, see the documentation for the language compiler you are using.