**Common Type System**

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/zcx1eb1e(VS.71).aspx)

[Microsoft Visual Studio 2005/.NET Framework 2.0](http://msdn.microsoft.com/en-us/library/zcx1eb1e(VS.80).aspx)

[.NET Framework 3.0](http://msdn.microsoft.com/en-us/library/zcx1eb1e(VS.85).aspx)

[Microsoft Visual Studio 2010/.NET Framework 4](http://msdn.microsoft.com/en-us/library/zcx1eb1e(VS.100).aspx)

The common type system defines how types are declared, used, and managed in the runtime, and is also an important part of the runtime's support for cross-language integration. The common type system performs the following functions:

* Establishes a framework that helps enable cross-language integration, type safety, and high performance code execution.
* Provides an object-oriented model that supports the complete implementation of many programming languages.
* Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.

**Common Type System Overview**

This section describes concepts and defines terms that will help you understand and work with your language's implementation of the common type system.

http://i.msdn.microsoft.com/Global/Images/clear.gif Classification of Types

The common type system supports two general categories of types, each of which is further divided into subcategories:

* Value types

Value types directly contain their data, and instances of value types are either allocated on the stack or allocated inline in a structure. Value types can be built-in (implemented by the runtime), user-defined, or enumerations. For a list of built-in value types, see the [.NET Framework Class Library](http://msdn.microsoft.com/en-us/library/hfa3fa08.aspx).

* Reference types

Reference types store a reference to the value's memory address, and are allocated on the heap. Reference types can be self-describing types, pointer types, or interface types. The type of a reference type can be determined from values of self-describing types. Self-describing types are further split into arrays and class types. The class types are user-defined classes, boxed value types, and [delegates](http://msdn.microsoft.com/en-us/library/96b1ayy4.aspx).

Variables that are value types each have their own copy of the data, and therefore operations on one variable do not affect other variables. Variables that are reference types can refer to the same object; therefore, operations on one variable can affect the same object referred to by another variable.

All types derive from the [System..::.Object](http://msdn.microsoft.com/en-us/library/system.object.aspx) base type.

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

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl34_ctl00_ctl03_code');" \o "Copy Code)

Class Class1

Public Value As Integer = 0

End Class 'Class1

Class Test

Shared Sub Main()

Dim val1 As Integer = 0

Dim val2 As Integer = val1

val2 = 123

Dim ref1 As New Class1()

Dim ref2 As Class1 = ref1

ref2.Value = 123

Console.WriteLine("Values: {0}, {1}", val1, val2)

Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value)

End Sub 'Main

End Class 'Test

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl34_ctl00_ctl04_code');" \o "Copy Code)

using System;

class Class1

{

public int Value = 0;

}

class Test

{

static void Main() {

int val1 = 0;

int val2 = val1;

val2 = 123;

Class1 ref1 = new Class1();

Class1 ref2 = ref1;

ref2.Value = 123;

Console.WriteLine("Values: {0}, {1}", val1, val2);

Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);

}

}

The output from this program is as follows.

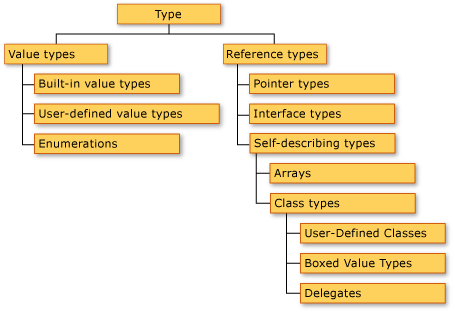
[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl34_ctl00_ctl05_code');" \o "Copy Code)

Values: 0, 123

Refs: 123, 123

The following diagram illustrates how these various types are related. Note that instances of types can be simply value types or self-describing types, even though there are subcategories of these types.

**Type classification**



For more information about each type, see [value types](http://msdn.microsoft.com/en-us/library/34yytbws.aspx), [enumerations](http://msdn.microsoft.com/en-us/library/6e3t9w21.aspx), [classes](http://msdn.microsoft.com/en-us/library/2s9w552e.aspx), [delegates](http://msdn.microsoft.com/en-us/library/96b1ayy4.aspx), [arrays](http://msdn.microsoft.com/en-us/library/45b76eb2.aspx), [interfaces](http://msdn.microsoft.com/en-us/library/f9846y7x.aspx), and [pointers](http://msdn.microsoft.com/en-us/library/cxx6f46y.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif Values and Objects

Values are binary representations of data, and types provide a way of interpreting this data. A value type is stored directly as a binary representation of the type's data. The value of a reference type is the location of the sequence of bits that represent the type's data.

Every value has an exact type that completely defines the value's representation and the operations that are defined on the value. Values of self-describing types are called objects. While it is always possible to determine the exact type of an object by examining its value, you cannot do so with a value type or pointer type. A value can have more than one type. A value of a type that implements an interface is also a value of that interface type. Likewise, a value of a type that derives from a base type is also a value of that base type.

http://i.msdn.microsoft.com/Global/Images/clear.gif Types and Assemblies

The runtime uses assemblies to locate and load types. The assembly manifest contains the information that the runtime uses to resolve all type references made within the scope of the assembly.

A type name in the runtime has two logical parts: the assembly name and the name of the type within the assembly. Two types with the same name but in different assemblies are defined as two distinct types.

Assemblies provide consistency between the scope of names seen by the developer and the scope of names seen by the runtime system. Developers author types in the context of an assembly. The content of the assembly a developer is building establishes the scope of names that will be available at run time.

http://i.msdn.microsoft.com/Global/Images/clear.gif Types and Namespaces

From the viewpoint of the runtime, a namespace is just a collection of type names. Particular languages might have constructs and corresponding syntax that help developers form logical groups of types, but these constructs are not used by the runtime when binding types. Thus, both the **Object** and **String** classes are part of the **System** namespace, but the runtime only recognizes the full names of each type, which are [System..::.Object](http://msdn.microsoft.com/en-us/library/system.object.aspx) and [System..::.String](http://msdn.microsoft.com/en-us/library/system.string.aspx), respectively.

You can build a single assembly that exposes types that look like they come from two different hierarchical namespaces, such as [System.Collections](http://msdn.microsoft.com/en-us/library/system.collections.aspx) and [System.Windows.Forms](http://msdn.microsoft.com/en-us/library/system.windows.forms.aspx). You can also build two assemblies that both export types whose names contain MyDll.MyClass.

If you create a tool to represent types in an assembly as belonging to a hierarchical namespace, the tool must enumerate the types in an assembly or group of assemblies and parse the type names to derive a hierarchical relationship.