Common Type System

.NET Framework (current version)

The common type system defines how types are declared, used, and managed in the common language 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.
- Provides a library that contains the primitive data types (such as Boolean, Byte, Char, Int32, and UInt64) used in application development.

This topic contains the following sections:

Types in the .NET Framework

All types in the .NET Framework are either value types or reference types.

Value types are data types whose objects are represented by the object's actual value. If an instance of a value type is assigned to a variable, that variable is given a fresh copy of the value.

Reference types are data types whose objects are represented by a reference (similar to a pointer) to the object's actual value. If a reference type is assigned to a variable, that variable references (points to) the original value. No copy is made.

The common type system in the .NET Framework supports the following five categories of types:

- Classes
- Structures
- Enumerations
- Interfaces
- Delegates

Classes

A class is a reference type that can be derived directly from another class and that is derived implicitly from System.Object. The class defines the operations that an object (which is an instance of the class) can perform (methods, events, or properties) and the data that the object contains (fields). Although a class generally includes both definition and implementation (unlike interfaces, for example, which contain only definition without implementation), it can have one or more members that have no implementation.

The following table describes some of the characteristics that a class may have. Each language that supports the runtime provides a way to indicate that a class or class member has one or more of these characteristics. However, individual programming languages that target the .NET Framework may not make all these characteristics available.

Characteristic	**Description**
sealed	Specifies that another class cannot be derived from this type.
implements	Indicates that the class uses one or more interfaces by providing implementations of interface members.
abstract	Indicates that the class cannot be instantiated. To use it, you must derive another class from it.
inherits	Indicates that instances of the class can be used anywhere the base class is specified. A derived class that inherits from a base class can use the implementation of any public members provided by the base class, or the derived class can override the implementation of the public members with its own implementation.
exported or not exported	Indicates whether a class is visible outside the assembly in which it is defined. This characteristic applies only to top-level classes and not to nested classes.

Note

A class can also be nested in a parent class or structure. Nested classes also have member characteristics. For more information, see Nested Types.

Class members that have no implementation are abstract members. A class that has one or more abstract members is itself abstract; new instances of it cannot be created. Some languages that target the runtime let you mark a class as abstract even if none of its members are abstract. You can use an abstract class when you want to encapsulate a basic set of functionality that derived classes can inherit or override when appropriate. Classes that are not abstract are referred to as concrete classes.

A class can implement any number of interfaces, but it can inherit from only one base class in addition to System.Object, from which all classes inherit implicitly. All classes must have at least one constructor, which initializes new instances of the class. If you do not explicitly define a constructor, most compilers will automatically provide a default (parameterless) constructor.

Structures

A structure is a value type that derives implicitly from System.ValueType, which in turn is derived from System.Object. A structure is very useful for representing values whose memory requirements are small, and for passing values as by-value parameters to methods that have strongly typed parameters. In the .NET Framework class library, all primitive data types (Boolean, Byte, Char, DateTime, Decimal, Double, Int16, Int32, Int64, SByte, Single, UInt16, UInt32, and UInt64) are defined as structures.

Like classes, structures define both data (the fields of the structure) and the operations that can be performed on that data (the methods of the structure). This means that you can call methods on structures, including the virtual methods defined on the System.Object and System.ValueType classes, and any methods defined on the value type itself. In other words, structures can have fields, properties, and events, as well as static and nonstatic methods. You can create instances of structures, pass them as parameters, store them as local variables, or store them in a field of another value type or reference type. Structures can also implement interfaces.

Value types also differ from classes in several respects. First, although they implicitly inherit from System.ValueType, they cannot directly inherit from any type. Similarly, all value types are sealed, which means that no other type can be derived from them. They also do not require constructors.

For each value type, the common language runtime supplies a corresponding boxed type, which is a class that has the same state and behavior as the value type. An instance of a value type is boxed when it is passed to a method that accepts a parameter of type System.Object. It is unboxed (that is, converted from an instance of a class back to an instance of a value type) when control returns from a method call that accepts a value type as a by-reference parameter. Some languages require that you use special syntax when the boxed type is required; others automatically use the boxed type when it is needed. When you define a value type, you are defining both the boxed and the unboxed type.

Enumerations

An enumeration (enum) is a value type that inherits directly from System.Enum and that supplies alternate names for the values of an underlying primitive type. An enumeration type has a name, an underlying type that must be one of the built-in signed or unsigned integer types (such as Byte, Int32, or UInt64), and a set of fields. The fields are static literal fields, each of which represents a constant. The same value can be assigned to multiple fields. When this occurs, you must mark one of the values as the primary enumeration value for reflection and string conversion.

You can assign a value of the underlying type to an enumeration and vice versa (no cast is required by the runtime). You can create an instance of an enumeration and call the methods of System.Enum, as well as any methods defined on the enumeration's underlying type. However, some languages might not let you pass an enumeration as a parameter when an instance of the underlying type is required (or vice versa).

The following additional restrictions apply to enumerations:

- They cannot define their own methods.
- They cannot implement interfaces.
- They cannot define properties or events.
- They cannot be generic, unless they are generic only because they are nested within a generic type. That is, an enumeration cannot have type parameters of its own.

Note

Nested types (including enumerations) created with Visual Basic, C#, and C++ include the type parameters of all enclosing generic types, and are therefore generic even if they do not have type parameters of their own. For more information, see "Nested Types" in the Type.MakeGenericType reference topic.

The FlagsAttribute attribute denotes a special kind of enumeration called a bit field. The runtime itself does not distinguish between traditional enumerations and bit fields, but your language might do so. When this distinction is made, bitwise operators can be used on bit fields, but not on enumerations, to generate unnamed values. Enumerations are generally used for lists of unique elements, such as days of the week, country or region names, and so on. Bit fields are generally used for lists of qualities or quantities that might occur in combination, such as `Red And Big And Fast`.

The following example shows how to use both bit fields and traditional enumerations.

C#

```
using System;
using System.Collections.Generic;

// A traditional enumeration of some root vegetables.
public enum SomeRootVegetables
{
    HorseRadish,
    Radish,
    Turnip
}

// A bit field or flag enumeration of harvesting seasons.
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}

public class Example
{
   public static void Main()
   {
       // Hash table of when vegetables are available.
       Dictionary<SomeRootVegetables, Seasons> AvailableIn = new
Dictionary<SomeRootVegetables, Seasons>();

       AvailableIn[SomeRootVegetables.HorseRadish] = Seasons.All;
       AvailableIn[SomeRootVegetables.Radish] = Seasons.Spring;
       AvailableIn[SomeRootVegetables.Turnip] = Seasons.Spring |
            Seasons.Autumn;

       // Array of the seasons, using the enumeration.
       Seasons[] theSeasons = new Seasons[] { Seasons.Summer, Seasons.Autumn,
            Seasons.Winter, Seasons.Spring };

       // Print information of what vegetables are available each season.
       foreach (Seasons season in theSeasons)
       {
          Console.Write(String.Format(
              "The following root vegetables are harvested in {0}:\n",
              season.ToString("G")));
          foreach (KeyValuePair<SomeRootVegetables, Seasons> item in AvailableIn)
          {
             // A bitwise comparison.
             if (((Seasons)item.Value & season) > 0)
                 Console.Write(String.Format("  {0:G}\n",
                      (SomeRootVegetables)item.Key));
          }
       }
```

```
      }
   }
   // The example displays the following output:
   //    The following root vegetables are harvested in Summer:
   //      HorseRadish
   //    The following root vegetables are harvested in Autumn:
   //      Turnip
   //      HorseRadish
   //    The following root vegetables are harvested in Winter:
   //      HorseRadish
   //    The following root vegetables are harvested in Spring:
   //      Turnip
   //      Radish
   //      HorseRadish
```

Interfaces

An interface defines a contract that specifies a "can do" relationship or a "has a" relationship. Interfaces are often used to implement functionality, such as comparing and sorting (the IComparable and IComparable<T> interfaces), testing for equality (the IEquatable<T> interface), or enumerating items in a collection (the IEnumerable and IEnumerable<T> interfaces). Interfaces can have properties, methods, and events, all of which are abstract members; that is, although the interface defines the members and their signatures, it leaves it to the type that implements the interface to define the functionality of each interface member. This means that any class or structure that implements an interface must supply definitions for the abstract members declared in the interface. An interface can require any implementing class or structure to also implement one or more other interfaces.

The following restrictions apply to interfaces:

- An interface can be declared with any accessibility, but interface members must all have public accessibility.
- Interfaces cannot define constructors.
- Interfaces cannot define fields.
- Interfaces can define only instance members. They cannot define static members.

Each language must provide rules for mapping an implementation to the interface that requires the member, because more than one interface can declare a member with the same signature, and these members can have separate implementations.

Delegates

Delegates are reference types that serve a purpose similar to that of function pointers in C++. They are used for event handlers and callback functions in the .NET Framework. Unlike function pointers, delegates are secure, verifiable, and type safe. A delegate type can represent any instance method or static method that has a compatible signature.

A parameter of a delegate is compatible with the corresponding parameter of a method if the type of the delegate parameter is more restrictive than the type of the method parameter, because this guarantees that an argument passed to the delegate can be passed safely to the method.

Similarly, the return type of a delegate is compatible with the return type of a method if the return type of the method is more restrictive than the return type of the delegate, because this guarantees that the return value of the method can be cast safely to the return type of the delegate.

For example, a delegate that has a parameter of type IEnumerable and a return type of Object can represent a method that has a parameter of type Object and a return value of type IEnumerable. For more information and example code, see Delegate.CreateDelegate(Type, Object, MethodInfo).

A delegate is said to be bound to the method it represents. In addition to being bound to the method, a delegate can be bound to an object. The object represents the first parameter of the method, and is passed to the method every time the delegate is invoked. If the method is an instance method, the bound object is passed as the implicit **this** parameter (**Me** in Visual Basic); if the method is static, the object is passed as the first formal parameter of the method, and the delegate signature must match the remaining parameters. For more information and example code, see System.Delegate.

All delegates inherit from System.MulticastDelegate, which inherits from System.Delegate. The C#, Visual Basic, and C++ languages do not allow inheritance from these types. Instead, they provide keywords for declaring delegates.

Because delegates inherit from MulticastDelegate, a delegate has an invocation list, which is a list of methods that the delegate represents and that are executed when the delegate is invoked. All methods in the list receive the arguments supplied when the delegate is invoked.

Note

The return value is not defined for a delegate that has more than one method in its invocation list, even if the delegate has a return type.

In many cases, such as with callback methods, a delegate represents only one method, and the only actions you have to take are creating the delegate and invoking it.

For delegates that represent multiple methods, the .NET Framework provides methods of the Delegate and MulticastDelegate delegate classes to support operations such as adding a method to a delegate's invocation list (the Delegate.Combine method), removing a method (the Delegate.Remove method), and getting the invocation list (the Delegate.GetInvocationList method).

Note

It is not necessary to use these methods for event-handler delegates in C#, C++, and Visual Basic, because these languages provide syntax for adding and removing event handlers.

Back to top

Type Definitions

A type definition includes the following:

- Any attributes defined on the type.
- The type's accessibility (visibility).
- The type's name.
- The type's base type.
- Any interfaces implemented by the type.

- Definitions for each of the type's members.

Attributes

Attributes provide additional user-defined metadata. Most commonly, they are used to store additional information about a type in its assembly, or to modify the behavior of a type member in either the design-time or run-time environment.

Attributes are themselves classes that inherit from System.Attribute. Languages that support the use of attributes each have their own syntax for applying attributes to a language element. Attributes can be applied to almost any language element; the specific elements to which an attribute can be applied are defined by the AttributeUsageAttribute that is applied to that attribute class.

Type Accessibility

All types have a modifier that governs their accessibility from other types. The following table describes the type accessibilities supported by the runtime.

Accessibility	Description
public	The type is accessible by all assemblies.
assembly	The type is accessible only from within its assembly.

The accessibility of a nested type depends on its accessibility domain, which is determined by both the declared accessibility of the member and the accessibility domain of the immediately containing type. However, the accessibility domain of a nested type cannot exceed that of the containing type.

The accessibility domain of a nested member `M` declared in a type `T` within a program `P` is defined as follows (noting that `M` might itself be a type):

- If the declared accessibility of `M` is **public**, the accessibility domain of `M` is the accessibility domain of `T`.
- If the declared accessibility of `M` is **protected internal**, the accessibility domain of `M` is the intersection of the accessibility domain of `T` with the program text of `P` and the program text of any type derived from `T` declared outside `P`.
- If the declared accessibility of `M` is **protected**, the accessibility domain of `M` is the intersection of the accessibility domain of `T` with the program text of `T` and any type derived from `T`.
- If the declared accessibility of `M` is **internal**, the accessibility domain of `M` is the intersection of the accessibility domain of `T` with the program text of `P`.
- If the declared accessibility of `M` is **private**, the accessibility domain of `M` is the program text of `T`.

Type Names

The common type system imposes only two restrictions on names:

- All names are encoded as strings of Unicode (16-bit) characters.
- Names are not permitted to have an embedded (16-bit) value of 0x0000.

However, most languages impose additional restrictions on type names. All comparisons are done on a byte-by-byte basis, and are therefore case-sensitive and locale-independent.

Although a type might reference types from other modules and assemblies, a type must be fully defined within one .NET Framework module. (Depending on compiler support, however, it can be divided into multiple source code files.) Type names need be unique only within a namespace. To fully identify a type, the type name must be qualified by the namespace that contains the implementation of the type.

Base Types and Interfaces

A type can inherit values and behaviors from another type. The common type system does not allow types to inherit from more than one base type.

A type can implement any number of interfaces. To implement an interface, a type must implement all the virtual members of that interface. A virtual method can be implemented by a derived type and can be invoked either statically or dynamically.

Back to top

Type Members

The runtime enables you to define members of your type, which specifies the behavior and state of a type. Type members include the following:

- Fields
- Properties
- Methods
- Constructors
- Events
- Nested types

Fields

A field describes and contains part of the type's state. Fields can be of any type supported by the runtime. Most commonly, fields are either **private** or **protected**, so that they are accessible only from within the class or from a derived class. If the value of a field can be modified from outside its type, a property set accessor is typically used. Publicly exposed fields are usually read-only and can be of two types:

- Constants, whose value is assigned at design time. These are static members of a class, although they are not defined using the **static** (**Shared** in Visual Basic) keyword.
- Read-only variables, whose values can be assigned in the class constructor.

The following example illustrates these two usages of read-only fields.

C#

```
using System;

public class Constants
{
   public const double Pi = 3.1416;
   public readonly DateTime BirthDate;

   public Constants(DateTime birthDate)
   {
      this.BirthDate = birthDate;
   }
}

public class Example
{
   public static void Main()
   {
      Constants con = new Constants(new DateTime(1974, 8, 18));
      Console.Write(Constants.Pi + "\n");
      Console.Write(con.BirthDate.ToString("d") + "\n");
   }
}
// The example displays the following output if run on a system whose current
// culture is en-US:
//    3.1417
//    8/18/1974
```

Properties

A property names a value or state of the type and defines methods for getting or setting the property's value. Properties can be primitive types, collections of primitive types, user-defined types, or collections of user-defined types. Properties are often used to keep the public interface of a type independent from the type's actual representation. This enables properties to reflect values that are not directly stored in the class (for example, when a property returns a computed value) or to perform validation before values are assigned to private fields. The following example illustrates the latter pattern.

C#

```
using System;

public class Person
{
   private int m_Age;

   public int Age
   {
      get { return m_Age; }
      set {
         if (value < 0 || value > 125)
         {
            throw new ArgumentOutOfRangeException("The value of the Age property must be
between 0 and 125.");
         }
         else
```

```
            {
               m_Age = value;
            }
         }
      }
   }
```

In addition to including the property itself, the Microsoft intermediate language (MSIL) for a type that contains a readable property includes a **get_***propertyname* method, and the MSIL for a type that contains a writable property includes a **set_***propertyname* method.

Methods

A method describes operations that are available on the type. A method's signature specifies the allowable types of all its parameters and of its return value.

Although most methods define the precise number of parameters required for method calls, some methods support a variable number of parameters. The final declared parameter of these methods is marked with the ParamArrayAttribute attribute. Language compilers typically provide a keyword, such as **params** in C# and **ParamArray** in Visual Basic, that makes explicit use of ParamArrayAttribute unnecessary.

Constructors

A constructor is a special kind of method that creates new instances of a class or structure. Like any other method, a constructor can include parameters; however, constructors have no return value (that is, they return **void**).

If the source code for a class does not explicitly define a constructor, the compiler includes a default (parameterless) constructor. However, if the source code for a class defines only parameterized constructors, the Visual Basic and C# compilers do not generate a parameterless constructor.

If the source code for a structure defines constructors, they must be parameterized; a structure cannot define a default (parameterless) constructor, and compilers do not generate parameterless constructors for structures or other value types. All value types do have an implicit default constructor. This constructor is implemented by the common language runtime and initializes all fields of the structure to their default values.

Events

An event defines an incident that can be responded to, and defines methods for subscribing to, unsubscribing from, and raising the event. Events are often used to inform other types of state changes. For more information, see Handling and Raising Events.

Nested Types

A nested type is a type that is a member of some other type. Nested types should be tightly coupled to their containing type and must not be useful as a general-purpose type. Nested types are useful when the declaring type uses and creates instances of the nested type, and use of the nested type is not exposed in public members.

Nested types are confusing to some developers and should not be publicly visible unless there is a compelling reason for visibility. In a well-designed library, developers should rarely have to use nested types to instantiate objects or declare variables.

Back to top

Characteristics of Type Members

The common type system allows type members to have a variety of characteristics; however, languages are not required to support all these characteristics. The following table describes member characteristics.

Characteristic	Can apply to	Description
abstract	Methods, properties, and events	The type does not supply the method's implementation. Types that inherit or implement abstract methods must supply an implementation for the method. The only exception is when the derived type is itself an abstract type. All abstract methods are virtual.
private, family, assembly, family and assembly, family or assembly, or public	All	Defines the accessibility of the member: private Accessible only from within the same type as the member, or within a nested type. family Accessible from within the same type as the member, and from derived types that inherit from it. assembly Accessible only in the assembly in which the type is defined. family and assembly Accessible only from types that qualify for both family and assembly access. family or assembly Accessible only from types that qualify for either family or assembly access. public Accessible from any type.
final	Methods, properties, and events	The virtual method cannot be overridden in a derived type.
initialize-only	Fields	The value can only be initialized, and cannot be written after initialization.
instance	Fields, methods, properties, and events	If a member is not marked as **static** (C# and C++), **Shared** (Visual Basic), **virtual** (C# and C++), or **Overridable** (Visual Basic), it is an instance member (there is no instance keyword). There will be as many copies of such members in memory as there are objects that use it.
literal	Fields	The value assigned to the field is a fixed value, known at compile time, of a built-in value type. Literal fields are sometimes referred to as constants.
newslot or override	All	Defines how the member interacts with inherited members that have the same signature: newslot Hides inherited members that have the same signature. override Replaces the definition of an inherited virtual method.

		The default is newslot.
static	Fields, methods, properties, and events	The member belongs to the type it is defined on, not to a particular instance of the type; the member exists even if an instance of the type is not created, and it is shared among all instances of the type.
virtual	Methods, properties, and events	The method can be implemented by a derived type and can be invoked either statically or dynamically. If dynamic invocation is used, the type of the instance that makes the call at run time (rather than the type known at compile time) determines which implementation of the method is called. To invoke a virtual method statically, the variable might have to be cast to a type that uses the desired version of the method.

Overloading

Each type member has a unique signature. Method signatures consist of the method name and a parameter list (the order and types of the method's arguments). Multiple methods with the same name can be defined within a type as long as their signatures differ. When two or more methods with the same name are defined, the method is said to be overloaded. For example, in System.Char, the IsDigit method is overloaded. One method takes a Char. The other method takes a String and an Int32.

Note

The return type is not considered part of a method's signature. That is, methods cannot be overloaded if they differ only by return type.

Inheriting, Overriding, and Hiding Members

A derived type inherits all members of its base type; that is, these members are defined on, and available to, the derived type. The behavior or qualities of inherited members can be modified in two ways:

- A derived type can hide an inherited member by defining a new member with the same signature. This might be done to make a previously public member private or to define new behavior for an inherited method that is marked as **final**.
- A derived type can override an inherited virtual method. The overriding method provides a new definition of the method that will be invoked based on the type of the value at run time rather than the type of the variable known at compile time. A method can override a virtual method only if the virtual method is not marked as **final** and the new method is at least as accessible as the virtual method.

See Also

.NET Framework Class Library
Common Language Runtime (CLR)
Type Conversion in the .NET Framework