

# Common type system

Article • 01/03/2024

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.

## Types in .NET

All types in .NET 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 .NET 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 .NET may not make all these characteristics available.

[ ] [Expand table](#)

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 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 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 .NET, 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 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.WriteLine(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 .NET. 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, .NET 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.

## 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.

[Expand table](#)

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 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.

## 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.WriteLine(Constants.Pi + "\n");
        Console.WriteLine(con.BirthDate.ToString("d") + "\n");
    }
}

// The example displays the following output if run on a system whose
// current
// culture is en-US:
//      3.1416
//      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 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 parameterless constructor, and compilers do not generate parameterless constructors for structures or other value types. All value types do have an implicit parameterless 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 [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.

## 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.

Expand table

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

Characteristic	Can apply to	Description
		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	<p>Defines the accessibility of the member:</p> <p>private</p> <p>Accessible only from within the same type as the member, or within a nested type.</p> <p>family</p> <p>Accessible from within the same type as the member, and from derived types that inherit from it.</p> <p>assembly</p> <p>Accessible only in the assembly in which the type is defined.</p> <p>family and assembly</p> <p>Accessible only from types that qualify for both family and assembly access.</p> <p>family or assembly</p> <p>Accessible only from types that qualify for either family or assembly access.</p> <p>public</p> <p>Accessible from any type.</p>
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 <code>static</code> (C# and C++), <code>Shared</code> (Visual Basic), <code>virtual</code> (C# and C++), or <code>Overridable</code> (Visual Basic), it is an instance member (there is no <code>instance</code> 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	<p>Defines how the member interacts with inherited members that have the same signature:</p> <p>newslot</p>

Characteristic	Can apply to	Description
		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.

## Inherit, override, and hide 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 API Browser](#)
- [Common Language Runtime](#)
- [Type Conversion in .NET](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Language independence and language-independent components

Article • 12/21/2022

.NET is language independent. This means that, as a developer, you can develop in one of the many languages that target .NET implementations, such as C#, F#, and Visual Basic. You can access the types and members of class libraries developed for .NET implementations 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're a component developer, your component can be accessed by any .NET 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](#).

## 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's 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.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
    private UInt16 personAge = 0;

    public UInt16 Age
    { get { return personAge; } }

    // The attempt to compile the example displays the following compiler
    // warning:
    //    Public1.cs(10,18): warning CS3003: Type of 'Person.Age' is not CLS-
    //    compliant
```

You can make the `Person` class CLS-compliant by changing the type of the `Age` property from `UInt16` to `Int16`, which is a CLS-compliant, 16-bit signed integer. You don't have to

change the type of the private `personAge` field.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
    private Int16 personAge = 0;

    public Int16 Age
    { get { return personAge; } }
}
```

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.

[\[+\] Expand table](#)

Category	See	Rule	Rule Number
Accessibility	<a href="#">Member accessibility</a>	Accessibility shall not be changed when overriding inherited methods, except when overriding a method inherited from a different assembly with accessibility <code>family-or-assembly</code> . In this case, the override shall have accessibility <code>family</code> .	10
Accessibility	<a href="#">Member accessibility</a>	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	12

Category	See	Rule	Rule Number
		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.	
Arrays	<a href="#">Arrays</a>	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	<a href="#">Attributes</a>	Attributes shall be of type <a href="#">System.Attribute</a> , or a type inheriting from it.	41
Attributes	<a href="#">Attributes</a>	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): <a href="#">System.Type</a> , <a href="#">System.String</a> , <a href="#">System.Char</a> , <a href="#">System.Boolean</a> , <a href="#">System.Byte</a> , <a href="#">System.Int16</a> , <a href="#">System.Int32</a> , <a href="#">System.Int64</a> , <a href="#">System.Single</a> , <a href="#">System.Double</a> , and any enumeration type based on a CLS-compliant base integer type.	34
Attributes	<a href="#">Attributes</a>	The CLS does not allow publicly visible required modifiers ( <code>modreq</code> , see Partition II), but does allow optional modifiers ( <code>modopt</code> , see Partition II) it does not understand.	35
Constructors	<a href="#">Constructors</a>	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	<a href="#">Constructors</a>	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	<a href="#">Enumerations</a>	The underlying type of an enum shall be a built-in CLS integer type, the name of the field shall be	7

Category	See	Rule	Rule Number
Enumerations	<a href="#">Enumerations</a>	There are two distinct kinds of enums, indicated by the presence or absence of the <a href="#">System.FlagsAttribute</a> (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 <code>enum</code> is not limited to the specified values.	8
Enumerations	<a href="#">Enumerations</a>	Literal static fields of an enum shall have the type of the enum itself.	9
Events	<a href="#">Events</a>	The methods that implement an event shall be marked <code>SpecialName</code> in the metadata.	29
Events	<a href="#">Events</a>	The accessibility of an event and of its accessors shall be identical.	30
Events	<a href="#">Events</a>	The <code>add</code> and <code>remove</code> methods for an event shall both either be present or absent.	31
Events	<a href="#">Events</a>	The <code>add</code> and <code>remove</code> methods for an event shall each take one parameter whose type defines the type of the event and that shall be derived from <a href="#">System.Delegate</a> .	32
Events	<a href="#">Events</a>	Events shall adhere to a specific naming pattern. The <code>SpecialName</code> attribute referred to in CLS rule 29 shall be ignored in appropriate name comparisons and shall adhere to identifier rules.	33
Exceptions	<a href="#">Exceptions</a>	Objects that are thrown shall be of type <a href="#">System.Exception</a> or a type inheriting from it. Nonetheless, CLS-compliant methods are not required to block the propagation of other types of exceptions.	40
General	<a href="#">CLS compliance rules</a>	CLS rules apply only to those parts of a type that are accessible or visible outside of the defining assembly.	1
General	<a href="#">CLS compliance rules</a>	Members of non-CLS compliant types shall not be marked CLS-compliant.	2

Category	See	Rule	Rule Number
Generics	<a href="#">Generic types and members</a>	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	<a href="#">Generic types and members</a>	The name of a generic type shall encode the number of type parameters declared on the non-nested type, or newly introduced to the type if nested, according to the rules defined above.	43
Generics	<a href="#">Generic types and members</a>	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.	44
Generics	<a href="#">Generic types and members</a>	Types used as constraints on generic parameters shall themselves be CLS-compliant.	45
Generics	<a href="#">Generic types and members</a>	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	<a href="#">Generic types and members</a>	For each abstract or virtual generic method, there shall be a default concrete (nonabstract) implementation	47
Interfaces	<a href="#">Interfaces</a>	CLS-compliant interfaces shall not require the definition of non-CLS compliant methods in order to implement them.	18
Interfaces	<a href="#">Interfaces</a>	CLS-compliant interfaces shall not define static methods, nor shall they define fields.	19
Members	<a href="#">Type members in general</a>	Global static fields and methods are not CLS-compliant.	36
Members	--	The value of a literal static is specified by using 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 <code>enum</code> ).	13
Members	<a href="#">Type members in</a>	The vararg constraint is not part of the CLS, and the only calling convention supported by the CLS is the	15

Category	See	Rule	Rule Number
	general	standard managed calling convention.	
Naming conventions	<a href="#">Naming conventions</a>	Assemblies shall follow Annex 7 of Technical Report 15 of the Unicode Standard 3.0 governing the set of characters permitted to start and be included in identifiers, available online at <a href="#">Unicode Normalization Forms</a> . Identifiers shall be in the canonical format defined by Unicode Normalization Form C. For CLS purposes, two identifiers are the same if their lowercase mappings (as specified by the Unicode locale-insensitive, one-to-one lowercase mappings) are the same. That is, for two identifiers to be considered different under the CLS they shall differ in more than simply their case. However, in order to override an inherited definition the CLI requires the precise encoding of the original declaration be used.	4
Overloading	<a href="#">Naming conventions</a>	All names introduced in a CLS-compliant scope shall be distinct independent of kind, except where the names are identical and resolved via overloading. That is, while the CTS allows a single type to use the same name for a method and a field, the CLS does not.	5
Overloading	<a href="#">Naming conventions</a>	Fields and nested types shall be distinct by identifier comparison alone, even though the CTS allows distinct signatures to be distinguished. Methods, properties, and events that 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	<a href="#">Overloads</a>	Only properties and methods can be overloaded.	37
Overloading	<a href="#">Overloads</a>	Properties and methods can be overloaded based only on the number and types of their parameters, except the conversion operators named <code>op_Implicit</code> and <code>op_Explicit</code> , 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 name and, for a specific set of type instantiations, they have the same parameter and return types, then all these methods shall be semantically equivalent at those type instantiations.	48
Properties	<a href="#">Properties</a>	The methods that implement the getter and setter methods of a property shall be marked <code>SpecialName</code>	24

Category	See	Rule	Rule Number
		in the metadata.	
Properties	<a href="#">Properties</a>	A property's accessors shall all be static, all be virtual, or all be instance.	26
Properties	<a href="#">Properties</a>	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 (that is, shall not be passed by reference).	27
Properties	<a href="#">Properties</a>	Properties shall adhere to a specific naming pattern. The <code>SpecialName</code> 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	<a href="#">Type conversion</a>	If either <code>op_Implicit</code> or <code>op_Explicit</code> is provided, an alternate means of providing the coercion shall be provided.	39
Types	<a href="#">Types and type member signatures</a>	Boxed value types are not CLS-compliant.	3
Types	<a href="#">Types and type member signatures</a>	All types appearing in a signature shall be CLS-compliant. All types composing an instantiated generic type shall be CLS-compliant.	11
Types	<a href="#">Types and type member signatures</a>	Typed references are not CLS-compliant.	14
Types	<a href="#">Types and type member signatures</a>	Unmanaged pointer types are not CLS-compliant.	17
Types	<a href="#">Types and type member signatures</a>	CLS-compliant classes, value types, and interfaces shall not require the implementation of non-CLS-compliant members	20
Types	<a href="#">Types and type member signatures</a>	<code>System.Object</code> is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS-compliant class.	23

Index to subsections:

- [Types and type member signatures](#)
- [Naming conventions](#)
- [Type conversion](#)
- [Arrays](#)
- [Interfaces](#)
- [Enumerations](#)
- [Type members in general](#)
- [Member accessibility](#)
- [Generic types and members](#)
- [Constructors](#)
- [Properties](#)
- [Events](#)
- [Overloads](#)
- [Exceptions](#)
- [Attributes](#)

## Types and type-member signatures

The [System.Object](#) type is CLS-compliant and is the base type of all object types in the .NET type system. Inheritance in .NET 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 [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.

C#

```
using System;

[assembly: CLSCompliant(true)]

[CLSCompliant(false)]
public class Counter
{
    UInt32 ctr;
```

```

public Counter()
{
    ctr = 0;
}

protected Counter(UInt32 ctr)
{
    this.ctr = ctr;
}

public override string ToString()
{
    return String.Format("{0}). ", ctr);
}

public UInt32 Value
{
    get { return ctr; }
}

public void Increment()
{
    ctr += (uint) 1;
}

public class NonZeroCounter : Counter
{
    public NonZeroCounter(int startIndex) : this((uint) startIndex)
    {
    }

    private NonZeroCounter(UInt32 startIndex) : base(startIndex)
    {
    }
}

// Compilation produces a compiler warning like the following:
// Type3.cs(37,14): warning CS3009: 'NonZeroCounter': base type 'Counter'
// is not
//         CLS-compliant
// Type3.cs(7,14): (Location of symbol related to previous warning)

```

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 [common type system](#) includes many 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.

[Expand table](#)

CLS-compliant type	Description
Byte	8-bit unsigned integer
Int16	16-bit signed integer
Int32	32-bit signed integer
Int64	64-bit signed integer
Half	Half-precision floating-point value
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.

[Expand table](#)

Non-compliant type	Description	CLS-compliant alternative
SByte	8-bit signed integer data type	Int16
UInt16	16-bit unsigned integer	Int32
UInt32	32-bit unsigned integer	Int64
UInt64	64-bit unsigned integer	Int64 (may overflow), BigInteger, or Double
IntPtr	Unsigned pointer or handle	IntPtr

The .NET 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 .NET 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 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<UInt32>`, and a constructor with parameters of type `UInt32` and `Nullable<UInt32>`. You get four compiler warnings when you try to compile this example.

C#

```
using System;

[assembly: CLSCompliant(true)]
```

```

public class InvoiceItem
{
    private uint invId = 0;
    private uint itemId = 0;
    private Nullable<uint> qty;

    public InvoiceItem(uint sku, Nullable<uint> quantity)
    {
        itemId = sku;
        qty = quantity;
    }

    public Nullable<uint> Quantity
    {
        get { return qty; }
        set { qty = value; }
    }

    public uint InvoiceId
    {
        get { return invId; }
        set { invId = value; }
    }
}

// The attempt to compile the example displays the following output:
//  Type1.cs(13,23): warning CS3001: Argument type 'uint' is not CLS-
//  compliant
//  Type1.cs(13,33): warning CS3001: Argument type 'uint?' is not CLS-
//  compliant
//  Type1.cs(19,26): warning CS3003: Type of 'InvoiceItem.Quantity' is not
//  CLS-compliant
//  Type1.cs(25,16): warning CS3003: Type of 'InvoiceItem.InvoiceId' is
//  not CLS-compliant

```

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

C#

```

using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
    private uint invId = 0;
    private uint itemId = 0;
    private Nullable<int> qty;

    public InvoiceItem(int sku, Nullable<int> quantity)
    {
        if (sku <= 0)
            throw new ArgumentOutOfRangeException("The item number is zero or

```

```

    negative.");
    itemId = (uint) sku;

    qty = quantity;
}

public Nullable<int> Quantity
{
    get { return qty; }
    set { qty = value; }
}

public int InvoiceId
{
    get { return (int) invId; }
    set {
        if (value <= 0)
            throw new ArgumentOutOfRangeException("The invoice number is
zero or negative.");
        invId = (uint) value; }
}
}

```

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:
//     UnmanagedType1.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](#). 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 punctuation, 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.

C#

```
public class Size
{
    private double a1;
    private double a2;

    public double Å
    {
        get { return a1; }
        set { a1 = value; }
    }

    public double Å
    {
        get { return a2; }
        set { a2 = value; }
    }
}

// Compilation produces a compiler warning like the following:
//     Naming2a.cs(16,18): warning CS3005: Identifier 'Size.Å' differing only
//     in case is not
//             CLS-compliant
//     Naming2a.cs(10,18): (Location of symbol related to previous warning)
//     Naming2a.cs(18,8): warning CS3005: Identifier 'Size.Å.get' differing
//     only in case is not
//             CLS-compliant
//     Naming2a.cs(12,8): (Location of symbol related to previous warning)
//     Naming2a.cs(19,8): warning CS3005: Identifier 'Size.Å.set' differing
//     only in case is not
//             CLS-compliant
//     Naming2a.cs(13,8): (Location of symbol related to previous warning)
```

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.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Converter
{
    public double Conversion(int number)
    {
        return (double) number;
    }

    public float Conversion(int number)
    {
        return (float) number;
    }

    public double Conversion(long number)
    {
        return (double) number;
    }

    public bool Conversion
    {
        get { return true; }
    }
}

// Compilation produces a compiler error like the following:
//   Naming3.cs(13,17): error CS0111: Type 'Converter' already defines a
//   member called
//           'Conversion' with the same parameter types
//   Naming3.cs(8,18): (Location of symbol related to previous error)
//   Naming3.cs(23,16): error CS0102: The type 'Converter' already contains
//   a definition for
//           'Conversion'
//   Naming3.cs(8,18): (Location of symbol related to previous error)
```

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 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 `FromXxx` and `ToXxx` methods.

The following example defines CLS-compliant implicit and explicit conversions. It creates a `UDouble` class that represents an unsigned, 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.

C#

```
using System;

public struct UDouble
{
    private double number;

    public UDouble(double value)
    {
        if (value < 0)
            throw new InvalidCastException("A negative value cannot be
converted to a UDouble.");

        number = value;
    }

    public UDouble(float value)
    {
        if (value < 0)
            throw new InvalidCastException("A negative value cannot be
converted to a UDouble.");

        number = value;
    }
}
```

```
public static readonly UDouble MinValue = (UDouble) 0.0;
public static readonly UDouble MaxValue = (UDouble) Double.MaxValue;

public static explicit operator Double(UDouble value)
{
    return value.number;
}

public static implicit operator Single(UDouble value)
{
    if (value.number > (double) Single.MaxValue)
        throw new InvalidCastException("A UDouble value is out of range of
the Single type.");

    return (float) value.number;
}

public static explicit operator UDouble(double value)
{
    if (value < 0)
        throw new InvalidCastException("A negative value cannot be
converted to a UDouble.");

    return new UDouble(value);
}

public static implicit operator UDouble(float value)
{
    if (value < 0)
        throw new InvalidCastException("A negative value cannot be
converted to a UDouble.");

    return new UDouble(value);
}

public static Double ToDouble(UDouble value)
{
    return (Double) value;
}

public static float ToSingle(UDouble value)
{
    return (float) value;
}

public static UDouble FromDouble(double value)
{
    return new UDouble(value);
}

public static UDouble FromSingle(float value)
{
    return new UDouble(value);
}
```

## 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. 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.

C#

```
[assembly: CLSCompliant(true)]  
  
public class Numbers  
{  
    public static Array GetTenPrimes()  
    {  
        Array arr = Array.CreateInstance(typeof(Int32), new int[] {10},  
        new int[] {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;  
    }  
}
```

- 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.

C#

```
using System;  
  
[assembly: CLSCompliant(true)]
```

```

public class Numbers
{
    public static UInt32[] GetTenPrimes()
    {
        uint[] arr = { 1u, 2u, 3u, 5u, 7u, 11u, 13u, 17u, 19u };
        return arr;
    }

    public static Object[] GetFivePrimes()
    {
        Object[] arr = { 1, 2, 3, 5u, 7u };
        return arr;
    }
}

// Compilation produces a compiler warning like the following:
//     Array2.cs(8,27): warning CS3002: Return type of
'Numbers.GetTenPrimes()' is not
//                 CLS-compliant

```

- 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.

C#

```

using System;
using System.Numerics;

[assembly: CLSCompliant(true)]

public class Numbers
{
    public static byte[] GetSquares(byte[] numbers)
    {
        byte[] numbersOut = new byte[numbers.Length];
        for (int ctr = 0; ctr < numbers.Length; ctr++) {
            int square = ((int) numbers[ctr]) * ((int) numbers[ctr]);
            if (square <= Byte.MaxValue)
                numbersOut[ctr] = (byte) square;
            // If there's an overflow, assign MaxValue to the
            corresponding
            // element.
            else
                numbersOut[ctr] = Byte.MaxValue;
        }
        return numbersOut;
    }

    public static BigInteger[] GetSquares(BigInteger[] numbers)
    {
        BigInteger[] numbersOut = new BigInteger[numbers.Length];

```

```
        for (int ctr = 0; ctr < numbers.Length; ctr++)
            numbersOut[ctr] = numbers[ctr] * numbers[ctr];

        return numbersOut;
    }
}
```

## 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.

C#

```
using System;

[assembly:CLSCCompliant(true)]

public interface INumber
{
    int Length();
    [CLSCCompliant(false)] ulong GetUnsigned();
}

// Attempting to compile the example displays output like the
// following:
//  Interface2.cs(8,32): warning CS3010: 'INumber.GetUnsigned()':
//  CLS-compliant interfaces
//                      must have only CLS-compliant members
```

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.

C#

```
using System;

[assembly: CLSCompliant(true)]

public interface IFahrenheit
{
    decimal GetTemperature();
}

public interface ICelsius
{
    decimal GetTemperature();
}

public class Temperature : ICelsius, IFahrenheit
{
    private decimal _value;

    public Temperature(decimal value)
    {
        // We assume that this is the Celsius value.
        _value = value;
    }

    decimal IFahrenheit.GetTemperature()
    {
        return _value * 9 / 5 + 32;
    }

    decimal ICelsius.GetTemperature()
    {
        return _value;
    }
}

public class Example
{
    public static void Main()
    {
        Temperature temp = new Temperature(100.0m);
        ICelsius cTemp = temp;
        IFahrenheit fTemp = temp;
        Console.WriteLine("Temperature in Celsius: {0} degrees",
                          cTemp.GetTemperature());
```

```

        Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                           fTemp.GetTemperature());
    }
}

// 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.

C#

```

using System;

[assembly: CLSCompliant(true)]

public enum Size : uint {
    Unspecified = 0,
    XSmall = 1,
    Small = 2,
    Medium = 3,
    Large = 4,
    XLarge = 5
};

public class Clothing
{
    public string Name;
    public string Type;
    public string Size;
}

// The attempt to compile the example displays a compiler warning like
// the following:
//  Enum3.cs(6,13): warning CS3009: 'Size': base type 'uint' is not
//  CLS-compliant

```

- 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 `Human`, 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.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Animal
{
    private string _species;

    public Animal(string species)
    {
        _species = species;
    }

    public virtual string Species
    {
        get { return _species; }
    }

    public override string ToString()
    {
        return _species;
    }
}

public class Human : Animal
{
    private string _name;

    public Human(string name) : base("Homo Sapiens")
    {
        _name = name;
    }

    public string Name
    {
        get { return _name; }
    }

    private override string Species
    {
        get { return base.Species; }
    }

    public override string ToString()
    {
        return _name;
    }
}
```

```

    }

}

public class Example
{
    public static void Main()
    {
        Human p = new Human("John");
        Console.WriteLine(p.Species);
        Console.WriteLine(p.ToString());
    }
}

// The example displays the following output:
//      error CS0621: 'Human.Species': virtual or abstract members cannot be
private

```

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.

C#

```

using System;
using System.Text;

public class StringWrapper
{
    string internalString;
    StringBuilder internalSB = null;
    bool useSB = false;

    public StringWrapper(StringOperationType type)
    {
        if (type == StringOperationType.Normal) {
            useSB = false;
        }
        else {
            useSB = true;
            internalSB = new StringBuilder();
        }
    }

    // The remaining source code...
}

internal enum StringOperationType { Normal, Dynamic }
// The attempt to compile the example displays the following output:
//      error CS0051: Inconsistent accessibility: parameter type

```

```
//           'StringOperationType' is less accessible than method
//           'StringWrapper.StringWrapper(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.

C#

```
using System;

[assembly:CLSCCompliant(true)]

public class Outer<T>
{
    T value;

    public Outer(T value)
    {
        this.value = value;
    }

    public class Inner1A : Outer<T>
    {
        public Inner1A(T value) : base(value)
        { }
    }

    public class Inner1B<U> : Outer<T>
    {
        U value2;

        public Inner1B(T value1, U value2) : base(value1)
        {
            this.value2 = value2;
        }
    }
}

public class Example
```

```

public static void Main()
{
    var inst1 = new Outer<String>("This");
    Console.WriteLine(inst1);

    var inst2 = new Outer<String>.Inner1A("Another");
    Console.WriteLine(inst2);

    var inst3 = new Outer<String>.Inner1B<int>("That", 2);
    Console.WriteLine(inst3);
}

// 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-compliant 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.

```

C#

using System;

[assembly:CLSCCompliant(true)]

[CLSCCompliant(false)] public class BaseClass
{ }

public class BaseCollection<T> where T : BaseClass
{ }

// Attempting to compile the example displays the following output:
//      warning CS3024: Constraint type 'BaseClass' is not CLS-compliant

```

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>`.

C#

```
using System;

[assembly:CLSCCompliant(true)]

public class Number<T> where T : struct
{
    // use Double as the underlying type, since its range is a superset of
    // the ranges of all numeric types except BigInteger.
    protected double number;

    public Number(T value)
    {
        try {
            this.number = Convert.ToDouble(value);
        }
        catch (OverflowException e) {
            throw new ArgumentException("value is too large.", e);
        }
        catch (InvalidCastException e) {
            throw new ArgumentException("The value parameter is not numeric.", e);
        }
    }

    public T Add(T value)
    {
        return (T) Convert.ChangeType(number + Convert.ToDouble(value),
typeof(T));
    }

    public T Subtract(T value)
    {
        return (T) Convert.ChangeType(number - Convert.ToDouble(value),
typeof(T));
    }
}

public class FloatingPoint<T> : Number<T>
{
    public FloatingPoint(T number) : base(number)
    {
        if (typeof(float) == number.GetType() ||
            typeof(double) == number.GetType() ||
            typeof(decimal) == number.GetType())
            this.number = Convert.ToDouble(number);
        else
            throw new ArgumentException("The number parameter is not a
floating-point number.");
    }
}
```

```
    }
}

// The attempt to compile the example displays the following output:
//      error CS0453: The type 'T' must be a non-nullable value type in
//          order to use it as parameter 'T' in the generic type or
// method 'Number<T>'
```

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

C#

```
using System;

[assembly:CLSCCompliant(true)]

public class Number<T> where T : struct
{
    // use Double as the underlying type, since its range is a superset of
    // the ranges of all numeric types except BigInteger.
    protected double number;

    public Number(T value)
    {
        try {
            this.number = Convert.ToDouble(value);
        }
        catch (OverflowException e) {
            throw new ArgumentException("value is too large.", e);
        }
        catch (InvalidCastException e) {
            throw new ArgumentException("The value parameter is not numeric.", e);
        }
    }

    public T Add(T value)
    {
        return (T) Convert.ChangeType(number + Convert.ToDouble(value),
            typeof(T));
    }

    public T Subtract(T value)
    {
        return (T) Convert.ChangeType(number - Convert.ToDouble(value),
            typeof(T));
    }
}

public class FloatingPoint<T> : Number<T> where T : struct
{
    public FloatingPoint(T number) : base(number)
    {
```

```
    if (typeof(float) == number.GetType() ||
        typeof(double) == number.GetType() ||
        typeof(decimal) == number.GetType())
        this.number = Convert.ToDouble(number);
    else
        throw new ArgumentException("The number parameter is not a
floating-point number.");
}
```

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>`. Language compilers can be even more restrictive. In this example, Visual Basic displays an error when it tries to compile `M4`.

C#

```

C1<long>)

protected void M4(C1<long>.N n) { } // CLS-compliant, C1<long>.N is
                                    // accessible in C2 (extends
C1<long>
}
// Attempting to compile the example displays output like the following:
//      Generics4.cs(9,22): warning CS3001: Argument type 'C1<int>.N' is
not CLS-compliant
//      Generics4.cs(18,22): warning CS3001: Argument type 'C1<int>.N' is
not CLS-compliant

```

## 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 because 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.

```

C#

using System;

[assembly: CLSCompliant(true)]

public class Person
{
    private string fName, lName, _id;

    public Person(string firstName, string lastName, string id)
    {
        if (String.IsNullOrEmpty(firstName + lastName))
            throw new ArgumentNullException("Either a first name or a last
name must be provided.");

        fName = firstName;
        lName = lastName;
        _id = id;
    }

    public string FirstName
    {

```

```

        get { return fName; }

    }

    public string LastName
    {
        get { return lName; }
    }

    public string Id
    {
        get { return _id; }
    }

    public override string ToString()
    {
        return String.Format("{0}{1}{2}", fName,
            String.IsNullOrEmpty(fName) ? "" : " ",
            lName);
    }
}

public class Doctor : Person
{
    public Doctor(string firstName, string lastName, string id)
    {
    }

    public override string ToString()
    {
        return "Dr. " + base.ToString();
    }
}

// Attempting to compile the example displays output like the
// following:
//   ctor1.cs(45,11): error CS1729: 'Person' does not contain a
// constructor that takes 0
//           arguments
//   ctor1.cs(10,11): (Location of symbol related to previous error)

```

- 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 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`.

### ① 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.

C#

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

[assembly: CLSCompliant(true)]

public class TemperatureChangedEventArgs : EventArgs
{
    private Decimal originalTemp;
    private Decimal newTemp;
    private DateTimeOffset when;

    public TemperatureChangedEventArgs(Decimal original, Decimal @new,
DateTimeOffset time)
    {
        originalTemp = original;
        newTemp = @new;
        when = time;
    }

    public Decimal OldTemperature
    {
        get { return originalTemp; }
    }

    public Decimal CurrentTemperature
    {
        get { return newTemp; }
    }

    public DateTimeOffset Time
    {
        get { return when; }
    }
}

public delegate void TemperatureChanged(Object sender,
TemperatureChangedEventArgs e);

public class Temperature
{
    private struct TemperatureInfo
    {
        public Decimal Temperature;
        public DateTimeOffset Recorded;
    }
```

```
public event TemperatureChanged TemperatureChanged;

private Decimal previous;
private Decimal current;
private Decimal tolerance;
private List<TemperatureInfo> tis = new List<TemperatureInfo>();

public Temperature(Decimal temperature, Decimal tolerance)
{
    current = temperature;
    TemperatureInfo ti = new TemperatureInfo();
    ti.Temperature = temperature;
    tis.Add(ti);
    ti.Recorded = DateTimeOffset.UtcNow;
    this.tolerance = tolerance;
}

public Decimal CurrentTemperature
{
    get { return current; }
    set {
        TemperatureInfo ti = new TemperatureInfo();
        ti.Temperature = value;
        ti.Recorded = DateTimeOffset.UtcNow;
        previous = current;
        current = value;
        if (Math.Abs(current - previous) >= tolerance)
            raise_TemperatureChanged(new
TemperatureChangedEventArgs(previous, current, ti.Recorded));
    }
}

public void raise_TemperatureChanged(TemperatureChangedEventArgs
EventArgs)
{
    if (TemperatureChanged == null)
        return;

    foreach (TemperatureChanged d in
TemperatureChanged.GetInvocationList()) {
        if (d.Method.Name.Contains("Duplicate"))
            Console.WriteLine("Duplicate event handler; event handler not
executed.");
        else
            d.Invoke(this, EventArgs);
    }
}

public class Example
{
    public Temperature temp;

    public static void Main()
```

```

{
    Example ex = new Example();
}

public Example()
{
    temp = new Temperature(65, 3);
    temp.TemperatureChanged += this.TemperatureNotification;
    RecordTemperatures();
    Example ex = new Example(temp);
    ex.RecordTemperatures();
}

public Example(Temperature t)
{
    temp = t;
    RecordTemperatures();
}

public void RecordTemperatures()
{
    temp.TemperatureChanged += this.DuplicateTemperatureNotification;
    temp.CurrentTemperature = 66;
    temp.CurrentTemperature = 63;
}

internal void TemperatureNotification(Object sender,
TemperatureChangedEventArgs e)
{
    Console.WriteLine("Notification 1: The temperature changed from {0} to
{1}", e.OldTemperature, e.CurrentTemperature);
}

public void DuplicateTemperatureNotification(Object sender,
TemperatureChangedEventArgs e)
{
    Console.WriteLine("Notification 2: The temperature changed from {0} to
{1}", e.OldTemperature, e.CurrentTemperature);
}
}

```

## 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.

### ① 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.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass
{
    string msg;

    public ErrorClass(string errorMessage)
    {
        msg = errorMessage;
    }

    public string Message
    {
        get { return msg; }
    }
}

public static class StringUtils
{
    public static string[] SplitString(this string value, int index)
    {
```

```

        if (index < 0 | index > value.Length) {
            ErrorClass badIndex = new ErrorClass("The index is not within the
string.");
            throw badIndex;
        }
        string[] retVal = { value.Substring(0, index - 1),
                            value.Substring(index) };
        return retVal;
    }
}
// Compilation produces a compiler error like the following:
// Exceptions1.cs(26,16): error CS0155: The type caught or thrown must be
derived from
//           System.Exception

```

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.

C#

```

using System;

[assembly: CLSCompliant(true)]

public class ErrorClass : Exception
{
    string msg;

    public ErrorClass(string errorMessage)
    {
        msg = errorMessage;
    }

    public override string Message
    {
        get { return msg; }
    }
}

public static class StringUtils
{
    public static string[] SplitString(this string value, int index)
    {
        if (index < 0 | index > value.Length) {
            ErrorClass badIndex = new ErrorClass("The index is not within the
string.");
            throw badIndex;
        }
        string[] retVal = { value.Substring(0, index - 1),
                            value.Substring(index) };
        return retVal;
    }
}

```

```
    }  
}
```

## Attributes

In .NET 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](#). A compiler error results only when the non-CLS-compliant attribute is applied, not when the class is defined.

C#

```
using System;  
  
[assembly: CLSCompliant(true)]  
  
[AttributeUsageAttribute(AttributeTargets.Class | AttributeTargets.Struct)]  
public class NumericAttribute  
{  
    private bool _isNumeric;  
  
    public NumericAttribute(bool isNumeric)  
    {  
        _isNumeric = isNumeric;  
    }  
  
    public bool IsNumeric  
    {  
        get { return _isNumeric; }  
    }  
}  
  
[Numeric(true)] public struct UDouble  
{  
    double Value;  
}  
// Compilation produces a compiler error like the following:  
//     Attribute1.cs(22,2): error CS0616: 'NumericAttribute' is not an  
//     attribute class  
//     Attribute1.cs(7,14): (Location of symbol related to previous error)
```

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. The C# compiler emits a warning but compiles successfully, whereas the Visual Basic compiler doesn't emit a warning or an error.

C#

```
using System;

[assembly:CLSClaimedAttribute(true)]

public enum DescriptorType { type, member };

public class Descriptor
{
    public DescriptorType Type;
    public String Description;
}

[AttributeUsage(AttributeTargets.All)]
public class DescriptionAttribute : Attribute
{
    private Descriptor desc;

    public DescriptionAttribute(Descriptor d)
    {
        desc = d;
    }
}
```

```
public Descriptor Descriptor
{ get { return desc; } }
}
// Attempting to compile the example displays output like the following:
//     warning CS3015: 'DescriptionAttribute' has no accessible
//                     constructors which use only CLS-compliant types
```

## The `CLSCompliantAttribute` attribute

The `CLSCompliantAttribute` attribute is used to indicate whether a program element complies with the Common Language Specification. The `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 .NET.
- 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. In this regard, 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 your 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.

C#

```
using System;
using System.Text;

[assembly:CLSCompliant(true)]

public class CharacterUtilities
{
    [CLSCompliant(false)] public static ushort ToUTF16(String s)
    {
        s = s.Normalize(NormalizationForm.FormC);
        return Convert.ToInt16(s[0]);
    }

    [CLSCompliant(false)] public static ushort ToUTF16(Char ch)
    {
        return Convert.ToInt16(ch);
    }

    // CLS-compliant alternative for ToUTF16(String).
    public static int ToUTF16CodeUnit(String s)
    {
        s = s.Normalize(NormalizationForm.FormC);
        return (int) Convert.ToInt16(s[0]);
    }

    // CLS-compliant alternative for ToUTF16(Char).
    public static int ToUTF16CodeUnit(Char ch)
    {
        return Convert.ToInt32(ch);
    }

    public bool HasMultipleRepresentations(String s)
    {
        String s1 = s.Normalize(NormalizationForm.FormC);
        return s.Equals(s1);
    }

    public int GetUnicodeCodePoint(Char ch)
    {
        if (Char.IsSurrogate(ch))
            throw new ArgumentException("ch cannot be a high or low
surrogate.");

        return Char.ConvertToUtf32(ch.ToString(), 0);
    }

    public int GetUnicodeCodePoint(Char[] chars)
```

```

{
    if (chars.Length > 2)
        throw new ArgumentException("The array has too many characters.");

    if (chars.Length == 2) {
        if (! Char.IsSurrogatePair(chars[0], chars[1]))
            throw new ArgumentException("The array must contain a low and a
high surrogate.");
        else
            return Char.ConvertToUtf32(chars[0], chars[1]);
    }
    else {
        return Char.ConvertToUtf32(chars.ToString(), 0);
    }
}

```

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 few possible meanings. One meaning 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 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 Convert.ToInt64(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 Math.Abs(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:

Console

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

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

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

Console

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

You then use the [Linker options](#) to compile the two modules into an assembly:

Console

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

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

C#

```
using System;  
  
public class Example  
{  
    public static void Main()  
    {  
        Double dbl = 0.0 - Double.Epsilon;  
        Console.WriteLine(NumericLib.NearZero(dbl));  
  
        string s = "war and peace";  
        Console.WriteLine(s.ToTitleCase());  
    }  
}  
// The example displays the following output:
```

```
//      True
//      War and Peace
```

To compile the Visual Basic code, use this command:

Console

```
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`:

Console

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

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

## **.NET feedback**

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Type conversion in .NET

Article • 09/15/2021

Every value has an associated type, which defines attributes such as the amount of space allocated to the value, the range of possible values it can have, and the members that it makes available. Many values can be expressed as more than one type. For example, the value 4 can be expressed as an integer or a floating-point value. Type conversion creates a value in a new type that is equivalent to the value of an old type, but does not necessarily preserve the identity (or exact value) of the original object.

.NET automatically supports the following conversions:

- Conversion from a derived class to a base class. This means, for example, that an instance of any class or structure can be converted to an [Object](#) instance. This conversion does not require a casting or conversion operator.
- Conversion from a base class back to the original derived class. In C#, this conversion requires a casting operator. In Visual Basic, it requires the  `CType` operator if  `Option Strict` is on.
- Conversion from a type that implements an interface to an interface object that represents that interface. This conversion does not require a casting or conversion operator.
- Conversion from an interface object back to the original type that implements that interface. In C#, this conversion requires a casting operator. In Visual Basic, it requires the  `CType` operator if  `Option Strict` is on.

In addition to these automatic conversions, .NET provides several features that support custom type conversion. These include the following:

- The  `Implicit` operator, which defines the available widening conversions between types. For more information, see the [Implicit Conversion with the Implicit Operator](#) section.
- The  `Explicit` operator, which defines the available narrowing conversions between types. For more information, see the [Explicit Conversion with the Explicit Operator](#) section.
- The [IConvertible](#) interface, which defines conversions to each of the base .NET data types. For more information, see [The IConvertible Interface](#) section.

- The [Convert](#) class, which provides a set of methods that implement the methods in the [IConvertible](#) interface. For more information, see [The Convert Class](#) section.
- The [TypeConverter](#) class, which is a base class that can be extended to support the conversion of a specified type to any other type. For more information, see [The TypeConverter Class](#) section.

## Implicit conversion with the `implicit` operator

Widening conversions involve the creation of a new value from the value of an existing type that has either a more restrictive range or a more restricted member list than the target type. Widening conversions cannot result in data loss (although they may result in a loss of precision). Because data cannot be lost, compilers can handle the conversion implicitly or transparently, without requiring the use of an explicit conversion method or a casting operator.

### ⓘ Note

Although code that performs an implicit conversion can call a conversion method or use a casting operator, their use is not required by compilers that support implicit conversions.

For example, the [Decimal](#) type supports implicit conversions from [Byte](#), [Char](#), [Int16](#), [Int32](#), [Int64](#), [SByte](#), [UInt16](#), [UInt32](#), and [UInt64](#) values. The following example illustrates some of these implicit conversions in assigning values to a [Decimal](#) variable.

C#

```
byte byteValue = 16;
short shortValue = -1024;
int intValue = -1034000;
long longValue = 1152921504606846976;
ulong ulongValue = UInt64.MaxValue;

decimal decimalValue;

decimalValue = byteValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is
{1}.",
    byteValue.GetType().Name, decimalValue);

decimalValue = shortValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is
{1}.",
    shortValue.GetType().Name, decimalValue);
```

```

decimalValue = intValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
    intValue.GetType().Name, decimalValue);

decimalValue = longValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
    longValue.GetType().Name, decimalValue);

decimalValue = ulongValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
    ulongValue.GetType().Name, decimalValue);

// The example displays the following output:
//     After assigning a Byte value, the Decimal value is 16.
//     After assigning a Int16 value, the Decimal value is -1024.
//     After assigning a Int32 value, the Decimal value is -1034000.
//     After assigning a Int64 value, the Decimal value is
1152921504606846976.
//     After assigning a UInt64 value, the Decimal value is
18446744073709551615.

```

If a particular language compiler supports custom operators, you can also define implicit conversions in your own custom types. The following example provides a partial implementation of a signed byte data type named `ByteWithSign` that uses sign-and-magnitude representation. It supports implicit conversion of `Byte` and `SByte` values to `ByteWithSign` values.

C#

```

public struct ByteWithSign
{
    private SByte signValue;
    private Byte value;

    public static implicit operator ByteWithSign(SByte value)
    {
        ByteWithSign newValue;
        newValue.signValue = (SByte)Math.Sign(value);
        newValue.value = (byte)Math.Abs(value);
        return newValue;
    }

    public static implicit operator ByteWithSign(Byte value)
    {
        ByteWithSign newValue;
        newValue.signValue = 1;
        newValue.value = value;
        return newValue;
    }
}

```

```
public override string ToString()
{
    return (signValue * value).ToString();
}
```

Client code can then declare a `ByteWithSign` variable and assign it `Byte` and `SByte` values without performing any explicit conversions or using any casting operators, as the following example shows.

C#

```
SByte sbyteValue = -120;
ByteWithSign value = sbyteValue;
Console.WriteLine(value);
value = Byte.MaxValue;
Console.WriteLine(value);
// The example displays the following output:
//      -120
//      255
```

## Explicit conversion with the `explicit` operator

Narrowing conversions involve the creation of a new value from the value of an existing type that has either a greater range or a larger member list than the target type. Because a narrowing conversion can result in a loss of data, compilers often require that the conversion be made explicit through a call to a conversion method or a casting operator. That is, the conversion must be handled explicitly in developer code.

### ⓘ Note

The major purpose of requiring a conversion method or casting operator for narrowing conversions is to make the developer aware of the possibility of data loss or an `OverflowException` so that it can be handled in code. However, some compilers can relax this requirement. For example, in Visual Basic, if `Option Strict` is off (its default setting), the Visual Basic compiler tries to perform narrowing conversions implicitly.

For example, the `UInt32`, `Int64`, and `UInt64` data types have ranges that exceed that the `Int32` data type, as the following table shows.

Type	Comparison with range of Int32
Int64	Int64.MaxValue is greater than Int32.MaxValue, and Int64.MinValue is less than (has a greater negative range than) Int32.MinValue.
UInt32	UInt32.MaxValue is greater than Int32.MaxValue.
UInt64	UInt64.MaxValue is greater than Int32.MaxValue.

To handle such narrowing conversions, .NET allows types to define an `Explicit` operator. Individual language compilers can then implement this operator using their own syntax, or a member of the `Convert` class can be called to perform the conversion. (For more information about the `Convert` class, see [The Convert Class](#) later in this topic.) The following example illustrates the use of language features to handle the explicit conversion of these potentially out-of-range integer values to `Int32` values.

C#

```
long number1 = int.MaxValue + 20L;
uint number2 = int.MaxValue - 1000;
ulong number3 = int.MaxValue;

int intNumber;

try
{
    intNumber = checked((int)number1);
    Console.WriteLine("After assigning a {0} value, the Integer value is
{1}.",
                      number1.GetType().Name, intNumber);
}
catch (OverflowException)
{
    if (number1 > int.MaxValue)
        Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                          number1, int.MaxValue);
    else
        Console.WriteLine("Conversion failed: {0} is less than {1}.",
                          number1, int.MinValue);
}

try
{
    intNumber = checked((int)number2);
    Console.WriteLine("After assigning a {0} value, the Integer value is
{1}.",
                      number2.GetType().Name, intNumber);
}
catch (OverflowException)
{
    Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                      number2, int.MaxValue);
}
```

```

        number2, int.MaxValue);
}

try
{
    intNumber = checked((int)number3);
    Console.WriteLine("After assigning a {0} value, the Integer value is
{1}.",
                      number3.GetType().Name, intNumber);
}
catch (OverflowException)
{
    Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                      number1, int.MaxValue);
}

// The example displays the following output:
//      Conversion failed: 2147483667 exceeds 2147483647.
//      After assigning a UInt32 value, the Integer value is 2147482647.
//      After assigning a UInt64 value, the Integer value is 2147483647.

```

Explicit conversions can produce different results in different languages, and these results can differ from the value returned by the corresponding [Convert](#) method. For example, if the [Double](#) value 12.63251 is converted to an [Int32](#), both the Visual Basic [CInt](#) method and the .NET [Convert.ToInt32\(Double\)](#) method round the [Double](#) to return a value of 13, but the C# [\(int\)](#) operator truncates the [Double](#) to return a value of 12. Similarly, the C# [\(int\)](#) operator does not support Boolean-to-integer conversion, but the Visual Basic [CInt](#) method converts a value of [true](#) to -1. On the other hand, the [Convert.ToInt32\(Boolean\)](#) method converts a value of [true](#) to 1.

Most compilers allow explicit conversions to be performed in a checked or unchecked manner. When a checked conversion is performed, an [OverflowException](#) is thrown when the value of the type to be converted is outside the range of the target type. When an unchecked conversion is performed under the same conditions, the conversion might not throw an exception, but the exact behavior becomes undefined and an incorrect value might result.

### ⓘ Note

In C#, checked conversions can be performed by using the [checked](#) keyword together with a casting operator, or by specifying the [/checked+](#) compiler option. Conversely, unchecked conversions can be performed by using the [unchecked](#) keyword together with the casting operator, or by specifying the [/checked-](#) compiler option. By default, explicit conversions are unchecked. In Visual Basic, checked conversions can be performed by clearing the [Remove integer overflow](#)

checks check box in the project's **Advanced Compiler Settings** dialog box, or by specifying the `/removeintchecks-` compiler option. Conversely, unchecked conversions can be performed by selecting the **Remove integer overflow checks** check box in the project's **Advanced Compiler Settings** dialog box or by specifying the `/removeintchecks+` compiler option. By default, explicit conversions are checked.

The following C# example uses the `checked` and `unchecked` keywords to illustrate the difference in behavior when a value outside the range of a `Byte` is converted to a `Byte`. The checked conversion throws an exception, but the unchecked conversion assigns `Byte.MaxValue` to the `Byte` variable.

C#

```
int largeValue = Int32.MaxValue;
byte newValue;

try
{
    newValue = unchecked((byte)largeValue);
    Console.WriteLine("Converted the {0} value {1} to the {2} value {3}.",
                      largeValue.GetType().Name, largeValue,
                      newValue.GetType().Name, newValue);
}
catch (OverflowException)
{
    Console.WriteLine("{0} is outside the range of the Byte data type.",
                      largeValue);
}

try
{
    newValue = checked((byte)largeValue);
    Console.WriteLine("Converted the {0} value {1} to the {2} value {3}.",
                      largeValue.GetType().Name, largeValue,
                      newValue.GetType().Name, newValue);
}
catch (OverflowException)
{
    Console.WriteLine("{0} is outside the range of the Byte data type.",
                      largeValue);
}

// The example displays the following output:
//     Converted the Int32 value 2147483647 to the Byte value 255.
//     2147483647 is outside the range of the Byte data type.
```

If a particular language compiler supports custom overloaded operators, you can also define explicit conversions in your own custom types. The following example provides a

partial implementation of a signed byte data type named `ByteWithSign` that uses sign-and-magnitude representation. It supports explicit conversion of `Int32` and `UInt32` values to `ByteWithSign` values.

C#

```
public struct ByteWithSignE
{
    private SByte signValue;
    private Byte value;

    private const byte.MaxValue = byte.MaxValue;
    private const int.MinValue = -1 * byte.MaxValue;

    public static explicit operator ByteWithSignE(int value)
    {
        // Check for overflow.
        if (value > ByteWithSignE.MaxValue || value <
ByteWithSignE.MinValue)
            throw new OverflowException(String.Format("{0} is out of range
of the ByteWithSignE data type.",
                                            value));

        ByteWithSignE newValue;
        newValue.signValue = (SByte)Math.Sign(value);
        newValue.value = (byte)Math.Abs(value);
        return newValue;
    }

    public static explicit operator ByteWithSignE(uint value)
    {
        if (value > ByteWithSignE.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the ByteWithSignE data type.",
                                            value));

        ByteWithSignE newValue;
        newValue.signValue = 1;
        newValue.value = (byte)value;
        return newValue;
    }

    public override string ToString()
    {
        return (signValue * value).ToString();
    }
}
```

Client code can then declare a `ByteWithSign` variable and assign it `Int32` and `UInt32` values if the assignments include a casting operator or a conversion method, as the following example shows.

C#

```
ByteWithSignE value;

try
{
    int intValue = -120;
    value = (ByteWithSignE)intValue;
    Console.WriteLine(value);
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}

try
{
    uint uintValue = 1024;
    value = (ByteWithSignE)uintValue;
    Console.WriteLine(value);
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}
// The example displays the following output:
//      -120
//      '1024' is out of range of the ByteWithSignE data type.
```

## The [IConvertible](#) interface

To support the conversion of any type to a common language runtime base type, .NET provides the [IConvertible](#) interface. The implementing type is required to provide the following:

- A method that returns the [TypeCode](#) of the implementing type.
- Methods to convert the implementing type to each common language runtime base type ([Boolean](#), [Byte](#), [DateTime](#), [Decimal](#), [Double](#), and so on).
- A generalized conversion method to convert an instance of the implementing type to another specified type. Conversions that are not supported should throw an [InvalidOperationException](#).

Each common language runtime base type (that is, the [Boolean](#), [Byte](#), [Char](#), [DateTime](#), [Decimal](#), [Double](#), [Int16](#), [Int32](#), [Int64](#), [SByte](#), [Single](#), [String](#), [UInt16](#), [UInt32](#), and [UInt64](#)), as well as the [DBNull](#) and [Enum](#) types, implement the [IConvertible](#) interface. However, these are explicit interface implementations; the conversion method can be called only

through an [IConvertible](#) interface variable, as the following example shows. This example converts an [Int32](#) value to its equivalent [Char](#) value.

C#

```
int codePoint = 1067;
IConvertible iConv = codePoint;
char ch = iConv.ToChar(null);
Console.WriteLine("Converted {0} to {1}.", codePoint, ch);
```

The requirement to call the conversion method on its interface rather than on the implementing type makes explicit interface implementations relatively expensive. Instead, we recommend that you call the appropriate member of the [Convert](#) class to convert between common language runtime base types. For more information, see the next section, [The Convert Class](#).

#### ⓘ Note

In addition to the [IConvertible](#) interface and the [Convert](#) class provided by .NET, individual languages may also provide ways to perform conversions. For example, C# uses casting operators; Visual Basic uses compiler-implemented conversion functions such as  `CType`,  `CInt`, and  `DirectCast`.

For the most part, the [IConvertible](#) interface is designed to support conversion between the base types in .NET. However, the interface can also be implemented by a custom type to support conversion of that type to other custom types. For more information, see the section [Custom Conversions with the ChangeType Method](#) later in this topic.

## The Convert class

Although each base type's [IConvertible](#) interface implementation can be called to perform a type conversion, calling the methods of the [System.Convert](#) class is the recommended language-neutral way to convert from one base type to another. In addition, the [Convert.ChangeType\(Object, Type, IFormatProvider\)](#) method can be used to convert from a specified custom type to another type.

## Conversions between base types

The [Convert](#) class provides a language-neutral way to perform conversions between base types and is available to all languages that target the common language runtime. It provides a complete set of methods for both widening and narrowing conversions, and

throws an [InvalidCastException](#) for conversions that are not supported (such as the conversion of a [DateTime](#) value to an integer value). Narrowing conversions are performed in a checked context, and an [OverflowException](#) is thrown if the conversion fails.

 **Important**

Because the [Convert](#) class includes methods to convert to and from each base type, it eliminates the need to call each base type's [IConvertible](#) explicit interface implementation.

The following example illustrates the use of the [System.Convert](#) class to perform several widening and narrowing conversions between .NET base types.

C#

```
// Convert an Int32 value to a Decimal (a widening conversion).
int integralValue = 12534;
decimal decimalValue = Convert.ToDecimal(integralValue);
Console.WriteLine("Converted the {0} value {1} to " +
                  "the {2} value {3:N2}。",
                  integralValue.GetType().Name,
                  integralValue,
                  decimalValue.GetType().Name,
                  decimalValue);

// Convert a Byte value to an Int32 value (a widening conversion).
byte byteValue = Byte.MaxValue;
int integralValue2 = Convert.ToInt32(byteValue);
Console.WriteLine("Converted the {0} value {1} to " +
                  "the {2} value {3:G}。",
                  byteValue.GetType().Name,
                  byteValue,
                  integralValue2.GetType().Name,
                  integralValue2);

// Convert a Double value to an Int32 value (a narrowing conversion).
double doubleValue = 16.32513e12;
try
{
    long longValue = Convert.ToInt64(doubleValue);
    Console.WriteLine("Converted the {0} value {1:E} to " +
                      "the {2} value {3:N0}。",
                      doubleValue.GetType().Name,
                      doubleValue,
                      longValue.GetType().Name,
                      longValue);
}
catch (OverflowException)
{
    Console.WriteLine("Unable to convert the {0:E} value {1}.");
}
```

```

                doubleValue.GetType().Name,
doubleValue);
}

// Convert a signed byte to a byte (a narrowing conversion).
sbyte sbyteValue = -16;
try
{
    byte byteValue2 = Convert.ToByte(sbyteValue);
    Console.WriteLine("Converted the {0} value {1} to " +
                      "the {2} value {3:G}.",
                      sbyteValue.GetType().Name,
                      sbyteValue,
                      byteValue2.GetType().Name,
                      byteValue2);
}
catch (OverflowException)
{
    Console.WriteLine("Unable to convert the {0} value {1}.",
                      sbyteValue.GetType().Name,
                      sbyteValue);
}
// The example displays the following output:
//     Converted the Int32 value 12534 to the Decimal value 12,534.00.
//     Converted the Byte value 255 to the Int32 value 255.
//     Converted the Double value 1.632513E+013 to the Int64 value
16,325,130,000,000.
//     Unable to convert the SByte value -16.

```

In some cases, particularly when converting to and from floating-point values, a conversion may involve a loss of precision, even though it does not throw an [OverflowException](#). The following example illustrates this loss of precision. In the first case, a [Decimal](#) value has less precision (fewer significant digits) when it is converted to a [Double](#). In the second case, a [Double](#) value is rounded from 42.72 to 43 in order to complete the conversion.

C#

```

double doubleValue;

// Convert a Double to a Decimal.
decimal decimalValue = 13956810.96702888123451471211m;
doubleValue = Convert.ToDouble(decimalValue);
Console.WriteLine("{0} converted to {1}.", decimalValue, doubleValue);

doubleValue = 42.72;
try
{
    int integerValue = Convert.ToInt32(doubleValue);
    Console.WriteLine("{0} converted to {1}.",
                      doubleValue, integerValue);
}

```

```
        }
        catch (OverflowException)
        {
            Console.WriteLine("Unable to convert {0} to an integer.",
                doubleValue);
        }
        // The example displays the following output:
        //      13956810.96702888123451471211 converted to 13956810.9670289.
        //      42.72 converted to 43.
```

For a table that lists both the widening and narrowing conversions supported by the [Convert](#) class, see [Type Conversion Tables](#).

## Custom conversions with the `ChangeType` method

In addition to supporting conversions to each of the base types, the [Convert](#) class can be used to convert a custom type to one or more predefined types. This conversion is performed by the [Convert.ChangeType\(Object, Type, IFormatProvider\)](#) method, which in turn wraps a call to the [IConvertible.ToType](#) method of the `value` parameter. This means that the object represented by the `value` parameter must provide an implementation of the [IConvertible](#) interface.

### ⓘ Note

Because the [Convert.ChangeType\(Object, Type\)](#) and [Convert.ChangeType\(Object, Type, IFormatProvider\)](#) methods use a `Type` object to specify the target type to which `value` is converted, they can be used to perform a dynamic conversion to an object whose type is not known at compile time. However, note that the [IConvertible](#) implementation of `value` must still support this conversion.

The following example illustrates a possible implementation of the [IConvertible](#) interface that allows a `TemperatureCelsius` object to be converted to a `TemperatureFahrenheit` object and vice versa. The example defines a base class, `Temperature`, that implements the [IConvertible](#) interface and overrides the [Object.ToString](#) method. The derived `TemperatureCelsius` and `TemperatureFahrenheit` classes each override the `ToType` and the `ToString` methods of the base class.

C#

```
using System;

public abstract class Temperature : IConvertible
{
```

```
protected decimal temp;

public Temperature(decimal temperature)
{
    this.temp = temperature;
}

public decimal Value
{
    get { return this.temp; }
    set { this.temp = value; }
}

public override string ToString()
{
    return temp.ToString(null as IFormatProvider) + "°";
}

// IConvertible implementations.
public TypeCode GetTypeCode()
{
    return TypeCode.Object;
}

public bool ToBoolean(IFormatProvider provider)
{
    throw new InvalidCastException(String.Format("Temperature-to-Boolean conversion is not supported."));
}

public byte ToByte(IFormatProvider provider)
{
    if (temp < Byte.MinValue || temp > Byte.MaxValue)
        throw new OverflowException(String.Format("{0} is out of range of the Byte data type.", temp));
    else
        return (byte)temp;
}

public char ToChar(IFormatProvider provider)
{
    throw new InvalidCastException("Temperature-to-Char conversion is not supported.");
}

public DateTime ToDateTime(IFormatProvider provider)
{
    throw new InvalidCastException("Temperature-to-DateTime conversion is not supported.");
}

public decimal ToDecimal(IFormatProvider provider)
{
    return temp;
}
```

```
public double ToDouble(IFormatProvider provider)
{
    return (double)temp;
}

public shortToInt16(IFormatProvider provider)
{
    if (temp < Int16.MinValue || temp > Int16.MaxValue)
        throw new OverflowException(String.Format("{0} is out of range
of the Int16 data type.", temp));
    else
        return (short)Math.Round(temp);
}

public intToInt32(IFormatProvider provider)
{
    if (temp < Int32.MinValue || temp > Int32.MaxValue)
        throw new OverflowException(String.Format("{0} is out of range
of the Int32 data type.", temp));
    else
        return (int)Math.Round(temp);
}

public longToInt64(IFormatProvider provider)
{
    if (temp < Int64.MinValue || temp > Int64.MaxValue)
        throw new OverflowException(String.Format("{0} is out of range
of the Int64 data type.", temp));
    else
        return (long)Math.Round(temp);
}

public sbyteToSByte(IFormatProvider provider)
{
    if (temp < SByte.MinValue || temp > SByte.MaxValue)
        throw new OverflowException(String.Format("{0} is out of range
of the SByte data type.", temp));
    else
        return (sbyte)temp;
}

public floatToSingle(IFormatProvider provider)
{
    return (float)temp;
}

public virtual stringToString(IFormatProvider provider)
{
    return temp.ToString(provider) + "°";
}

// If conversionType is implemented by another IConvertible method, call
it.
public virtual objectToType(Type conversionType, IFormatProvider
```

```

provider)
{
    switch (Type.GetTypeCode(conversionType))
    {
        case TypeCode.Boolean:
            return this.ToBoolean(provider);
        case TypeCode.Byte:
            return this.ToByte(provider);
        case TypeCode.Char:
            return this.ToChar(provider);
        case TypeCode.DateTime:
            return this.ToDateTime(provider);
        case TypeCode.Decimal:
            return this.ToDecimal(provider);
        case TypeCode.Double:
            return this.ToDouble(provider);
        case TypeCode.Empty:
            throw new NullReferenceException("The target type is
null.");
        case TypeCode.Int16:
            return this.ToInt16(provider);
        case TypeCode.Int32:
            return this.ToInt32(provider);
        case TypeCode.Int64:
            return this.ToInt64(provider);
        case TypeCode.Object:
            // Leave conversion of non-base types to derived classes.
            throw new InvalidCastException(String.Format("Cannot convert
from Temperature to {0}.",
                                                conversionType.Name));
        case TypeCode.SByte:
            return this.ToSByte(provider);
        case TypeCode.Single:
            return this.ToSingle(provider);
        case TypeCode.String:
            IConvertible iconv = this;
            return iconv.ToString(provider);
        case TypeCode.UInt16:
            return this.ToUInt16(provider);
        case TypeCode.UInt32:
            return this.ToUInt32(provider);
        case TypeCode.UInt64:
            return this.ToUInt64(provider);
        default:
            throw new InvalidCastException("Conversion not supported.");
    }
}

public ushort ToUInt16(IFormatProvider provider)
{
    if (temp < UInt16.MinValue || temp > UInt16.MaxValue)
        throw new OverflowException(String.Format("{0} is out of range
of the UInt16 data type.", temp));
    else
        return (ushort)Math.Round(temp);
}

```

```

    }

    public uint ToUInt32(IFormatProvider provider)
    {
        if (temp < UInt32.MinValue || temp > UInt32.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the UInt32 data type.", temp));
        else
            return (uint)Math.Round(temp);
    }

    public ulong ToUInt64(IFormatProvider provider)
    {
        if (temp < UInt64.MinValue || temp > UInt64.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the UInt64 data type.", temp));
        else
            return (ulong)Math.Round(temp);
    }
}

public class TemperatureCelsius : Temperature, IConvertible
{
    public TemperatureCelsius(decimal value) : base(value)
    {
    }

    // Override ToString methods.
    public override string ToString()
    {
        return this.ToString(null);
    }

    public override string ToString(IFormatProvider provider)
    {
        return temp.ToString(provider) + "°C";
    }

    // If conversionType is a implemented by another IConvertible method,
    // call it.
    public override object ToType(Type conversionType, IFormatProvider
provider)
    {
        // For non-objects, call base method.
        if (Type.GetTypeCode(conversionType) != TypeCode.Object)
        {
            return base.ToType(conversionType, provider);
        }
        else
        {
            if (conversionType.Equals(typeof(TemperatureCelsius)))
                return this;
            else if (conversionType.Equals(typeof(TemperatureFahrenheit)))
                return new TemperatureFahrenheit((decimal)this.temp * 9 / 5
+ 32);
        }
    }
}

```

```

        else
            throw new InvalidCastException(String.Format("Cannot convert
from Temperature to {0}.",
                                            conversionType.Name));
    }
}

public class TemperatureFahrenheit : Temperature, IConvertible
{
    public TemperatureFahrenheit(decimal value) : base(value)
    {
    }

    // Override ToString methods.
    public override string ToString()
    {
        return this.ToString(null);
    }

    public override string ToString(IFormatProvider provider)
    {
        return temp.ToString(provider) + "°F";
    }

    public override object ToType(Type conversionType, IFormatProvider
provider)
    {
        // For non-objects, call base method.
        if (Type.GetTypeCode(conversionType) != TypeCode.Object)
        {
            return base.ToType(conversionType, provider);
        }
        else
        {
            // Handle conversion between derived classes.
            if (conversionType.Equals(typeof(TemperatureFahrenheit)))
                return this;
            else if (conversionType.Equals(typeof(TemperatureCelsius)))
                return new TemperatureCelsius((decimal)(this.temp - 32) * 5
/ 9);
            // Unspecified object type: throw an InvalidCastException.
            else
                throw new InvalidCastException(String.Format("Cannot convert
from Temperature to {0}.",
                                            conversionType.Name));
        }
    }
}

```

The following example illustrates several calls to these `IConvertible` implementations to convert `TemperatureCelsius` objects to `TemperatureFahrenheit` objects and vice versa.

C#

```
TemperatureCelsius tempC1 = new TemperatureCelsius(0);
TemperatureFahrenheit tempF1 =
(TemperatureFahrenheit)Convert.ChangeType(tempC1,
typeof(TemperatureFahrenheit), null);
Console.WriteLine("{0} equals {1}.", tempC1, tempF1);
TemperatureCelsius tempC2 = (TemperatureCelsius)Convert.ChangeType(tempC1,
typeof(TemperatureCelsius), null);
Console.WriteLine("{0} equals {1}.", tempC1, tempC2);
TemperatureFahrenheit tempF2 = new TemperatureFahrenheit(212);
TemperatureCelsius tempC3 = (TemperatureCelsius)Convert.ChangeType(tempF2,
typeof(TemperatureCelsius), null);
Console.WriteLine("{0} equals {1}.", tempF2, tempC3);
TemperatureFahrenheit tempF3 =
(TemperatureFahrenheit)Convert.ChangeType(tempF2,
typeof(TemperatureFahrenheit), null);
Console.WriteLine("{0} equals {1}.", tempF2, tempF3);
// The example displays the following output:
//      0°C equals 32°F.
//      0°C equals 0°C.
//      212°F equals 100°C.
//      212°F equals 212°F.
```

## The TypeConverter class

.NET also allows you to define a type converter for a custom type by extending the [System.ComponentModel.TypeConverter](#) class and associating the type converter with the type through a [System.ComponentModel.TypeConverterAttribute](#) attribute. The following table highlights the differences between this approach and implementing the [IConvertible](#) interface for a custom type.

### ⓘ Note

Design-time support can be provided for a custom type only if it has a type converter defined for it.

Conversion using TypeConverter	Conversion using IConvertible
Is implemented for a custom type by deriving a separate class from <a href="#">TypeConverter</a> . This derived class is associated with the custom type by applying a <a href="#">TypeConverterAttribute</a> attribute.	Is implemented by a custom type to perform conversion. A user of the type invokes an <a href="#">IConvertible</a> conversion method on the type.
Can be used both at design time and at run time.	Can be used only at run time.

Conversion using TypeConverter	Conversion using IConvertible
Uses reflection; therefore, is slower than conversion enabled by <a href="#">IConvertible</a> .	Does not use reflection.
Allows two-way type conversions from the custom type to other data types, and from other data types to the custom type. For example, a <a href="#">TypeConverter</a> defined for <code>MyType</code> allows conversions from <code>MyType</code> to <a href="#">String</a> , and from <a href="#">String</a> to <code>MyType</code> .	Allows conversion from a custom type to other data types, but not from other data types to the custom type.

For more information about using type converters to perform conversions, see [System.ComponentModel.TypeConverter](#).

## See also

- [System.Convert](#)
- [IConvertible](#)
- [Type Conversion Tables](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Type conversion tables in .NET

Article • 09/15/2021

Widening conversion occurs when a value of one type is converted to another type that is of equal or greater size. A narrowing conversion occurs when a value of one type is converted to a value of another type that is of a smaller size. The tables in this topic illustrate the behaviors exhibited by both types of conversions.

## Widening conversions

The following table describes the widening conversions that can be performed without the loss of information.

[ ] [Expand table](#)

Type	Can be converted without data loss to
Byte	UInt16, Int16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
SByte	Int16, Int32, Int64, Single, Double, Decimal
Int16	Int32, Int64, Single, Double, Decimal
UInt16	UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Char	UInt16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Int32	Int64, Double, Decimal
UInt32	Int64, UInt64, Double, Decimal
Int64	Decimal
UInt64	Decimal
Single	Double

Some widening conversions to [Single](#) or [Double](#) can cause a loss of precision. The following table describes the widening conversions that sometimes result in a loss of information.

[ ] [Expand table](#)

Type	Can be converted to
Int32	Single

Type	Can be converted to
UInt32	Single
Int64	Single, Double
UInt64	Single, Double
Decimal	Single, Double

## Narrowing conversions

A narrowing conversion to [Single](#) or [Double](#) can cause a loss of information. If the target type cannot properly express the magnitude of the source, the resulting type is set to the constant `PositiveInfinity` or `NegativeInfinity`. `PositiveInfinity` results from dividing a positive number by zero and is also returned when the value of a [Single](#) or [Double](#) exceeds the value of the `.MaxValue` field. `NegativeInfinity` results from dividing a negative number by zero and is also returned when the value of a [Single](#) or [Double](#) falls below the value of the `.MinValue` field. A conversion from a [Double](#) to a [Single](#) might result in `PositiveInfinity` or `NegativeInfinity`.

A narrowing conversion can also result in a loss of information for other data types. However, an [OverflowException](#) is thrown if the value of a type that is being converted falls outside of the range specified by the target type's `.MaxValue` and `.MinValue` fields, and the conversion is checked by the runtime to ensure that the value of the target type does not exceed its `.MaxValue` or `.MinValue`. Conversions that are performed with the [System.Convert](#) class are always checked in this manner.

The following table lists conversions that throw an [OverflowException](#) using [System.Convert](#) or any checked conversion if the value of the type being converted is outside the defined range of the resulting type.

[ ] [Expand table](#)

Type	Can be converted to
Byte	<a href="#">SByte</a>
SByte	<a href="#">Byte</a> , <a href="#">UInt16</a> , <a href="#">UInt32</a> , <a href="#">UInt64</a>
Int16	<a href="#">Byte</a> , <a href="#">SByte</a> , <a href="#">UInt16</a>
UInt16	<a href="#">Byte</a> , <a href="#">SByte</a> , <a href="#">Int16</a>
Int32	<a href="#">Byte</a> , <a href="#">SByte</a> , <a href="#">Int16</a> , <a href="#">UInt16</a> , <a href="#">UInt32</a>

Type	Can be converted to
UInt32	Byte, SByte, Int16, UInt16, Int32
Int64	Byte, SByte, Int16, UInt16, Int32, UInt32, UInt64
UInt64	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64
Decimal	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64
Single	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64
Double	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64

## See also

- [System.Convert](#)
- [Type Conversion in .NET](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Convert class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The static [Convert](#) class contains methods that are primarily used to support conversion to and from the base data types in .NET. The supported base types are [Boolean](#), [Char](#), [SByte](#), [Byte](#), [Int16](#), [Int32](#), [Int64](#), [UInt16](#), [UInt32](#), [UInt64](#), [Single](#), [Double](#), [Decimal](#), [DateTime](#) and [String](#). In addition, the [Convert](#) class includes methods to support other kinds of conversions.

## Conversions to and from base types

A conversion method exists to convert every base type to every other base type. However, the actual call to a particular conversion method can produce one of five outcomes, depending on the value of the base type at run time and the target base type. These five outcomes are:

- No conversion. This occurs when an attempt is made to convert from a type to itself (for example, by calling [Convert.ToInt32\(Int32\)](#) with an argument of type [Int32](#)). In this case, the method simply returns an instance of the original type.
- An [InvalidOperationException](#). This occurs when a particular conversion is not supported. An [InvalidOperationException](#) is thrown for the following conversions:
  - Conversions from [Char](#) to [Boolean](#), [Single](#), [Double](#), [Decimal](#), or [DateTime](#).
  - Conversions from [Boolean](#), [Single](#), [Double](#), [Decimal](#), or [DateTime](#) to [Char](#).
  - Conversions from [DateTime](#) to any other type except [String](#).
  - Conversions from any other type, except [String](#), to [DateTime](#).
- A [FormatException](#). This occurs when the attempt to convert a string value to any other base type fails because the string is not in the proper format. The exception is thrown for the following conversions:
  - A string to be converted to a [Boolean](#) value does not equal [Boolean.TrueString](#) or [Boolean.FalseString](#).
  - A string to be converted to a [Char](#) value consists of multiple characters.
  - A string to be converted to any numeric type is not recognized as a valid number.
  - A string to be converted to a [DateTime](#) is not recognized as a valid date and time value.

- A successful conversion. For conversions between two different base types not listed in the previous outcomes, all widening conversions as well as all narrowing conversions that do not result in a loss of data will succeed and the method will return a value of the targeted base type.
- An [OverflowException](#). This occurs when a narrowing conversion results in a loss of data. For example, trying to convert a [Int32](#) instance whose value is 10000 to a [Byte](#) type throws an [OverflowException](#) because 10000 is outside the range of the [Byte](#) data type.

An exception will not be thrown if the conversion of a numeric type results in a loss of precision (that is, the loss of some least significant digits). However, an exception will be thrown if the result is larger than can be represented by the particular conversion method's return value type.

For example, when a [Double](#) is converted to a [Single](#), a loss of precision might occur but no exception is thrown. However, if the magnitude of the [Double](#) is too large to be represented by a [Single](#), an overflow exception is thrown.

## Non-decimal numbers

The [Convert](#) class includes static methods that you can call to convert integral values to non-decimal string representations, and to convert strings that represent non-decimal numbers to integral values. Each of these conversion methods includes a `base` argument that lets you specify the number system; binary (base 2), octal (base 8), and hexadecimal (base 16), as well as decimal (base 10). There is a set of methods to convert each of the CLS-compliant primitive integral types to a string, and one to convert a string to each of the primitive integral types:

- [ToString\(Byte, Int32\)](#) and [ToByte\(String, Int32\)](#), to convert a byte value to and from a string in a specified base.
- [ToString\(Int16, Int32\)](#) and [ToInt16\(String, Int32\)](#), to convert a 16-bit signed integer to and from a string in a specified base.
- [ToString\(Int32, Int32\)](#) and [ToInt32\(String, Int32\)](#), to convert a 32-bit signed integer to and from a string in a specified base.
- [ToString\(Int64, Int32\)](#) and [ToInt64\(String, Int32\)](#), to convert a 64-bit signed integer to and from a string in a specified base.
- [ToSByte\(String, Int32\)](#), to convert the string representation of a byte value in a specified format to a signed byte.

- [ToInt16\(String, Int32\)](#), to convert the string representation of an integer in a specified format to an unsigned 16-bit integer.
- [ToInt32\(String, Int32\)](#), to convert the string representation of an integer in a specified format to an unsigned 32-bit integer.
- [ToInt64\(String, Int32\)](#), to convert the string representation of an integer in a specified format to an unsigned 64-bit integer.

The following example converts the value of [Int16.MaxValue](#) to a string in all supported numeric formats. It then converts the string value back to a [Int16](#) value.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        int[] baseValues = { 2, 8, 10, 16 };
        short value = Int16.MaxValue;
        foreach (var baseValue in baseValues) {
            String s = Convert.ToString(value, baseValue);
            short value2 = Convert.ToInt16(s, baseValue);

            Console.WriteLine("{0} --> {1} (base {2}) --> {3}",
                value, s, baseValue, value2);
        }
    }
    // The example displays the following output:
    //      32767 --> 11111111111111 (base 2) --> 32767
    //      32767 --> 77777 (base 8) --> 32767
    //      32767 --> 32767 (base 10) --> 32767
    //      32767 --> 7fff (base 16) --> 32767
}
```

## Conversions from custom objects to base types

In addition to supporting conversions between the base types, the [Convert](#) method supports conversion of any custom type to any base type. To do this, the custom type must implement the [IConvertible](#) interface, which defines methods for converting the implementing type to each of the base types. Conversions that are not supported by a particular type should throw an [InvalidOperationException](#).

When the [ChangeType](#) method is passed a custom type as its first parameter, or when the [Convert.ToType](#) method (such as [Convert.ToInt32\(Object\)](#)) or

`Convert.ToDouble(Object, IFormatProvider)` is called and passed an instance of a custom type as its first parameter, the `Convert` method, in turn, calls the custom type's `IConvertible` implementation to perform the conversion. For more information, see [Type Conversion in .NET](#).

## Culture-specific formatting information

All the base type conversion methods and the `ChangeType` method include overloads that have a parameter of type `IFormatProvider`. For example, the `Convert.ToBoolean` method has the following two overloads:

- `Convert.ToBoolean(Object, IFormatProvider)`
- `Convert.ToBoolean(String, IFormatProvider)`

The `IFormatProvider` parameter can supply culture-specific formatting information to assist the conversion process. However, it is ignored by most of the base type conversion methods. It is used only by the following base type conversion methods. If a `null` `IFormatProvider` argument is passed to these methods, the `CultureInfo` object that represents the current culture is used.

- By methods that convert a value to a numeric type. The `IFormatProvider` parameter is used by the overload that has parameters of type `String` and `IFormatProvider`. It is also used by the overload that has parameters of type `Object` and `IFormatProvider` if the object's run-time type is a `String`.
- By methods that convert a value to a date and time. The `IFormatProvider` parameter is used by the overload that has parameters of type `String` and `IFormatProvider`. It is also used by the overload that has parameters of type `Object` and `IFormatProvider` if the object's run-time type is a `String`.
- By the `Convert.ToString` overloads that include an `IFormatProvider` parameter and that convert either a numeric value to a string or a `DateTime` value to a string.

However, any user-defined type that implements `IConvertible` can make use of the `IFormatProvider` parameter.

## Base64 encoding

Base64 encoding converts binary data to a string. Data expressed as base-64 digits can be easily conveyed over data channels that can only transmit 7-bit characters. The `Convert` class includes the following methods to support base64 encoding: A set of

methods support converting an array of bytes to and from a [String](#) or to and from an array of Unicode characters consisting of base-64 digit characters.

- [ToBase64String](#), which converts a byte array to a base64-encoded string.
- [ToBase64CharArray](#), which converts a byte array to a base64-encoded character array.
- [FromBase64String](#), which converts a base64-encoded string to a byte array.
- [FromBase64CharArray](#), which converts a base64-encoded character array to a byte array.

## Other common conversions

You can use other .NET classes to perform some conversions that aren't supported by the static methods of the [Convert](#) class. These include:

- Conversion to byte arrays

The [BitConverter](#) class provides methods that convert the primitive numeric types (including [Boolean](#)) to byte arrays and from byte arrays back to primitive data types.

- Character encoding and decoding

The [Encoding](#) class and its derived classes (such as [UnicodeEncoding](#) and [UTF8Encoding](#)) provide methods to encode a character array or a string (that is, to convert them to a byte array in a particular encoding) and to decode an encoded byte array (that is, convert a byte array back to UTF16-encoded Unicode characters). For more information, see [Character Encoding in .NET](#).

## Examples

The following example demonstrates some of the conversion methods in the [Convert](#) class, including [ToInt32](#), [.ToBoolean](#), and [ToString](#).

C#

```
double dNumber = 23.15;

try {
    // Returns 23
    int iNumber = System.Convert.ToInt32(dNumber);
}
catch (System.OverflowException) {
    System.Console.WriteLine(
```

```
        "Overflow in double to int conversion.");
}
// Returns True
bool bNumber = System.Convert.ToBoolean(dNumber);

// Returns "23.15"
string strNumber = System.Convert.ToString(dNumber);

try {
    // Returns '2'
    char chrNumber = System.Convert.ToChar(strNumber[0]);
}
catch (System.ArgumentNullException) {
    System.Console.WriteLine("String is null");
}
catch (System.FormatException) {
    System.Console.WriteLine("String length is greater than 1.");
}

// System.Console.ReadLine() returns a string and it
// must be converted.
int newInteger = 0;
try {
    System.Console.WriteLine("Enter an integer:");
    newInteger = System.Convert.ToInt32(
        System.Console.ReadLine());
}
catch (System.ArgumentNullException) {
    System.Console.WriteLine("String is null.");
}
catch (System.FormatException) {
    System.Console.WriteLine("String does not consist of an " +
        "optional sign followed by a series of digits.");
}
catch (System.OverflowException) {
    System.Console.WriteLine(
        "Overflow in string to int conversion.");
}

System.Console.WriteLine("Your integer as a double is {0}",
    System.Convert.ToDouble(newInteger));
```

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



## .NET feedback

.NET is an open source project.  
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

# Choosing between anonymous and tuple types

Article • 03/09/2023

Choosing the appropriate type involves considering its usability, performance, and tradeoffs compared to other types. Anonymous types have been available since C# 3.0, while generic [System.Tuple<T1,T2>](#) types were introduced with .NET Framework 4.0. Since then new options have been introduced with language level support, such as [System.ValueTuple<T1,T2>](#) - which as the name implies, provide a value type with the flexibility of anonymous types. In this article, you'll learn when it's appropriate to choose one type over the other.

## Usability and functionality

Anonymous types were introduced in C# 3.0 with Language-Integrated Query (LINQ) expressions. With LINQ, developers often project results from queries into anonymous types that hold a few select properties from the objects they're working with. Consider the following example, that instantiates an array of [DateTime](#) objects, and iterates through them projecting into an anonymous type with two properties.

C#

```
var dates = new[]
{
    DateTime.UtcNow.AddHours(-1),
    DateTime.UtcNow,
    DateTime.UtcNow.AddHours(1),
};

foreach (var anonymous in
    dates.Select(
        date => new { Formatted = $"{date:MMM dd, yyyy hh:mm zzz}", date.Ticks }))
{
    Console.WriteLine($"Ticks: {anonymous.Ticks}, Formatted: {anonymous.Formatted}");
}
```

Anonymous types are instantiated by using the [new](#) operator, and the property names and types are inferred from the declaration. If two or more anonymous object initializers in the same assembly specify a sequence of properties that are in the same order and

that have the same names and types, the compiler treats the objects as instances of the same type. They share the same compiler-generated type information.

The previous C# snippet projects an anonymous type with two properties, much like the following compiler-generated C# class:

```
C#  
  
internal sealed class f__AnonymousType0  
{  
    public string Formatted { get; }  
    public long Ticks { get; }  
  
    public f__AnonymousType0(string formatted, long ticks)  
    {  
        Formatted = formatted;  
        Ticks = ticks;  
    }  
}
```

For more information, see [anonymous types](#). The same functionality exists with tuples when projecting into LINQ queries, you can select properties into tuples. These tuples flow through the query, just as anonymous types would. Now consider the following example using the `System.Tuple<string, long>`.

```
C#  
  
var dates = new[]  
{  
    DateTime.UtcNow.AddHours(-1),  
    DateTime.UtcNow,  
    DateTime.UtcNow.AddHours(1),  
};  
  
foreach (var tuple in  
    dates.Select(  
        date => new Tuple<string, long>($"{date:MMM dd, yyyy hh:mm  
zzz}", date.Ticks)))  
{  
    Console.WriteLine($"Ticks: {tuple.Item2}, formatted: {tuple.Item1}");  
}
```

With the `System.Tuple<T1,T2>`, the instance exposes numbered item properties, such as `Item1`, and `Item2`. These property names can make it more difficult to understand the intent of the property values, as the property name only provides the ordinal. Furthermore, the `System.Tuple` types are reference `class` types. The `System.ValueTuple<T1,T2>` however, is a value `struct` type. The following C# snippet,

uses `ValueTuple<string, long>` to project into. In doing so, it assigns using a literal syntax.

```
C#  
  
var dates = new[]  
{  
    DateTime.UtcNow.AddHours(-1),  
    DateTime.UtcNow,  
    DateTime.UtcNow.AddHours(1),  
};  
  
foreach (var (formatted, ticks) in  
    dates.Select(  
        date => (Formatted: ${date:MMM dd, yyyy at hh:mm zzz},  
date.Ticks)))  
{  
    Console.WriteLine($"Ticks: {ticks}, formatted: {formatted}");  
}
```

For more information about tuples, see [Tuple types \(C# reference\)](#) or [Tuples \(Visual Basic\)](#).

The previous examples are all functionally equivalent, however, there are slight differences in their usability and their underlying implementations.

## Tradeoffs

You might want to always use `ValueTuple` over `Tuple`, and anonymous types, but there are tradeoffs you should consider. The `ValueTuple` types are mutable, whereas `Tuple` are read-only. Anonymous types can be used in expression trees, while tuples cannot. The following table is an overview of some of the key differences.

## Key differences

Name	Access modifier	Type	Custom member name	Deconstruction support	Expression tree support
Anonymous types	<code>internal</code>	<code>class</code>	✓	✗	✓
<code>Tuple</code>	<code>public</code>	<code>class</code>	✗	✗	✓
<code>ValueTuple</code>	<code>public</code>	<code>struct</code>	✓	✓	✗

# Serialization

One important consideration when choosing a type, is whether or not it will need to be serialized. Serialization is the process of converting the state of an object into a form that can be persisted or transported. For more information, see [serialization](#). When serialization is important, creating a `class` or `struct` is preferred over anonymous types or tuple types.

# Performance

Performance between these types depends on the scenario. The major impact involves the tradeoff between allocations and copying. In most scenarios, the impact is small. When major impacts could arise, measurements should be taken to inform the decision.

# Conclusion

As a developer choosing between tuples and anonymous types, there are several factors to consider. Generally speaking, if you're not working with [expression trees](#), and you're comfortable with tuple syntax then choose [ValueTuple](#) as they provide a value type with the flexibility to name properties. If you're working with expression trees, and you'd prefer to name properties, choose anonymous types. Otherwise, use [Tuple](#).

## See also

- [Anonymous types](#)
- [Expression trees](#)
- [Tuple types \(C# reference\)](#)
- [Tuples \(Visual Basic\)](#)
- [Type design guidelines](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

# .NET class library overview

Article • 04/19/2023

.NET APIs include classes, interfaces, delegates, and value types that expedite and optimize the development process and provide access to system functionality. To facilitate interoperability between languages, most .NET types are CLS-compliant and can therefore be used from any programming language whose compiler conforms to the common language specification (CLS).

.NET types are the foundation on which .NET applications, components, and controls are built. .NET includes types that perform the following functions:

- Represent base data types and exceptions.
- Encapsulate data structures.
- Perform I/O.
- Access information about loaded types.
- Invoke .NET security checks.
- Provide data access, rich client-side GUI, and server-controlled, client-side GUI.

.NET provides a rich set of interfaces, as well as abstract and concrete (non-abstract) classes. You can use the concrete classes as-is or, in many cases, derive your own classes from them. To use the functionality of an interface, you can either create a class that implements the interface or derive a class from one of the .NET classes that implements the interface.

## Naming conventions

.NET types use a dot syntax naming scheme that connotes a hierarchy. This technique groups related types into namespaces so they can be searched and referenced more easily. The first part of the full name—up to the rightmost dot—is the namespace name. The last part of the name is the type name. For example,

`System.Collections.Generic.List<T>` represents the `List<T>` type, which belongs to the `System.Collections.Generic` namespace. The types in `System.Collections.Generic` can be used to work with generic collections.

This naming scheme makes it easy for library developers extending .NET to create hierarchical groups of types and name them in a consistent, informative manner. It also allows types to be unambiguously identified by their full name (that is, by their namespace and type name), which prevents type name collisions. Library developers are expected to use the following convention when creating names for their namespaces:

*CompanyName.TechologyName*

For example, the namespace `Microsoft.Word` conforms to this guideline.

The use of naming patterns to group related types into namespaces is a useful way to build and document class libraries. However, this naming scheme has no effect on visibility, member access, inheritance, security, or binding. A namespace can be partitioned across multiple assemblies and a single assembly can contain types from multiple namespaces. The assembly provides the formal structure for versioning, deployment, security, loading, and visibility in the common language runtime.

For more information on namespaces and type names, see [Common Type System](#).

## System namespace

The `System` namespace is the root namespace for fundamental types in .NET. This namespace includes classes that represent the base data types used by all applications, for example, `Object` (the root of the inheritance hierarchy), `Byte`, `Char`, `Array`, `Int32`, and `String`. Many of these types correspond to the primitive data types that your programming language uses. When you write code using .NET types, you can use your language's corresponding keyword when a .NET base data type is expected.

The following table lists the base types that .NET supplies, briefly describes each type, and indicates the corresponding type in Visual Basic, C#, C++, and F#.

Category	Class name	Description	Visual Basic data type	C# data type	C++/CLI data type	F# data type
Integer	<code>Byte</code>	An 8-bit unsigned integer.	<code>Byte</code>	<code>byte</code>	<code>unsigned char</code>	<code>byte</code>
	<code>SByte</code>	An 8-bit signed integer.  Not CLS-compliant.	<code>SByte</code>	<code>sbyte</code>	<code>char or signed char</code>	<code>sbyte</code>
	<code>Int16</code>	A 16-bit signed integer.	<code>Short</code>	<code>short</code>	<code>short</code>	<code>int16</code>
	<code>Int32</code>	A 32-bit signed integer.	<code>Integer</code>	<code>int</code>	<code>int or long</code>	<code>int</code>
	<code>Int64</code>	A 64-bit signed integer.	<code>Long</code>	<code>long</code>	<code>__int64</code>	<code>int64</code>

Category	Class name	Description	Visual Basic data type	C# data type	C++/CLI data type	F# data type
	UInt16	A 16-bit unsigned integer.  Not CLS-compliant.	UShort	ushort	unsigned short	uint16
	UInt32	A 32-bit unsigned integer.  Not CLS-compliant.	UInteger	uint	unsigned int or unsigned long	uint32
	UInt64	A 64-bit unsigned integer.  Not CLS-compliant.	ULong	ulong	unsigned __int64	uint64
Floating point	Half	A half-precision (16-bit) floating-point number.				
	Single	A single-precision (32-bit) floating-point number.	Single	float	float	float32 or single
	Double	A double-precision (64-bit) floating-point number.	Double	double	double	float or double
Logical	Boolean	A Boolean value (true or false).	Boolean	bool	bool	bool
Other	Char	A Unicode (16-bit) character.	Char	char	wchar_t	char
	Decimal	A decimal (128-bit) value.	Decimal	decimal	Decimal	decimal
	IntPtr	A signed integer whose size depends on the underlying platform (a 32-bit value on a 32-bit platform and a 64-bit value on a 64-bit platform).		nint		unativeint
	UIntPtr	An unsigned integer whose size depends		nuint		unativeint

Category	Class name	Description	Visual Basic data type	C# data type	C++/CLI data type	F# data type
		on the underlying platform (a 32-bit value on a 32-bit platform and a 64-bit value on a 64-bit platform).				
		Not CLS-compliant.				
	Object	The root of the object hierarchy.	Object	object	Object^	obj
	String	An immutable, fixed-length string of Unicode characters.	String	string	String^	string

In addition to the base data types, the [System](#) namespace contains over 100 classes, ranging from classes that handle exceptions to classes that deal with core runtime concepts, such as application domains and the garbage collector. The [System](#) namespace also contains many second-level namespaces.

For more information about namespaces, use the [.NET API Browser](#) to browse the .NET Class Library. The API reference documentation provides documentation on each namespace, its types, and each of their members.

## Data structures

.NET includes a set of data structures that are the workhorses of many .NET apps. These are mostly collections, but also include other types.

- [Array](#) - Represents an array of strongly typed objects that can be accessed by index. Has a fixed size, per its construction.
- [List<T>](#) - Represents a strongly typed list of objects that can be accessed by index. Is automatically resized as needed.
- [Dictionary<TKey,TValue>](#) - Represents a collection of values that are indexed by a key. Values can be accessed via key. Is automatically resized as needed.
- [Uri](#) - Provides an object representation of a uniform resource identifier (URI) and easy access to the parts of the URI.
- [DateTime](#) - Represents an instant in time, typically expressed as a date and time of day.

# Utility APIs

.NET includes a set of utility APIs that provide functionality for many important tasks.

- [HttpClient](#) - An API for sending HTTP requests and receiving HTTP responses from a resource identified by a URI.
- [XDocument](#) - An API for loading and querying XML documents with LINQ.
- [StreamReader](#) - An API for reading files.
- [StreamWriter](#) - An API for writing files.

# App-model APIs

There are many app models that can be used with .NET, for example:

- [ASP.NET](#) - A web framework for building web sites and services. Supported on Windows, Linux, and macOS (depends on ASP.NET version).
- [.NET MAUI](#) - An app platform for building native apps that run on Windows, macOS, iOS, and Android using C#.
- [Windows Desktop](#) - Includes Windows Presentation Foundation (WPF) and Windows Forms.

## See also

- [Runtime libraries overview](#)
- [Common type system](#)
- [.NET API browser](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Object class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Object](#) class is the ultimate base class of all .NET classes; it is the root of the type hierarchy.

Because all classes in .NET are derived from [Object](#), every method defined in the [Object](#) class is available in all objects in the system. Derived classes can and do override some of these methods, including:

- [Equals](#): Supports comparisons between objects.
- [Finalize](#): Performs cleanup operations before an object is automatically reclaimed.
- [GetHashCode](#): Generates a number corresponding to the value of the object to support the use of a hash table.
- [ToString](#): Manufactures a human-readable text string that describes an instance of the class.

Languages typically don't require a class to declare inheritance from [Object](#) because the inheritance is implicit.

## Performance considerations

If you're designing a class, such as a collection, that must handle any type of object, you can create class members that accept instances of the [Object](#) class. However, the process of boxing and unboxing a type carries a performance cost. If you know your new class will frequently handle certain value types you can use one of two tactics to minimize the cost of boxing.

- Create a general method that accepts an [Object](#) type, and a set of type-specific method overloads that accept each value type you expect your class to frequently handle. If a type-specific method exists that accepts the calling parameter type, no boxing occurs and the type-specific method is invoked. If there is no method argument that matches the calling parameter type, the parameter is boxed and the general method is invoked.
- Design your type and its members to use [generics](#). The common language runtime creates a closed generic type when you create an instance of your class and specify a generic type argument. The generic method is type-specific and can be invoked without boxing the calling parameter.

Although it's sometimes necessary to develop general purpose classes that accept and return [Object](#) types, you can improve performance by also providing a type-specific class to handle a frequently used type. For example, providing a class that is specific to setting and getting Boolean values eliminates the cost of boxing and unboxing Boolean values.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

## [.NET feedback](#)

.NET is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

# System.Object.Equals method

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

This article pertains to the [Object.Equals\(Object\)](#) method.

The type of comparison between the current instance and the `obj` parameter depends on whether the current instance is a reference type or a value type.

- If the current instance is a reference type, the [Equals\(Object\)](#) method tests for reference equality, and a call to the [Equals\(Object\)](#) method is equivalent to a call to the [ReferenceEquals](#) method. Reference equality means that the object variables that are compared refer to the same object. The following example illustrates the result of such a comparison. It defines a `Person` class, which is a reference type, and calls the `Person` class constructor to instantiate two new `Person` objects, `person1a` and `person2`, which have the same value. It also assigns `person1a` to another object variable, `person1b`. As the output from the example shows, `person1a` and `person1b` are equal because they reference the same object. However, `person1a` and `person2` are not equal, although they have the same value.

C#

```
using System;

// Define a reference type that does not override Equals.
public class Person
{
    private string personName;

    public Person(string name)
    {
        this.personName = name;
    }

    public override string ToString()
    {
        return this.personName;
    }
}

public class Example1
{
    public static void Main()
    {
```

```

Person person1a = new Person("John");
Person person1b = person1a;
Person person2 = new Person(person1a.ToString());

Console.WriteLine("Calling Equals:");
Console.WriteLine("person1a and person1b: {0}",
person1a.Equals(person1b));
Console.WriteLine("person1a and person2: {0}",
person1a.Equals(person2));

Console.WriteLine("\nCasting to an Object and calling Equals:");
Console.WriteLine("person1a and person1b: {0}", ((object)
person1a).Equals((object) person1b));
Console.WriteLine("person1a and person2: {0}", ((object)
person1a).Equals((object) person2));
}

// The example displays the following output:
//      person1a and person1b: True
//      person1a and person2: False
//
//      Casting to an Object and calling Equals:
//      person1a and person1b: True
//      person1a and person2: False

```

- If the current instance is a value type, the `Equals(Object)` method tests for value equality. Value equality means the following:
  - The two objects are of the same type. As the following example shows, a `Byte` object that has a value of 12 does not equal an `Int32` object that has a value of 12, because the two objects have different run-time types.

C#

```

byte value1 = 12;
int value2 = 12;

object object1 = value1;
object object2 = value2;

Console.WriteLine("{0} ({1}) = {2} ({3}): {4}",
object1, object1.GetType().Name,
object2, object2.GetType().Name,
object1.Equals(object2));

// The example displays the following output:
//      12 (Byte) = 12 (Int32): False

```

- The values of the public and private fields of the two objects are equal. The following example tests for value equality. It defines a `Person` structure, which is

a value type, and calls the `Person` class constructor to instantiate two new `Person` objects, `person1` and `person2`, which have the same value. As the output from the example shows, although the two object variables refer to different objects, `person1` and `person2` are equal because they have the same value for the private `personName` field.

C#

```
using System;

// Define a value type that does not override Equals.
public struct Person3
{
    private string personName;

    public Person3(string name)
    {
        this.personName = name;
    }

    public override string ToString()
    {
        return this.personName;
    }
}

public struct Example3
{
    public static void Main()
    {
        Person3 person1 = new Person3("John");
        Person3 person2 = new Person3("John");

        Console.WriteLine("Calling Equals:");
        Console.WriteLine(person1.Equals(person2));

        Console.WriteLine("\nCasting to an Object and calling
Equals:");
        Console.WriteLine(((object) person1).Equals((object)
person2));
    }
}

// The example displays the following output:
//      Calling Equals:
//      True
//
//      Casting to an Object and calling Equals:
//      True
```

Because the `Object` class is the base class for all types in .NET, the `Object.Equals(Object)` method provides the default equality comparison for all other types. However, types

often override the [Equals](#) method to implement value equality. For more information, see the Notes for Callers and Notes for Inheritors sections.

## Notes for the Windows Runtime

When you call the [Equals\(Object\)](#) method overload on a class in the Windows Runtime, it provides the default behavior for classes that don't override [Equals\(Object\)](#). This is part of the support that .NET provides for the Windows Runtime (see [.NET Support for Windows Store Apps and Windows Runtime](#)). Classes in the Windows Runtime don't inherit [Object](#), and currently don't implement an [Equals\(Object\)](#) method. However, they appear to have [ToString](#), [Equals\(Object\)](#), and [GetHashCode](#) methods when you use them in your C# or Visual Basic code, and .NET provides the default behavior for these methods.

ⓘ Note

Windows Runtime classes that are written in C# or Visual Basic can override the [Equals\(Object\)](#) method overload.

## Notes for callers

Derived classes frequently override the [Object.Equals\(Object\)](#) method to implement value equality. In addition, types also frequently provide an additional strongly typed overload to the [Equals](#) method, typically by implementing the [IEquatable<T>](#) interface. When you call the [Equals](#) method to test for equality, you should know whether the current instance overrides [Object.Equals](#) and understand how a particular call to an [Equals](#) method is resolved. Otherwise, you may be performing a test for equality that is different from what you intended, and the method may return an unexpected value.

The following example provides an illustration. It instantiates three [StringBuilder](#) objects with identical strings, and then makes four calls to [Equals](#) methods. The first method call returns `true`, and the remaining three return `false`.

C#

```
using System;
using System.Text;

public class Example5
{
    public static void Main()
```

```

{
    StringBuilder sb1 = new StringBuilder("building a string...");
    StringBuilder sb2 = new StringBuilder("building a string...");

    Console.WriteLine("sb1.Equals(sb2): {0}", sb1.Equals(sb2));
    Console.WriteLine("((Object) sb1).Equals(sb2): {0}",
                      ((Object) sb1).Equals(sb2));
    Console.WriteLine("Object.Equals(sb1, sb2): {0}",
                      Object.Equals(sb1, sb2));

    Object sb3 = new StringBuilder("building a string...");
    Console.WriteLine("\nsb3.Equals(sb2): {0}", sb3.Equals(sb2));
}

// The example displays the following output:
//     sb1.Equals(sb2): True
//     ((Object) sb1).Equals(sb2): False
//     Object.Equals(sb1, sb2): False
//
//     sb3.Equals(sb2): False

```

In the first case, the strongly typed `StringBuilder.Equals(StringBuilder)` method overload, which tests for value equality, is called. Because the strings assigned to the two `StringBuilder` objects are equal, the method returns `true`. However, `StringBuilder` does not override `Object.Equals(Object)`. Because of this, when the `StringBuilder` object is cast to an `Object`, when a `StringBuilder` instance is assigned to a variable of type `Object`, and when the `Object.Equals(Object, Object)` method is passed two `StringBuilder` objects, the default `Object.Equals(Object)` method is called. Because `StringBuilder` is a reference type, this is equivalent to passing the two `StringBuilder` objects to the `ReferenceEquals` method. Although all three `StringBuilder` objects contain identical strings, they refer to three distinct objects. As a result, these three method calls return `false`.

You can compare the current object to another object for reference equality by calling the `ReferenceEquals` method. In Visual Basic, you can also use the `is` keyword (for example, `If Me Is otherObject Then ...`).

## Notes for inheritors

When you define your own type, that type inherits the functionality defined by the `Equals` method of its base type. The following table lists the default implementation of the `Equals` method for the major categories of types in .NET.

[ ] [Expand table](#)

Type category	Equality defined by	Comments
Class derived directly from <a href="#">Object</a>	<a href="#">Object.Equals(Object)</a>	Reference equality; equivalent to calling <a href="#">Object.ReferenceEquals</a> .
Structure	<a href="#">ValueType.Equals</a>	Value equality; either direct byte-by-byte comparison or field-by-field comparison using reflection.
Enumeration	<a href="#">Enum.Equals</a>	Values must have the same enumeration type and the same underlying value.
Delegate	<a href="#">MulticastDelegate.Equals</a>	Delegates must have the same type with identical invocation lists.
Interface	<a href="#">Object.Equals(Object)</a>	Reference equality.

For a value type, you should always override [Equals](#), because tests for equality that rely on reflection offer poor performance. You can also override the default implementation of [Equals](#) for reference types to test for value equality instead of reference equality and to define the precise meaning of value equality. Such implementations of [Equals](#) return `true` if the two objects have the same value, even if they are not the same instance. The type's implementer decides what constitutes an object's value, but it is typically some or all the data stored in the instance variables of the object. For example, the value of a [String](#) object is based on the characters of the string; the [String.Equals\(Object\)](#) method overrides the [Object.Equals\(Object\)](#) method to return `true` for any two string instances that contain the same characters in the same order.

The following example shows how to override the [Object.Equals\(Object\)](#) method to test for value equality. It overrides the [Equals](#) method for the `Person` class. If `Person` accepted its base class implementation of equality, two `Person` objects would be equal only if they referenced a single object. However, in this case, two `Person` objects are equal if they have the same value for the `Person.Id` property.

C#

```
public class Person
{
    private string idNumber;
    private string personName;

    public Person(string name, string id)
    {
        this.personName = name;
        this.idNumber = id;
    }
}
```

```

public override bool Equals(Object obj)
{
    Person6 personObj = obj as Person6;
    if (personObj == null)
        return false;
    else
        return idNumber.Equals(personObj.idNumber);
}

public override int GetHashCode()
{
    return this.idNumber.GetHashCode();
}
}

public class Example6
{
    public static void Main()
    {
        Person6 p1 = new Person6("John", "63412895");
        Person6 p2 = new Person6("Jack", "63412895");
        Console.WriteLine(p1.Equals(p2));
        Console.WriteLine(Object.Equals(p1, p2));
    }
}

// The example displays the following output:
//      True
//      True

```

In addition to overriding `Equals`, you can implement the `IEquatable<T>` interface to provide a strongly typed test for equality.

The following statements must be true for all implementations of the `Equals(Object)` method. In the list, `x`, `y`, and `z` represent object references that are not `null`.

- `x.Equals(x)` returns `true`.
- `x.Equals(y)` returns the same value as `y.Equals(x)`.
- `x.Equals(y)` returns `true` if both `x` and `y` are `NaN`.
- If `(x.Equals(y) && y.Equals(z))` returns `true`, then `x.Equals(z)` returns `true`.
- Successive calls to `x.Equals(y)` return the same value as long as the objects referenced by `x` and `y` are not modified.
- `x.Equals(null)` returns `false`.

Implementations of `Equals` must not throw exceptions; they should always return a value. For example, if `obj` is `null`, the `Equals` method should return `false` instead of throwing an `ArgumentNullException`.

Follow these guidelines when overriding `Equals(Object)`:

- Types that implement `IComparable` must override `Equals(Object)`.
- Types that override `Equals(Object)` must also override `GetHashCode`; otherwise, hash tables might not work correctly.
- You should consider implementing the `IEquatable<T>` interface to support strongly typed tests for equality. Your `IEquatable<T>.Equals` implementation should return results that are consistent with `Equals`.
- If your programming language supports operator overloading and you overload the equality operator for a given type, you must also override the `Equals(Object)` method to return the same result as the equality operator. This helps ensure that class library code that uses `Equals` (such as `ArrayList` and `Hashtable`) behaves in a manner that is consistent with the way the equality operator is used by application code.

## Guidelines for reference types

The following guidelines apply to overriding `Equals(Object)` for a reference type:

- Consider overriding `Equals` if the semantics of the type are based on the fact that the type represents some value(s).
- Most reference types must not overload the equality operator, even if they override `Equals`. However, if you are implementing a reference type that is intended to have value semantics, such as a complex number type, you must override the equality operator.
- You should not override `Equals` on a mutable reference type. This is because overriding `Equals` requires that you also override the `GetHashCode` method, as discussed in the previous section. This means that the hash code of an instance of a mutable reference type can change during its lifetime, which can cause the object to be lost in a hash table.

## Guidelines for value types

The following guidelines apply to overriding `Equals(Object)` for a value type:

- If you are defining a value type that includes one or more fields whose values are reference types, you should override [Equals\(Object\)](#). The [Equals\(Object\)](#) implementation provided by [ValueType](#) performs a byte-by-byte comparison for value types whose fields are all value types, but it uses reflection to perform a field-by-field comparison of value types whose fields include reference types.
- If you override [Equals](#) and your development language supports operator overloading, you must overload the equality operator.
- You should implement the [IEquatable<T>](#) interface. Calling the strongly typed [IEquatable<T>.Equals](#) method avoids boxing the [obj](#) argument.

## Examples

The following example shows a [Point](#) class that overrides the [Equals](#) method to provide value equality, and a [Point3D](#) class that is derived from [Point](#). Because [Point](#) overrides [Object.Equals\(Object\)](#) to test for value equality, the [Object.Equals\(Object\)](#) method is not called. However, [Point3D.Equals](#) calls [Point.Equals](#) because [Point](#) implements [Object.Equals\(Object\)](#) in a manner that provides value equality.

C#

```
using System;

class Point2
{
    protected int x, y;

    public Point2() : this(0, 0)
    { }

    public Point2(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public override bool Equals(Object obj)
    {
        //Check for null and compare run-time types.
        if ((obj == null) || !this.GetType().Equals(obj.GetType()))
        {
            return false;
        }
        else
        {
            Point2 p = (Point2)obj;
            return (x == p.x) && (y == p.y);
        }
    }
}
```

```

        }

    }

    public override int GetHashCode()
    {
        return (x << 2) ^ y;
    }

    public override string ToString()
    {
        return String.Format("Point2({0}, {1})", x, y);
    }
}

sealed class Point3D : Point2
{
    int z;

    public Point3D(int x, int y, int z) : base(x, y)
    {
        this.z = z;
    }

    public override bool Equals(Object obj)
    {
        Point3D pt3 = obj as Point3D;
        if (pt3 == null)
            return false;
        else
            return base.Equals((Point2)obj) && z == pt3.z;
    }

    public override int GetHashCode()
    {
        return (base.GetHashCode() << 2) ^ z;
    }

    public override String ToString()
    {
        return String.Format("Point2({0}, {1}, {2})", x, y, z);
    }
}

class Example7
{
    public static void Main()
    {
        Point2 point2D = new Point2(5, 5);
        Point3D point3Da = new Point3D(5, 5, 2);
        Point3D point3Db = new Point3D(5, 5, 2);
        Point3D point3Dc = new Point3D(5, 5, -1);

        Console.WriteLine("{0} = {1}: {2}",
                          point2D, point3Da, point2D.Equals(point3Da));
        Console.WriteLine("{0} = {1}: {2}",

```

```

        point2D, point3Db, point2D.Equals(point3Db));
Console.WriteLine("{0} = {1}: {2}",
                  point3Da, point3Db, point3Da.Equals(point3Db));
Console.WriteLine("{0} = {1}: {2}",
                  point3Da, point3Dc, point3Da.Equals(point3Dc));
    }
}

// The example displays the following output:
//      Point2(5, 5) = Point2(5, 5, 2): False
//      Point2(5, 5) = Point2(5, 5, 2): False
//      Point2(5, 5, 2) = Point2(5, 5, 2): True
//      Point2(5, 5, 2) = Point2(5, 5, -1): False

```

The `Point.Equals` method checks to make sure that the `obj` argument is not `null` and that it references an instance of the same type as this object. If either check fails, the method returns `false`.

The `Point.Equals` method calls the `GetType` method to determine whether the run-time types of the two objects are identical. If the method used a check of the form `obj is Point` in C# or `TryCast(obj, Point)` in Visual Basic, the check would return `true` in cases where `obj` is an instance of a derived class of `Point`, even though `obj` and the current instance are not of the same run-time type. Having verified that both objects are of the same type, the method casts `obj` to type `Point` and returns the result of comparing the instance fields of the two objects.

In `Point3D.Equals`, the inherited `Point.Equals` method, which overrides `Object.Equals(Object)`, is invoked before anything else is done. Because `Point3D` is a sealed class (`NotInheritable` in Visual Basic), a check in the form `obj is Point` in C# or `TryCast(obj, Point)` in Visual Basic is adequate to ensure that `obj` is a `Point3D` object. If it is a `Point3D` object, it is cast to a `Point` object and passed to the base class implementation of `Equals`. Only when the inherited `Point.Equals` method returns `true` does the method compare the `z` instance fields introduced in the derived class.

The following example defines a `Rectangle` class that internally implements a rectangle as two `Point` objects. The `Rectangle` class also overrides `Object.Equals(Object)` to provide for value equality.

C#

```

using System;

class Rectangle
{
    private Point a, b;

```

```

    public Rectangle(int upLeftX, int upLeftY, int downRightX, int
downRightY)
    {
        this.a = new Point(upLeftX, upLeftY);
        this.b = new Point(downRightX, downRightY);
    }

    public override bool Equals(Object obj)
    {
        // Perform an equality check on two rectangles (Point object pairs).
        if (obj == null || GetType() != obj.GetType())
            return false;
        Rectangle r = (Rectangle)obj;
        return a.Equals(r.a) && b.Equals(r.b);
    }

    public override int GetHashCode()
    {
        return Tuple.Create(a, b).GetHashCode();
    }

    public override String ToString()
    {
        return String.Format("Rectangle({0}, {1}, {2}, {3})",
a.x, a.y, b.x, b.y);
    }
}

class Point
{
    internal int x;
    internal int y;

    public Point(int X, int Y)
    {
        this.x = X;
        this.y = Y;
    }

    public override bool Equals (Object obj)
    {
        // Performs an equality check on two points (integer pairs).
        if (obj == null || GetType() != obj.GetType()) return false;
        Point p = (Point)obj;
        return (x == p.x) && (y == p.y);
    }

    public override int GetHashCode()
    {
        return Tuple.Create(x, y).GetHashCode();
    }
}

class Example
{

```

```

public static void Main()
{
    Rectangle r1 = new Rectangle(0, 0, 100, 200);
    Rectangle r2 = new Rectangle(0, 0, 100, 200);
    Rectangle r3 = new Rectangle(0, 0, 150, 200);

    Console.WriteLine("{0} = {1}: {2}", r1, r2, r1.Equals(r2));
    Console.WriteLine("{0} = {1}: {2}", r1, r3, r1.Equals(r3));
    Console.WriteLine("{0} = {1}: {2}", r2, r3, r2.Equals(r3));
}
}

// The example displays the following output:
//    Rectangle(0, 0, 100, 200) = Rectangle(0, 0, 100, 200): True
//    Rectangle(0, 0, 100, 200) = Rectangle(0, 0, 150, 200): False
//    Rectangle(0, 0, 100, 200) = Rectangle(0, 0, 150, 200): False

```

Some languages such as C# and Visual Basic support operator overloading. When a type overloads the equality operator, it must also override the [Equals\(Object\)](#) method to provide the same functionality. This is typically accomplished by writing the [Equals\(Object\)](#) method in terms of the overloaded equality operator, as in the following example.

C#

```

using System;

public struct Complex
{
    public double re, im;

    public override bool Equals(Object obj)
    {
        return obj is Complex && this == (Complex)obj;
    }

    public override int GetHashCode()
    {
        return Tuple.Create(re, im).GetHashCode();
    }

    public static bool operator ==(Complex x, Complex y)
    {
        return x.re == y.re && x.im == y.im;
    }

    public static bool operator !=(Complex x, Complex y)
    {
        return !(x == y);
    }

    public override String ToString()
    {

```

```

        return String.Format("({0}, {1})", re, im);
    }

}

class MyClass
{
    public static void Main()
    {
        Complex cmplx1, cmplx2;

        cmplx1.re = 4.0;
        cmplx1.im = 1.0;

        cmplx2.re = 2.0;
        cmplx2.im = 1.0;

        Console.WriteLine("{0} <> {1}: {2}", cmplx1, cmplx2, cmplx1 != cmplx2);
        Console.WriteLine("{0} = {1}: {2}", cmplx1, cmplx2,
cmplx1.Equals(cmplx2));

        cmplx2.re = 4.0;

        Console.WriteLine("{0} = {1}: {2}", cmplx1, cmplx2, cmplx1 == cmplx2);
        Console.WriteLine("{0} = {1}: {2}", cmplx1, cmplx2,
cmplx1.Equals(cmplx2));
    }
}

// The example displays the following output:
//      (4, 1) <> (2, 1): True
//      (4, 1) = (2, 1): False
//      (4, 1) = (4, 1): True
//      (4, 1) = (4, 1): True

```

Because `Complex` is a value type, it cannot be derived from. Therefore, the override to `Equals(Object)` method need not call `GetType` to determine the precise run-time type of each object, but can instead use the `is` operator in C# or the `TypeOf` operator in Visual Basic to check the type of the `obj` parameter.

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



## .NET feedback

.NET is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

# System.Object.Finalize method

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Finalize](#) method is used to perform cleanup operations on unmanaged resources held by the current object before the object is destroyed. The method is protected and therefore is accessible only through this class or through a derived class.

## How finalization works

The [Object](#) class provides no implementation for the [Finalize](#) method, and the garbage collector does not mark types derived from [Object](#) for finalization unless they override the [Finalize](#) method.

If a type does override the [Finalize](#) method, the garbage collector adds an entry for each instance of the type to an internal structure called the finalization queue. The finalization queue contains entries for all the objects in the managed heap whose finalization code must run before the garbage collector can reclaim their memory. The garbage collector then calls the [Finalize](#) method automatically under the following conditions:

- After the garbage collector has discovered that an object is inaccessible, unless the object has been exempted from finalization by a call to the [GC.SuppressFinalize](#) method.
- **On .NET Framework only**, during shutdown of an application domain, unless the object is exempt from finalization. During shutdown, even objects that are still accessible are finalized.

[Finalize](#) is automatically called only once on a given instance, unless the object is re-registered by using a mechanism such as [GC.ReRegisterForFinalize](#) and the [GC.SuppressFinalize](#) method has not been subsequently called.

[Finalize](#) operations have the following limitations:

- The exact time when the finalizer executes is undefined. To ensure deterministic release of resources for instances of your class, implement a [Close](#) method or provide a [IDisposable.Dispose](#) implementation.
- The finalizers of two objects are not guaranteed to run in any specific order, even if one object refers to the other. That is, if Object A has a reference to Object B and

both have finalizers, Object B might have already been finalized when the finalizer of Object A starts.

- The thread on which the finalizer runs is unspecified.

The [Finalize](#) method might not run to completion or might not run at all under the following exceptional circumstances:

- If another finalizer blocks indefinitely (goes into an infinite loop, tries to obtain a lock it can never obtain, and so on). Because the runtime tries to run finalizers to completion, other finalizers might not be called if a finalizer blocks indefinitely.
- If the process terminates without giving the runtime a chance to clean up. In this case, the runtime's first notification of process termination is a `DLL_PROCESS_DETACH` notification.

The runtime continues to finalize objects during shutdown only while the number of finalizable objects continues to decrease.

If [Finalize](#) or an override of [Finalize](#) throws an exception, and the runtime is not hosted by an application that overrides the default policy, the runtime terminates the process and no active `try/finally` blocks or finalizers are executed. This behavior ensures process integrity if the finalizer cannot free or destroy resources.

## Overriding the [Finalize](#) method

You should override [Finalize](#) for a class that uses unmanaged resources, such as file handles or database connections that must be released when the managed object that uses them is discarded during garbage collection. You shouldn't implement a [Finalize](#) method for managed objects because the garbage collector releases managed resources automatically.

### **Important**

If a [SafeHandle](#) object is available that wraps your unmanaged resource, the recommended alternative is to implement the dispose pattern with a safe handle and not override [Finalize](#). For more information, see [The SafeHandle alternative](#) section.

The [Object.Finalize](#) method does nothing by default, but you should override [Finalize](#) only if necessary, and only to release unmanaged resources. Reclaiming memory tends to take much longer if a finalization operation runs, because it requires at least two garbage collections. In addition, you should override the [Finalize](#) method for reference

types only. The common language runtime only finalizes reference types. It ignores finalizers on value types.

The scope of the [Object.Finalize](#) method is `protected`. You should maintain this limited scope when you override the method in your class. By keeping a [Finalize](#) method `protected`, you prevent users of your application from calling an object's [Finalize](#) method directly.

Every implementation of [Finalize](#) in a derived type must call its base type's implementation of [Finalize](#). This is the only case in which application code is allowed to call [Finalize](#). An object's [Finalize](#) method shouldn't call a method on any objects other than that of its base class. This is because the other objects being called could be collected at the same time as the calling object, such as in the case of a common language runtime shutdown.

#### Note

The C# compiler does not allow you to override the [Finalize](#) method. Instead, you provide a finalizer by implementing a [destructor](#) for your class. A C# destructor automatically calls the destructor of its base class.

Visual C++ also provides its own syntax for implementing the [Finalize](#) method. For more information, see the "Destructors and finalizers" section of [How to: Define and Consume Classes and Structs \(C++/CLI\)](#).

Because garbage collection is non-deterministic, you do not know precisely when the garbage collector performs finalization. To release resources immediately, you can also choose to implement the [dispose pattern](#) and the [IDisposable](#) interface. The [IDisposable.Dispose](#) implementation can be called by consumers of your class to free unmanaged resources, and you can use the [Finalize](#) method to free unmanaged resources in the event that the [Dispose](#) method is not called.

[Finalize](#) can take almost any action, including resurrecting an object (that is, making the object accessible again) after it has been cleaned up during garbage collection. However, the object can only be resurrected once; [Finalize](#) cannot be called on resurrected objects during garbage collection.

## The SafeHandle alternative

Creating reliable finalizers is often difficult, because you cannot make assumptions about the state of your application, and because unhandled system exceptions such as

[OutOfMemoryException](#) and [StackOverflowException](#) terminate the finalizer. Instead of implementing a finalizer for your class to release unmanaged resources, you can use an object that is derived from the [System.Runtime.InteropServices.SafeHandle](#) class to wrap your unmanaged resources, and then implement the dispose pattern without a finalizer. The .NET Framework provides the following classes in the [Microsoft.Win32](#) namespace that are derived from [System.Runtime.InteropServices.SafeHandle](#):

- [SafeFileHandle](#) is a wrapper class for a file handle.
- [SafeMemoryMappedFileHandle](#) is a wrapper class for memory-mapped file handles.
- [SafeMemoryMappedViewHandle](#) is a wrapper class for a pointer to a block of unmanaged memory.
- [SafeNCryptKeyHandle](#), [SafeNCryptProviderHandle](#), and [SafeNCryptSecretHandle](#) are wrapper classes for cryptographic handles.
- [SafePipeHandle](#) is a wrapper class for pipe handles.
- [SafeRegistryHandle](#) is a wrapper class for a handle to a registry key.
- [SafeWaitHandle](#) is a wrapper class for a wait handle.

The following example uses the [dispose pattern](#) with safe handles instead of overriding the [Finalize](#) method. It defines a [FileAssociation](#) class that wraps registry information about the application that handles files with a particular file extension. The two registry handles returned as [out](#) parameters by Windows [RegOpenKeyEx](#) function calls are passed to the [SafeRegistryHandle](#) constructor. The type's protected [Dispose](#) method then calls the [SafeRegistryHandle.Dispose](#) method to free these two handles.

C#

```
using Microsoft.Win32.SafeHandles;
using System;
using System.ComponentModel;
using System.IO;
using System.Runtime.InteropServices;

public class FileAssociationInfo : IDisposable
{
    // Private variables.
    private String ext;
    private String openCmd;
    private String args;
    private SafeRegistryHandle hExtHandle, hAppIdHandle;

    // Windows API calls.
    [DllImport("advapi32.dll", CharSet= CharSet.Auto, SetLastError=true)]
    private static extern int RegOpenKeyEx(IntPtr hKey,
                                          String lpSubKey, int ulOptions, int samDesired,
                                          out IntPtr phkResult);
    [DllImport("advapi32.dll", CharSet= CharSet.Unicode, EntryPoint =

```

```
"RegQueryValueExW",
    SetLastError=true)]
private static extern int RegQueryValueEx(IntPtr hKey,
    string lpValueName, int lpReserved, out uint lpType,
    string lpData, ref uint lpcbData);
[DllImport("advapi32.dll", SetLastError = true)]
private static extern int RegSetValueEx(IntPtr hKey,
[MarshalAs(UnmanagedType.LPStr)] string lpValueName,
    int Reserved, uint dwType,
[MarshalAs(UnmanagedType.LPStr)] string lpData,
    int cpData);
[DllImport("advapi32.dll", SetLastError=true)]
private static extern int RegCloseKey(UIntPtr hKey);

// Windows API constants.
private const int HKEY_CLASSES_ROOT = unchecked((int) 0x80000000);
private const int ERROR_SUCCESS = 0;

private const int KEY_QUERY_VALUE = 1;
private const int KEY_SET_VALUE = 0x2;

private const uint REG_SZ = 1;

private const int MAX_PATH = 260;

public FileAssociationInfo(String fileExtension)
{
    int retVal = 0;
    uint lpType = 0;

    if (!fileExtension.StartsWith("."))
        fileExtension = "." + fileExtension;
    ext = fileExtension;

    IntPtr hExtension = IntPtr.Zero;
    // Get the file extension value.
    retVal = RegOpenKeyEx(new IntPtr(HKEY_CLASSES_ROOT), fileExtension, 0,
KEY_QUERY_VALUE, out hExtension);
    if (retVal != ERROR_SUCCESS)
        throw new Win32Exception(retVal);
    // Instantiate the first SafeRegistryHandle.
    hExtHandle = new SafeRegistryHandle(hExtension, true);

    string appId = new string(' ', MAX_PATH);
    uint appIdLength = (uint) appId.Length;
    retVal = RegQueryValueEx(hExtHandle.DangerousGetHandle(),
String.Empty, 0, out lpType, appId, ref appIdLength);
    if (retVal != ERROR_SUCCESS)
        throw new Win32Exception(retVal);
    // We no longer need the hExtension handle.
    hExtHandle.Dispose();

    // Determine the number of characters without the terminating null.
    appId = appId.Substring(0, (int) appIdLength / 2 - 1) +
@"\shell\open\Command";
```

```

// Open the application identifier key.
string exeName = new string(' ', MAX_PATH);
uint exeNameLength = (uint) exeName.Length;
IntPtr hAppId;
retVal = RegOpenKeyEx(new IntPtr(HKEY_CLASSES_ROOT), appId, 0,
KEY_QUERY_VALUE | KEY_SET_VALUE,
                     out hAppId);
if (retVal != ERROR_SUCCESS)
    throw new Win32Exception(retVal);

// Instantiate the second SafeRegistryHandle.
hAppIdHandle = new SafeRegistryHandle(hAppId, true);

// Get the executable name for this file type.
string exePath = new string(' ', MAX_PATH);
uint exePathLength = (uint) exePath.Length;
retVal = RegQueryValueEx(hAppIdHandle.DangerousGetHandle(),
String.Empty, 0, out lpType, exePath, ref exePathLength);
if (retVal != ERROR_SUCCESS)
    throw new Win32Exception(retVal);

// Determine the number of characters without the terminating null.
exePath = exePath.Substring(0, (int) exePathLength / 2 - 1);
// Remove any environment strings.
exePath = Environment.ExpandEnvironmentVariables(exePath);

int position = exePath.IndexOf('%');
if (position >= 0) {
    args = exePath.Substring(position);
    // Remove command line parameters ('%0', etc.).
    exePath = exePath.Substring(0, position).Trim();
}
openCmd = exePath;
}

public String Extension
{ get { return ext; } }

public String Open
{ get { return openCmd; }
set {
    if (hAppIdHandle.Invalid | hAppIdHandle.Closed)
        throw new InvalidOperationException("Cannot write to registry
key.");
    if (! File.Exists(value)) {
        string message = String.Format("{0}' does not exist", value);
        throw new FileNotFoundException(message);
    }
    string cmd = value + " %1";
    int retVal = RegSetValueEx(hAppIdHandle.DangerousGetHandle(),
String.Empty, 0,
                           REG_SZ, value, value.Length + 1);
    if (retVal != ERROR_SUCCESS)
        throw new Win32Exception(retVal);
}
}

```

```
    } }

public void Dispose()
{
    Dispose(disposing: true);
    GC.SuppressFinalize(this);
}

protected void Dispose(bool disposing)
{
    // Ordinarily, we release unmanaged resources here;
    // but all are wrapped by safe handles.

    // Release disposable objects.
    if (disposing) {
        if (hExtHandle != null) hExtHandle.Dispose();
        if (hAppIdHandle != null) hAppIdHandle.Dispose();
    }
}
}
```

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Object.GetHashCode method

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

The [GetHashCode](#) method provides a hash code for algorithms that need quick checks of object equality. A hash code is a numeric value that is used to insert and identify an object in a hash-based collection, such as the [Dictionary<TKey,TValue>](#) class, the [Hashtable](#) class, or a type derived from the [DictionaryBase](#) class.

## ⓘ Note

For information about how hash codes are used in hash tables and for some additional hash code algorithms, see the [Hash Function](#) entry in Wikipedia.

Two objects that are equal return hash codes that are equal. However, the reverse is not true: equal hash codes do not imply object equality, because different (unequal) objects can have identical hash codes. Furthermore, .NET does not guarantee the default implementation of the [GetHashCode](#) method, and the value this method returns may differ between .NET implementations, such as different versions of .NET Framework and .NET Core, and platforms, such as 32-bit and 64-bit platforms. For these reasons, do not use the default implementation of this method as a unique object identifier for hashing purposes. Two consequences follow from this:

- You should not assume that equal hash codes imply object equality.
- You should never persist or use a hash code outside the application domain in which it was created, because the same object may hash across application domains, processes, and platforms.

## ⚠ Warning

A hash code is intended for efficient insertion and lookup in collections that are based on a hash table. A hash code is not a permanent value. For this reason:

- Do not serialize hash code values or store them in databases.
- Do not use the hash code as the key to retrieve an object from a keyed collection.
- Do not send hash codes across application domains or processes. In some cases, hash codes may be computed on a per-process or per-application

domain basis.

- Do not use the hash code instead of a value returned by a cryptographic hashing function if you need a cryptographically strong hash. For cryptographic hashes, use a class derived from the [System.Security.Cryptography.HashAlgorithm](#) or [System.Security.Cryptography.KeyedHashAlgorithm](#) class.
- Do not test for equality of hash codes to determine whether two objects are equal. (Unequal objects can have identical hash codes.) To test for equality, call the [ReferenceEquals](#) or [Equals](#) method.

The [GetHashCode](#) method can be overridden by a derived type. If [GetHashCode](#) is not overridden, hash codes for reference types are computed by calling the [Object.GetHashCode](#) method of the base class, which computes a hash code based on an object's reference; for more information, see [RuntimeHelpers.GetHashCode](#). In other words, two objects for which the [ReferenceEquals](#) method returns `true` have identical hash codes. If value types do not override [GetHashCode](#), the [ValueType.GetHashCode](#) method of the base class uses reflection to compute the hash code based on the values of the type's fields. In other words, value types whose fields have equal values have equal hash codes. For more information about overriding [GetHashCode](#), see the "Notes to Inheritors" section.

#### **Warning**

If you override the [GetHashCode](#) method, you should also override [Equals](#), and vice versa. If your overridden [Equals](#) method returns `true` when two objects are tested for equality, your overridden [GetHashCode](#) method must return the same value for the two objects.

If an object that is used as a key in a hash table does not provide a useful implementation of [GetHashCode](#), you can specify a hash code provider by supplying an [IEqualityComparer](#) implementation to one of the overloads of the [Hashtable](#) class constructor.

## Notes for the Windows Runtime

When you call the [GetHashCode](#) method on a class in the Windows Runtime, it provides the default behavior for classes that don't override [GetHashCode](#). This is part of the support that .NET provides for the Windows Runtime (see [.NET Support for Windows](#)

Store Apps and Windows Runtime). Classes in the Windows Runtime don't inherit [Object](#), and currently don't implement a [GetHashCode](#). However, they appear to have [ToString](#), [Equals\(Object\)](#), and [GetHashCode](#) methods when you use them in your C# or Visual Basic code, and the .NET Framework provides the default behavior for these methods.

① Note

Windows Runtime classes that are written in C# or Visual Basic can override the [GetHashCode](#) method.

## Examples

One of the simplest ways to compute a hash code for a numeric value that has the same or a smaller range than the [Int32](#) type is to simply return that value. The following example shows such an implementation for a [Number](#) structure.

C#

```
using System;

public struct Number
{
    private int n;

    public Number(int value)
    {
        n = value;
    }

    public int Value
    {
        get { return n; }
    }

    public override bool Equals(Object obj)
    {
        if (obj == null || ! (obj is Number))
            return false;
        else
            return n == ((Number) obj).n;
    }

    public override int GetHashCode()
    {
        return n;
    }
}
```

```

        public override string ToString()
    {
        return n.ToString();
    }
}

public class Example1
{
    public static void Main()
    {
        Random rnd = new Random();
        for (int ctr = 0; ctr <= 9; ctr++) {
            int randomN = rnd.Next(Int32.MinValue, Int32.MaxValue);
            Number n = new Number(randomN);
            Console.WriteLine("n = {0,12}, hash code = {1,12}", n,
n.GetHashCode());
        }
    }
}

// The example displays output like the following:
//      n = -634398368, hash code = -634398368
//      n = 2136747730, hash code = 2136747730
//      n = -1973417279, hash code = -1973417279
//      n = 1101478715, hash code = 1101478715
//      n = 2078057429, hash code = 2078057429
//      n = -334489950, hash code = -334489950
//      n = -68958230, hash code = -68958230
//      n = -379951485, hash code = -379951485
//      n = -31553685, hash code = -31553685
//      n = 2105429592, hash code = 2105429592

```

Frequently, a type has multiple data fields that can participate in generating the hash code. One way to generate a hash code is to combine these fields using an `XOR` (eXclusive OR) operation, as shown in the following example.

C#

```

using System;

// A type that represents a 2-D point.
public struct Point2
{
    private int x;
    private int y;

    public Point2(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

```

public override bool Equals(Object obj)
{
    if (! (obj is Point2)) return false;

    Point2 p = (Point2) obj;
    return x == p.x & y == p.y;
}

public override int GetHashCode()
{
    return x ^ y;
}

public class Example3
{
    public static void Main()
    {
        Point2 pt = new Point2(5, 8);
        Console.WriteLine(pt.GetHashCode());

        pt = new Point2(8, 5);
        Console.WriteLine(pt.GetHashCode());
    }
}

// The example displays the following output:
//      13
//      13

```

The previous example returns the same hash code for (n1, n2) and (n2, n1), and so may generate more collisions than are desirable. A number of solutions are available so that hash codes in these cases are not identical. One is to return the hash code of a `Tuple` object that reflects the order of each field. The following example shows a possible implementation that uses the `Tuple<T1,T2>` class. Note, though, that the performance overhead of instantiating a `Tuple` object may significantly impact the overall performance of an application that stores large numbers of objects in hash tables.

C#

```

using System;

public struct Point3
{
    private int x;
    private int y;

    public Point3(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

```

```

public override bool Equals(Object obj)
{
    if (obj is Point3)
    {
        Point3 p = (Point3) obj;
        return x == p.x & y == p.y;
    }
    else
    {
        return false;
    }
}

public override int GetHashCode()
{
    return Tuple.Create(x, y).GetHashCode();
}
}

public class Example
{
    public static void Main()
    {
        Point3 pt = new Point3(5, 8);
        Console.WriteLine(pt.GetHashCode());

        pt = new Point3(8, 5);
        Console.WriteLine(pt.GetHashCode());
    }
}
// The example displays the following output:
//      173
//      269

```

A second alternative solution involves weighting the individual hash codes by left-shifting the hash codes of successive fields by two or more bits. Optimally, bits shifted beyond bit 31 should wrap around rather than be discarded. Since bits are discarded by the left-shift operators in both C# and Visual Basic, this requires creating a left shift-and-wrap method like the following:

C#

```

public int ShiftAndWrap(int value, int positions)
{
    positions = positions & 0x1F;

    // Save the existing bit pattern, but interpret it as an unsigned
    // integer.
    uint number = BitConverter.ToUInt32(BitConverter.GetBytes(value), 0);
    // Preserve the bits to be discarded.
    uint wrapped = number >> (32 - positions);

```

```
// Shift and wrap the discarded bits.
    return BitConverter.ToInt32(BitConverter.GetBytes((number << positions)
| wrapped), 0);
}
```

The following example then uses this shift-and-wrap method to compute the hash code of the `Point` structure used in the previous examples.

C#

```
using System;

public struct Point
{
    private int x;
    private int y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public override bool Equals(Object obj)
    {
        if (!(obj is Point)) return false;

        Point p = (Point) obj;
        return x == p.x & y == p.y;
    }

    public override int GetHashCode()
    {
        return ShiftAndWrap(x.GetHashCode(), 2) ^ y.GetHashCode();
    }

    private int ShiftAndWrap(int value, int positions)
    {
        positions = positions & 0x1F;

        // Save the existing bit pattern, but interpret it as an unsigned
        // integer.
        uint number = BitConverter.ToInt32(BitConverter.GetBytes(value),
0);
        // Preserve the bits to be discarded.
        uint wrapped = number >> (32 - positions);
        // Shift and wrap the discarded bits.
        return BitConverter.ToInt32(BitConverter.GetBytes((number <<
positions) | wrapped), 0);
    }
}

public class Example2
```

```
{  
    public static void Main()  
    {  
        Point pt = new Point(5, 8);  
        Console.WriteLine(pt.GetHashCode());  
  
        pt = new Point(8, 5);  
        Console.WriteLine(pt.GetHashCode());  
    }  
}  
// The example displays the following output:  
//      28  
//      37
```

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Object.ToString method

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

[Object.ToString](#) is a common formatting method in .NET. It converts an object to its string representation so that it is suitable for display. (For information about formatting support in .NET, see [Formatting Types](#).) Default implementations of the [Object.ToString](#) method return the fully qualified name of the object's type.

## ⓘ Important

You may have reached this page by following the link from the member list of another type. That is because that type does not override [Object.ToString](#). Instead, it inherits the functionality of the [Object.ToString](#) method.

Types frequently override the [Object.ToString](#) method to provide a more suitable string representation of a particular type. Types also frequently overload the [Object.ToString](#) method to provide support for format strings or culture-sensitive formatting.

## The default Object.ToString() method

The default implementation of the [ToString](#) method returns the fully qualified name of the type of the [Object](#), as the following example shows.

C#

```
Object obj = new Object();
Console.WriteLine(obj.ToString());

// The example displays the following output:
//      System.Object
```

Because [Object](#) is the base class of all reference types in .NET, this behavior is inherited by reference types that do not override the [ToString](#) method. The following example illustrates this. It defines a class named `Object1` that accepts the default implementation of all [Object](#) members. Its [ToString](#) method returns the object's fully qualified type name.

C#

```
using System;
using Examples;

namespace Examples
{
    public class Object1
    {
    }
}

public class Example5
{
    public static void Main()
    {
        object obj1 = new Object1();
        Console.WriteLine(obj1.ToString());
    }
}
// The example displays the following output:
// Examples.Object1
```

## Override the Object.ToString() method

Types commonly override the [Object.ToString](#) method to return a string that represents the object instance. For example, the base types such as [Char](#), [Int32](#), and [String](#) provide [ToString](#) implementations that return the string form of the value that the object represents. The following example defines a class, `Object2`, that overrides the [ToString](#) method to return the type name along with its value.

C#

```
using System;

public class Object2
{
    private object value;

    public Object2(object value)
    {
        this.value = value;
    }

    public override string ToString()
    {
        return base.ToString() + ":" + value.ToString();
    }
}

public class Example6
```

```

{
    public static void Main()
    {
        Object2 obj2 = new Object2('a');
        Console.WriteLine(obj2.ToString());
    }
}
// The example displays the following output:
//      Object2: a

```

The following table lists the type categories in .NET and indicates whether or not they override the [Object.ToString](#) method.

[] [Expand table](#)

Type category	Overrides <code>Object.ToString()</code>	Behavior
Class	n/a	n/a
Structure	Yes ( <a href="#">ValueType.ToString</a> )	Same as <code>Object.ToString()</code>
Enumeration	Yes ( <a href="#">Enum.ToString</a> ())	The member name
Interface	No	n/a
Delegate	No	n/a

See the Notes to Inheritors section for additional information on overriding [ToString](#).

## Overload the `ToString` method

In addition to overriding the parameterless [Object.ToString](#) method, many types overload the `ToString` method to provide versions of the method that accept parameters. Most commonly, this is done to provide support for variable formatting and culture-sensitive formatting.

The following example overloads the `ToString` method to return a result string that includes the value of various fields of an `Automobile` class. It defines four format strings: G, which returns the model name and year; D, which returns the model name, year, and number of doors; C, which returns the model name, year, and number of cylinders; and A, which returns a string with all four field values.

C#

```
using System;
```

```
public class Automobile
{
    private int _doors;
    private string _cylinders;
    private int _year;
    private string _model;

    public Automobile(string model, int year, int doors,
                      string cylinders)
    {
        _model = model;
        _year = year;
        _doors = doors;
        _cylinders = cylinders;
    }

    public int Doors
    { get { return _doors; } }

    public string Model
    { get { return _model; } }

    public int Year
    { get { return _year; } }

    public string Cylinders
    { get { return _cylinders; } }

    public override string ToString()
    {
        return ToString("G");
    }

    public string ToString(string fmt)
    {
        if (string.IsNullOrEmpty(fmt))
            fmt = "G";

        switch (fmt.ToUpperInvariant())
        {
            case "G":
                return string.Format("{0} {1}", _year, _model);
            case "D":
                return string.Format("{0} {1}, {2} dr.",
                                     _year, _model, _doors);
            case "C":
                return string.Format("{0} {1}, {2}",
                                     _year, _model, _cylinders);
            case "A":
                return string.Format("{0} {1}, {2} dr. {3}",
                                     _year, _model, _doors, _cylinders);
            default:
                string msg = string.Format("'{}' is an invalid format string",
                                           fmt);
                throw new ArgumentException(msg);
        }
    }
}
```

```

        }
    }

public class Example7
{
    public static void Main()
    {
        var auto = new Automobile("Lynx", 2016, 4, "V8");
        Console.WriteLine(auto.ToString());
        Console.WriteLine(auto.ToString("A"));
    }
}
// The example displays the following output:
//      2016 Lynx
//      2016 Lynx, 4 dr. V8

```

The following example calls the overloaded `Decimal.ToString(String, IFormatProvider)` method to display culture-sensitive formatting of a currency value.

C#

```

using System;
using System.Globalization;

public class Example8
{
    public static void Main()
    {
        string[] cultureNames = { "en-US", "en-GB", "fr-FR",
                                  "hr-HR", "ja-JP" };
        Decimal value = 1603.49m;
        foreach (var cultureName in cultureNames) {
            CultureInfo culture = new CultureInfo(cultureName);
            Console.WriteLine("{0}: {1}",
                value.ToString("C2", culture));
        }
    }
}
// The example displays the following output:
//      en-US: $1,603.49
//      en-GB: £1,603.49
//      fr-FR: 1 603,49 €
//      hr-HR: 1.603,49 kn
//      ja-JP: ¥1,603.49

```

For more information on format strings and culture-sensitive formatting, see [Formatting Types](#). For the format strings supported by numeric values, see [Standard Numeric Format Strings](#) and [Custom Numeric Format Strings](#). For the format strings supported by

date and time values, see [Standard Date and Time Format Strings](#) and [Custom Date and Time Format Strings](#).

## Extend the `Object.ToString` method

Because a type inherits the default `Object.ToString` method, you may find its behavior undesirable and want to change it. This is particularly true of arrays and collection classes. While you may expect the `ToString` method of an array or collection class to display the values of its members, it instead displays the type fully qualified type name, as the following example shows.

C#

```
int[] values = { 1, 2, 4, 8, 16, 32, 64, 128 };
Console.WriteLine(values.ToString());

List<int> list = new List<int>(values);
Console.WriteLine(list.ToString());

// The example displays the following output:
//      System.Int32[]
//      System.Collections.Generic.List`1[System.Int32]
```

You have several options to produce the result string that you'd like.

- If the type is an array, a collection object, or an object that implements the `IEnumerable` or `IEnumerable<T>` interfaces, you can enumerate its elements by using the `foreach` statement in C# or the `For Each...Next` construct in Visual Basic.
- If the class is not `sealed` (in C#) or `NotInheritable` (in Visual Basic), you can develop a wrapper class that inherits from the base class whose `Object.ToString` method you want to customize. At a minimum, this requires that you do the following:
  1. Implement any necessary constructors. Derived classes do not inherit their base class constructors.
  2. Override the `Object.ToString` method to return the result string that you'd like.

The following example defines a wrapper class for the `List<T>` class. It overrides the `Object.ToString` method to display the value of each method of the collection rather than the fully qualified type name.

C#

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

public class CList<T> : List<T>
{
    public CList(IEnumerable<T> collection) : base(collection)
    { }

    public CList() : base()
    {}

    public override string ToString()
    {
        string retVal = string.Empty;
        foreach (T item in this) {
            if (string.IsNullOrEmpty(retVal))
                retVal += item.ToString();
            else
                retVal += string.Format(", {0}", item);
        }
        return retVal;
    }
}

public class Example2
{
    public static void Main()
    {
        var list2 = new CList<int>();
        list2.Add(1000);
        list2.Add(2000);
        Console.WriteLine(list2.ToString());
    }
}
// The example displays the following output:
//    1000, 2000
```

- Develop an [extension method](#) that returns the result string that you want. Note that you can't override the default `Object.ToString` method in this way—that is, your extension class (in C#) or module (in Visual Basic) cannot have a parameterless method named `ToString` that's called in place of the original type's `ToString` method. You'll have to provide some other name for your parameterless `ToString` replacement.

The following example defines two methods that extend the `List<T>` class: a parameterless `ToString2` method, and a `ToString` method with a `String` parameter that represents a format string.

C#

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

public static class StringExtensions
{
    public static string ToString2<T>(this List<T> l)
    {
        string retVal = string.Empty;
        foreach (T item in l)
            retVal += string.Format("{0}{1}", string.IsNullOrEmpty(retVal)
? "" : ", ", item);
        return string.IsNullOrEmpty(retVal) ? "{}" : "{ " + retVal + " }";
    }

    public static string ToString<T>(this List<T> l, string fmt)
    {
        string retVal = string.Empty;
        foreach (T item in l) {
            IFormattable ifmt = item as IFormattable;
            if (ifmt != null)
                retVal += string.Format("{0}{1}",
                    string.IsNullOrEmpty(retVal) ?
                    "" : ", ", ifmt.ToString(fmt,
null));
            else
                retVal += ToString2(l);
        }
        return string.IsNullOrEmpty(retVal) ? "{}" : "{ " + retVal + " }";
    }
}

public class Example3
{
    public static void Main()
    {
        List<int> list = new List<int>();
        list.Add(1000);
        list.Add(2000);
        Console.WriteLine(list.ToString2());
        Console.WriteLine(list.ToString("N0"));
    }
}
// The example displays the following output:
//      { 1000, 2000 }
//      { 1,000, 2,000 }
```

# Notes for the Windows Runtime

When you call the `ToString` method on a class in the Windows Runtime, it provides the default behavior for classes that don't override `ToString`. This is part of the support that .NET provides for the Windows Runtime (see [.NET Support for Windows Store Apps and Windows Runtime](#)). Classes in the Windows Runtime don't inherit `Object`, and don't always implement a `ToString`. However, they always appear to have `ToString`, `Equals(Object)`, and `GetHashCode` methods when you use them in your C# or Visual Basic code, and .NET provides a default behavior for these methods.

The common language runtime uses `IStringable.ToString` on a Windows Runtime object before falling back to the default implementation of `Object.ToString`.

## ⓘ Note

Windows Runtime classes that are written in C# or Visual Basic can override the `ToString` method.

## The Windows Runtime and the `IStringable` Interface

The Windows Runtime includes an `IStringable` interface whose single method, `IStringable.ToString`, provides basic formatting support comparable to that provided by `Object.ToString`. To prevent ambiguity, you should not implement `IStringable` on managed types.

When managed objects are called by native code or by code written in languages such as JavaScript or C++/CX, they appear to implement `IStringable`. The common language runtime automatically routes calls from `IStringable.ToString` to `Object.ToString` if `IStringable` is not implemented on the managed object.

## ⚠ Warning

Because the common language runtime auto-implements `IStringable` for all managed types in Windows Store apps, we recommend that you do not provide your own `IStringable` implementation. Implementing `IStringable` may result in unintended behavior when calling `ToString` from the Windows Runtime, C++/CX, or JavaScript.

If you do choose to implement `IStringable` in a public managed type that's exported in a Windows Runtime component, the following restrictions apply:

- You can define the [IStringable](#) interface only in a "class implements" relationship, as follows:

```
C#  
  
public class NewClass : IStringable
```

- You cannot implement [IStringable](#) on an interface.
- You cannot declare a parameter to be of type [IStringable](#).
- [IStringable](#) cannot be the return type of a method, property, or field.
- You cannot hide your [IStringable](#) implementation from base classes by using a method definition such as the following:

```
C#  
  
public class NewClass : IStringable  
{  
    public new string ToString()  
    {  
        return "New ToString in NewClass";  
    }  
}
```

Instead, the [IStringable.ToString](#) implementation must always override the base class implementation. You can hide a [ToString](#) implementation only by invoking it on a strongly typed class instance.

Under a variety of conditions, calls from native code to a managed type that implements [IStringable](#) or hides its [ToString](#) implementation can produce unexpected behavior.

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

# System.Nullable class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Nullable](#) class supports value types that can be assigned `null`.

A type is said to be nullable if it can be assigned a value or can be assigned `null`, which means the type has no value whatsoever. By default, all reference types, such as [String](#), are nullable, but all value types, such as [Int32](#), are not.

In C# and Visual Basic, you mark a value type as nullable by using the `?` notation after the value type. For example, `int?` in C# or `Integer?` in Visual Basic declares an integer value type that can be assigned `null`.

The [Nullable](#) class provides complementary support for the [Nullable<T>](#) structure. The [Nullable](#) class supports obtaining the underlying type of a nullable type, and comparison and equality operations on pairs of nullable types whose underlying value type does not support generic comparison and equality operations.

## Boxing and unboxing

When a nullable type is boxed, the common language runtime automatically boxes the underlying value of the [Nullable<T>](#) object, not the [Nullable<T>](#) object itself. That is, if the [HasValue](#) property is `true`, the contents of the [Value](#) property is boxed.

If the [HasValue](#) property of a nullable type is `false`, the result of the boxing operation is `null`. When the underlying value of a nullable type is unboxed, the common language runtime creates a new [Nullable<T>](#) structure initialized to the underlying value.

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

# Generics in .NET

Article • 02/17/2023

Generics let you tailor a method, class, structure, or interface to the precise data type it acts upon. For example, instead of using the [Hashtable](#) class, which allows keys and values to be of any type, you can use the [Dictionary<TKey, TValue>](#) generic class and specify the types allowed for the key and the value. Among the benefits of generics are increased code reusability and type safety.

## Define and use generics

Generics are classes, structures, interfaces, and methods that have placeholders (type parameters) for one or more of the types that they store or use. A generic collection class might use a type parameter as a placeholder for the type of objects that it stores. The type parameters appear as the types of its fields and the parameter types of its methods. A generic method might use its type parameter as the type of its return value or as the type of one of its formal parameters.

The following code illustrates a simple generic class definition.

```
C#  
  
public class SimpleGenericClass<T>  
{  
    public T Field;  
}
```

When you create an instance of a generic class, you specify the actual types to substitute for the type parameters. This establishes a new generic class, referred to as a constructed generic class, with your chosen types substituted everywhere that the type parameters appear. The result is a type-safe class that is tailored to your choice of types, as the following code illustrates.

```
C#  
  
public static void Main()  
{  
    SimpleGenericClass<string> g = new SimpleGenericClass<string>();  
    g.Field = "A string";  
    //...  
    Console.WriteLine("SimpleGenericClass.Field           = \"{0}\",",  
        g.Field);  
    Console.WriteLine("SimpleGenericClass.Field.GetType() = {0}",
```

```
g.Field.GetType().FullName);  
}
```

## Terminology

The following terms are used to discuss generics in .NET:

- A *generic type definition* is a class, structure, or interface declaration that functions as a template, with placeholders for the types that it can contain or use. For example, the [System.Collections.Generic.Dictionary< TKey, TValue >](#) class can contain two types: keys and values. Because a generic type definition is only a template, you cannot create instances of a class, structure, or interface that is a generic type definition.
- *Generic type parameters*, or *type parameters*, are the placeholders in a generic type or method definition. The [System.Collections.Generic.Dictionary< TKey, TValue >](#) generic type has two type parameters, `TKey` and `TValue`, that represent the types of its keys and values.
- A *constructed generic type*, or *constructed type*, is the result of specifying types for the generic type parameters of a generic type definition.
- A *generic type argument* is any type that is substituted for a generic type parameter.
- The general term *generic type* includes both constructed types and generic type definitions.
- *Covariance* and *contravariance* of generic type parameters enable you to use constructed generic types whose type arguments are more derived (covariance) or less derived (contravariance) than a target constructed type. Covariance and contravariance are collectively referred to as *variance*. For more information, see [Covariance and contravariance](#).
- *Constraints* are limits placed on generic type parameters. For example, you might limit a type parameter to types that implement the [System.Collections.Generic.IComparer< T >](#) generic interface, to ensure that instances of the type can be ordered. You can also constrain type parameters to types that have a particular base class, that have a parameterless constructor, or that are reference types or value types. Users of the generic type cannot substitute type arguments that do not satisfy the constraints.

- A *generic method definition* is a method with two parameter lists: a list of generic type parameters and a list of formal parameters. Type parameters can appear as the return type or as the types of the formal parameters, as the following code shows.

```
C#
T MyGenericMethod<T>(T arg)
{
    T temp = arg;
    //...
    return temp;
}
```

Generic methods can appear on generic or nongeneric types. It's important to note that a method is not generic just because it belongs to a generic type, or even because it has formal parameters whose types are the generic parameters of the enclosing type. A method is generic only if it has its own list of type parameters. In the following code, only method `G` is generic.

```
C#
class A
{
    T G<T>(T arg)
    {
        T temp = arg;
        //...
        return temp;
    }
}
class MyGenericClass<T>
{
    T M(T arg)
    {
        T temp = arg;
        //...
        return temp;
    }
}
```

## Advantages and disadvantages of generics

There are many advantages to using generic collections and delegates:

- Type safety. Generics shift the burden of type safety from you to the compiler. There is no need to write code to test for the correct data type because it is

enforced at compile time. The need for type casting and the possibility of run-time errors are reduced.

- Less code and code is more easily reused. There is no need to inherit from a base type and override members. For example, the [LinkedList<T>](#) is ready for immediate use. For example, you can create a linked list of strings with the following variable declaration:

C#

```
LinkedList<string> llist = new LinkedList<string>();
```

- Better performance. Generic collection types generally perform better for storing and manipulating value types because there is no need to box the value types.
- Generic delegates enable type-safe callbacks without the need to create multiple delegate classes. For example, the [Predicate<T>](#) generic delegate allows you to create a method that implements your own search criteria for a particular type and to use your method with methods of the [Array](#) type such as [Find](#), [FindLast](#), and [FindAll](#).
- Generics streamline dynamically generated code. When you use generics with dynamically generated code you do not need to generate the type. This increases the number of scenarios in which you can use lightweight dynamic methods instead of generating entire assemblies. For more information, see [How to: Define and Execute Dynamic Methods](#) and [DynamicMethod](#).

The following are some limitations of generics:

- Generic types can be derived from most base classes, such as [MarshalByRefObject](#) (and constraints can be used to require that generic type parameters derive from base classes like [MarshalByRefObject](#)). However, .NET does not support context-bound generic types. A generic type can be derived from [ContextBoundObject](#), but trying to create an instance of that type causes a [TypeLoadException](#).
- Enumerations cannot have generic type parameters. An enumeration can be generic only incidentally (for example, because it is nested in a generic type that is defined using Visual Basic, C#, or C++). For more information, see "Enumerations" in [Common Type System](#).
- Lightweight dynamic methods cannot be generic.
- In Visual Basic, C#, and C++, a nested type that is enclosed in a generic type cannot be instantiated unless types have been assigned to the type parameters of

all enclosing types. Another way of saying this is that in reflection, a nested type that is defined using these languages includes the type parameters of all its enclosing types. This allows the type parameters of enclosing types to be used in the member definitions of a nested type. For more information, see "Nested Types" in [MakeGenericType](#).

 **Note**

A nested type that is defined by emitting code in a dynamic assembly or by using the [Ilasm.exe \(IL Assembler\)](#) is not required to include the type parameters of its enclosing types; however, if it does not include them, the type parameters are not in scope in the nested class.

For more information, see "Nested Types" in [MakeGenericType](#).

## Class library and language support

.NET provides a number of generic collection classes in the following namespaces:

- The [System.Collections.Generic](#) namespace contains most of the generic collection types provided by .NET, such as the [List<T>](#) and [Dictionary< TKey, TValue >](#) generic classes.
- The [System.Collections.ObjectModel](#) namespace contains additional generic collection types, such as the [ReadOnlyCollection<T>](#) generic class, that are useful for exposing object models to users of your classes.

Generic interfaces for implementing sort and equality comparisons are provided in the [System](#) namespace, along with generic delegate types for event handlers, conversions, and search predicates.

The [System.Numerics](#) namespace provides generic interfaces for mathematical functionality (available in .NET 7 and later versions). For more information, see [Generic math](#).

Support for generics has been added to the [System.Reflection](#) namespace for examining generic types and generic methods, to [System.Reflection.Emit](#) for emitting dynamic assemblies that contain generic types and methods, and to [System.CodeDom](#) for generating source graphs that include generics.

The common language runtime provides new opcodes and prefixes to support generic types in Microsoft intermediate language (MSIL), including [Stelem](#), [Ldelem](#), [Unbox\\_Any](#),

Constrained, and [Readonly](#).

Visual C++, C#, and Visual Basic all provide full support for defining and using generics. For more information about language support, see [Generic Types in Visual Basic](#), [Introduction to Generics](#), and [Overview of Generics in Visual C++](#).

## Nested types and generics

A type that is nested in a generic type can depend on the type parameters of the enclosing generic type. The common language runtime considers nested types to be generic, even if they do not have generic type parameters of their own. When you create an instance of a nested type, you must specify type arguments for all enclosing generic types.

## Related articles

Title	Description
<a href="#">Generic Collections in .NET</a>	Describes generic collection classes and other generic types in .NET.
<a href="#">Generic Delegates for Manipulating Arrays and Lists</a>	Describes generic delegates for conversions, search predicates, and actions to be taken on elements of an array or collection.
<a href="#">Generic math</a>	Describes how you can perform mathematical operations generically.
<a href="#">Generic Interfaces</a>	Describes generic interfaces that provide common functionality across families of generic types.
<a href="#">Covariance and Contravariance</a>	Describes covariance and contravariance in generic type parameters.
<a href="#">Commonly Used Collection Types</a>	Provides summary information about the characteristics and usage scenarios of the collection types in .NET, including generic types.
<a href="#">When to Use Generic Collections</a>	Describes general rules for determining when to use generic collection types.
<a href="#">How to: Define a Generic Type with Reflection Emit</a>	Explains how to generate dynamic assemblies that include generic types and methods.
<a href="#">Generic Types in Visual Basic</a>	Describes the generics feature for Visual Basic users, including how-to topics for using and defining generic types.

Title	Description
<a href="#">Introduction to Generics</a>	Provides an overview of defining and using generic types for C# users.
<a href="#">Overview of Generics in Visual C++</a>	Describes the generics feature for C++ users, including the differences between generics and templates.

# Reference

- [System.Collections.Generic](#)
- [System.Collections.ObjectModel](#)
- [System.Reflection.Emit.OpCodes](#)

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Generic types overview

Article • 07/23/2022

Developers use generics all the time in .NET, whether implicitly or explicitly. When you use LINQ in .NET, did you ever notice that you're working with `IEnumerable<T>`? Or if you ever saw an online sample of a "generic repository" for talking to databases using Entity Framework, did you see that most methods return `IQueryable<T>`? You may have wondered what the `T` is in these examples and why it's in there.

First introduced in .NET Framework 2.0, generics are essentially a "code template" that allows developers to define `type-safe` data structures without committing to an actual data type. For example, `List<T>` is a `generic collection` that can be declared and used with any type, such as `List<int>`, `List<string>`, or `List<Person>`.

To understand why generics are useful, let's take a look at a specific class before and after adding generics: `ArrayList`. In .NET Framework 1.0, the `ArrayList` elements were of type `Object`. Any element added to the collection was silently converted into an `Object`. The same would happen when reading elements from the list. This process is known as `boxing` and `unboxing`, and it impacts performance. Aside from performance, however, there's no way to determine the type of data in the list at compile time, which makes for some fragile code. Generics solve this problem by defining the type of data each instance of list will contain. For example, you can only add integers to `List<int>` and only add Persons to `List<Person>`.

Generics are also available at run time. The runtime knows what type of data structure you're using and can store it in memory more efficiently.

The following example is a small program that illustrates the efficiency of knowing the data structure type at run time:

C#

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

namespace GenericsExample {
    class Program {
        static void Main(string[] args) {
            //generic list
            List<int> ListGeneric = new List<int> { 5, 9, 1, 4 };
            //non-generic list
            ArrayList ListNonGeneric = new ArrayList { 5, 9, 1, 4 };
        }
    }
}
```

```

    // timer for generic list sort
    Stopwatch s = Stopwatch.StartNew();
    ListGeneric.Sort();
    s.Stop();
    Console.WriteLine($"Generic Sort: {ListGeneric} \n Time taken:
{s.Elapsed.TotalMilliseconds}ms");

    //timer for non-generic list sort
    Stopwatch s2 = Stopwatch.StartNew();
    ListNonGeneric.Sort();
    s2.Stop();
    Console.WriteLine($"Non-Generic Sort: {ListNonGeneric} \n Time
taken: {s2.Elapsed.TotalMilliseconds}ms");
    Console.ReadLine();
}
}
}

```

This program produces output similar to the following:

Console

```

Generic Sort: System.Collections.Generic.List`1[System.Int32]
Time taken: 0.0034ms
Non-Generic Sort: System.Collections.ArrayList
Time taken: 0.2592ms

```

The first thing you can notice here is that sorting the generic list is significantly faster than sorting the non-generic list. You might also notice that the type for the generic list is distinct ([System.Int32]), whereas the type for the non-generic list is generalized. Because the runtime knows the generic `List<int>` is of type `Int32`, it can store the list elements in an underlying integer array in memory, while the non-generic `ArrayList` has to cast each list element to an object. As this example shows, the extra casts take up time and slow down the list sort.

An additional advantage of the runtime knowing the type of your generic is a better debugging experience. When you're debugging a generic in C#, you know what type each element is in your data structure. Without generics, you would have no idea what type each element was.

## See also

- [C# Programming Guide - Generics](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Generic collections in .NET

Article • 09/15/2021

The .NET class library provides a number of generic collection classes in the [System.Collections.Generic](#) and [System.Collections.ObjectModel](#) namespaces. For more detailed information about these classes, see [Commonly Used Collection Types](#).

## System.Collections.Generic

Many of the generic collection types are direct analogs of nongeneric types.

[Dictionary<TKey, TValue>](#) is a generic version of [Hashtable](#); it uses the generic structure [KeyValuePair<TKey, TValue>](#) for enumeration instead of [DictionaryEntry](#).

[List<T>](#) is a generic version of [ArrayList](#). There are generic [Queue<T>](#) and [Stack<T>](#) classes that correspond to the nongeneric versions.

There are generic and nongeneric versions of [SortedList<TKey, TValue>](#). Both versions are hybrids of a dictionary and a list. The [SortedDictionary<TKey, TValue>](#) generic class is a pure dictionary and has no nongeneric counterpart.

The [LinkedList<T>](#) generic class is a true linked list. It has no nongeneric counterpart.

## System.Collections.ObjectModel

The [Collection<T>](#) generic class provides a base class for deriving your own generic collection types. The [ReadOnlyCollection<T>](#) class provides an easy way to produce a read-only collection from any type that implements the [IList<T>](#) generic interface. The [KeyedCollection<TKey, TItem>](#) generic class provides a way to store objects that contain their own keys.

## Other generic types

The [Nullable<T>](#) generic structure allows you to use value types as if they could be assigned `null`. This can be useful when working with database queries, where fields that contain value types can be missing. The generic type parameter can be any value type.

### Note

In C# and Visual Basic, it is not necessary to use [Nullable<T>](#) explicitly because the language has syntax for nullable types. See [Nullable value types \(C# reference\)](#)

## and Nullable value types (Visual Basic).

The `ArraySegment<T>` generic structure provides a way to delimit a range of elements within a one-dimensional, zero-based array of any type. The generic type parameter is the type of the array's elements.

The `EventHandler<TEventArgs>` generic delegate eliminates the need to declare a delegate type to handle events, if your event follows the event-handling pattern used by .NET. For example, suppose you have created a `MyEventArgs` class, derived from `EventArgs`, to hold the data for your event. You can then declare the event as follows:

C#

```
public event EventHandler<MyEventArgs> MyEvent;
```

## See also

- [System.Collections.Generic](#)
- [System.Collections.ObjectModel](#)
- [Generics](#)
- [Generic Delegates for Manipulating Arrays and Lists](#)
- [Generic Interfaces](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Generic Delegates for Manipulating Arrays and Lists

Article • 09/15/2021

This topic provides an overview of generic delegates for conversions, search predicates, and actions to be taken on elements of an array or collection.

## Generic Delegates for Manipulating Arrays and Lists

The [Action<T>](#) generic delegate represents a method that performs some action on an element of the specified type. You can create a method that performs the desired action on the element, create an instance of the [Action<T>](#) delegate to represent that method, and then pass the array and the delegate to the [Array.ForEach](#) static generic method. The method is called for every element of the array.

The [List<T>](#) generic class also provides a [ForEach](#) method that uses the [Action<T>](#) delegate. This method is not generic.

### ⓘ Note

This makes an interesting point about generic types and methods. The [Array.ForEach](#) method must be static ([Shared](#) in Visual Basic) and generic because [Array](#) is not a generic type; the only reason you can specify a type for [Array.ForEach](#) to operate on is that the method has its own type parameter list. By contrast, the nongeneric [List<T>.ForEach](#) method belongs to the generic class [List<T>](#), so it simply uses the type parameter of its class. The class is strongly typed, so the method can be an instance method.

The [Predicate<T>](#) generic delegate represents a method that determines whether a particular element meets criteria you define. You can use it with the following static generic methods of [Array](#) to search for an element or a set of elements: [Exists](#), [Find](#), [FindAll](#), [FindIndex](#), [FindLast](#), [FindLastIndex](#), and [TrueForAll](#).

[Predicate<T>](#) also works with the corresponding nongeneric instance methods of the [List<T>](#) generic class.

The [Comparison<T>](#) generic delegate allows you to provide a sort order for array or list elements that do not have a native sort order, or to override the native sort order.

Create a method that performs the comparison, create an instance of the [Comparison<T>](#) delegate to represent your method, and then pass the array and the delegate to the [Array.Sort<T>\(T\[\], Comparison<T>\)](#) static generic method. The [List<T>](#) generic class provides a corresponding instance method overload, [List<T>.Sort\(Comparison<T>\)](#).

The [Converter<TInput,TOutput>](#) generic delegate allows you to define a conversion between two types, and to convert an array of one type into an array of the other, or to convert a list of one type to a list of the other. Create a method that converts the elements of the existing list to a new type, create a delegate instance to represent the method, and use the [Array.ConvertAll](#) generic static method to produce an array of the new type from the original array, or the [List<T>.ConvertAll<TOutput>\(Converter<T,TOutput>\)](#) generic instance method to produce a list of the new type from the original list.

## Chaining Delegates

Many of the methods that use these delegates return an array or list, which can be passed to another method. For example, if you want to select certain elements of an array, convert those elements to a new type, and save them in a new array, you can pass the array returned by the [FindAll](#) generic method to the [ConvertAll](#) generic method. If the new element type lacks a natural sort order, you can pass the array returned by the [ConvertAll](#) generic method to the [Sort<T>\(T\[\], Comparison<T>\)](#) generic method.

## See also

- [System.Collections.Generic](#)
- [System.Collections.ObjectModel](#)
- [Generics](#)
- [Generic Collections in the .NET](#)
- [Generic Interfaces](#)
- [Covariance and Contravariance](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

# Generic math

Article • 04/21/2023

.NET 7 introduces new math-related generic interfaces to the base class library. The availability of these interfaces means you can constrain a type parameter of a generic type or method to be "number-like". In addition, C# 11 and later lets you define [static virtual interface members](#). Because operators must be declared as `static`, this new C# feature lets operators be declared in the new interfaces for number-like types.

Together, these innovations allow you to perform mathematical operations generically—that is, without having to know the exact type you're working with. For example, if you wanted to write a method that adds two numbers, previously you had to add an overload of the method for each type (for example, `static int Add(int first, int second)` and `static float Add(float first, float second)`). Now you can write a single, generic method, where the type parameter is constrained to be a number-like type. For example:

C#

```
static T Add<T>(T left, T right)
    where T : INumber<T>
{
    return left + right;
}
```

In this method, the type parameter `T` is constrained to be a type that implements the new [INumber<TSelf>](#) interface. [INumber<TSelf>](#) implements the [IAdditionOperators<TSelf,TOther,TResult>](#) interface, which contains the `+` operator. That allows the method to generically add the two numbers. The method can be used with any of .NET's built-in numeric types, because they've all been updated to implement [INumber<TSelf>](#) in .NET 7.

Library authors will benefit most from the generic math interfaces, because they can simplify their code base by removing "redundant" overloads. Other developers will benefit indirectly, because the APIs they consume may start supporting more types.

## The interfaces

The interfaces were designed to be both fine-grained enough that users can define their own interfaces on top, while also being granular enough that they're easy to consume. To that extent, there are a few core numeric interfaces that most users will interact with,

such as [INumber<TSelf>](#) and [IBinaryInteger<TSelf>](#). The more fine-grained interfaces, such as [IAdditionOperators<TSelf,TOther,TResult>](#) and [ITrigonometricFunctions<TSelf>](#), support these types and are available for developers who define their own domain-specific numeric interfaces.

- [Numeric interfaces](#)
- [Operator interfaces](#)
- [Function interfaces](#)
- [Parsing and formatting interfaces](#)

## Numeric interfaces

This section describes the interfaces in [System.Numerics](#) that describe number-like types and the functionality available to them.

[\[\] Expand table](#)

Interface name	Description
<a href="#">IBinaryFloatingPointIEEE754&lt;TSelf&gt;</a>	Exposes APIs common to <i>binary</i> floating-point types <sup>1</sup> that implement the IEEE 754 standard.
<a href="#">IBinaryInteger&lt;TSelf&gt;</a>	Exposes APIs common to binary integers <sup>2</sup> .
<a href="#">IBinaryNumber&lt;TSelf&gt;</a>	Exposes APIs common to binary numbers.
<a href="#">IFloatingPoint&lt;TSelf&gt;</a>	Exposes APIs common to floating-point types.
<a href="#">IFloatingPointIEEE754&lt;TSelf&gt;</a>	Exposes APIs common to floating-point types that implement the IEEE 754 standard.
<a href="#">INumber&lt;TSelf&gt;</a>	Exposes APIs common to comparable number types (effectively the "real" number domain).
<a href="#">INumberBase&lt;TSelf&gt;</a>	Exposes APIs common to all number types (effectively the "complex" number domain).
<a href="#">ISignedNumber&lt;TSelf&gt;</a>	Exposes APIs common to all signed number types (such as the concept of <code>NegativeOne</code> ).
<a href="#">IUnsignedNumber&lt;TSelf&gt;</a>	Exposes APIs common to all unsigned number types.
<a href="#">IAdditiveIdentity&lt;TSelf,TResult&gt;</a>	Exposes the concept of <code>(x + T.AdditiveIdentity) == x</code> .
<a href="#">IMinMaxValue&lt;TSelf&gt;</a>	Exposes the concept of <code>T.MinValue</code> and <code>T.MaxValue</code> .
<a href="#">IMultiplicativeIdentity&lt;TSelf,TResult&gt;</a>	Exposes the concept of <code>(x * T.MultiplicativeIdentity) == x</code> .

<sup>1</sup>The binary floating-point types are `Double` (`double`), `Half`, and `Single` (`float`).

<sup>2</sup>The binary integer types are `Byte` (`byte`), `Int16` (`short`), `Int32` (`int`), `Int64` (`long`), `Int128`, `IntPtr` (`nint`), `SByte` (`sbyte`), `UInt16` (`ushort`), `UInt32` (`uint`), `UInt64` (`ulong`), `UInt128`, and `UIntPtr` (`nuint`).

The interface you're most likely to use directly is `INumber<TSelf>`, which roughly corresponds to a *real* number. If a type implements this interface, it means that a value has a sign (this includes `unsigned` types, which are considered positive) and can be compared to other values of the same type. `INumberBase<TSelf>` confers more advanced concepts, such as *complex* and *imaginary* numbers, for example, the square root of a negative number. Other interfaces, such as `IFloatingPoint<TSelf>`, were created because not all operations make sense for all number types—for example, calculating the floor of a number only makes sense for floating-point types. In the .NET base class library, the floating-point type `Double` implements `IFloatingPoint<TSelf>` but `Int32` doesn't.

Several of the interfaces are also implemented by various other types, including `Char`, `DateOnly`, `DateTime`, `DateTimeOffset`, `Decimal`, `Guid`, `TimeOnly`, and `TimeSpan`.

The following table shows some of the core APIs exposed by each interface.

[ ] [Expand table](#)

Interface	API name	Description
<code>IBinaryInteger&lt;TSelf&gt;</code>	<code>DivRem</code>	Computes the quotient and remainder simultaneously.
	<code>LeadingZeroCount</code>	Counts the number of leading zero bits in the binary representation.
	<code>PopCount</code>	Counts the number of set bits in the binary representation.
	<code>RotateLeft</code>	Rotates bits left, sometimes also called a circular left shift.
	<code>RotateRight</code>	Rotates bits right, sometimes also called a circular right shift.
	<code>TrailingZeroCount</code>	Counts the number of trailing zero bits in the binary representation.
<code>IFloatingPoint&lt;TSelf&gt;</code>	<code>Ceiling</code>	Rounds the value towards positive infinity. +4.5 becomes +5, and -4.5 becomes -4.

Interface	API name	Description
	<code>Floor</code>	Rounds the value towards negative infinity. +4.5 becomes +4, and -4.5 becomes -5.
	<code>Round</code>	Rounds the value using the specified rounding mode.
	<code>Truncate</code>	Rounds the value towards zero. +4.5 becomes +4, and -4.5 becomes -4.
<code>IFloatingPoint&lt;TSelf&gt;</code>	<code>E</code>	Gets a value representing Euler's number for the type.
	<code>Epsilon</code>	Gets the smallest representable value that's greater than zero for the type.
	<code>NaN</code>	Gets a value representing <code>NaN</code> for the type.
	<code>NegativeInfinity</code>	Gets a value representing <code>-Infinity</code> for the type.
	<code>NegativeZero</code>	Gets a value representing <code>-Zero</code> for the type.
	<code>Pi</code>	Gets a value representing <code>Pi</code> for the type.
	<code>PositiveInfinity</code>	Gets a value representing <code>+Infinity</code> for the type.
	<code>Tau</code>	Gets a value representing <code>Tau</code> ( <code>2 * Pi</code> ) for the type.
	(Other)	(Implements the full set of interfaces listed under <a href="#">Function interfaces</a> .)
<code>INumber&lt;TSelf&gt;</code>	<code>Clamp</code>	Restricts a value to no more and no less than the specified min and max value.
	<code>CopySign</code>	Sets the sign of a specified value to the same as another specified value.
	<code>Max</code>	Returns the greater of two values, returning <code>NaN</code> if either input is <code>NaN</code> .
	<code>MaxNumber</code>	Returns the greater of two values, returning the number if one input is <code>NaN</code> .
	<code>Min</code>	Returns the lesser of two values, returning <code>NaN</code> if either input is <code>NaN</code> .

Interface	API name	Description
	<code>MinNumber</code>	Returns the lesser of two values, returning the number if one input is <code>NaN</code> .
	<code>Sign</code>	Returns -1 for negative values, 0 for zero, and +1 for positive values.
<code>INumberBase&lt;TSelf&gt;</code>	<code>One</code>	Gets the value 1 for the type.
	<code>Radix</code>	Gets the radix, or base, for the type. <code>Int32</code> returns 2. <code>Decimal</code> returns 10.
	<code>Zero</code>	Gets the value 0 for the type.
	<code>CreateChecked</code>	Creates a value, throwing an <a href="#">OverflowException</a> if the input can't fit. <sup>1</sup>
	<code>CreateSaturating</code>	Creates a value, clamping to <code>T.MinValue</code> or <code>T.MaxValue</code> if the input can't fit. <sup>1</sup>
	<code>CreateTruncating</code>	Creates a value from another value, wrapping around if the input can't fit. <sup>1</sup>
	<code>IsComplexNumber</code>	Returns true if the value has a non-zero real part and a non-zero imaginary part.
	<code>IsEvenInteger</code>	Returns true if the value is an even integer. 2.0 returns <code>true</code> , and 2.2 returns <code>false</code> .
	<code>IsFinite</code>	Returns true if the value is not infinite and not <code>NaN</code> .
	<code>IsImaginaryNumber</code>	Returns true if the value has a zero real part. This means <code>0</code> is imaginary and <code>1 + 1i</code> isn't.
	<code>IsInfinity</code>	Returns true if the value represents infinity.
	<code>IsInteger</code>	Returns true if the value is an integer. 2.0 and 3.0 return <code>true</code> , and 2.2 and 3.1 return <code>false</code> .
	<code>IsNaN</code>	Returns true if the value represents <code>NaN</code> .
	<code>IsNegative</code>	Returns true if the value is negative. This includes -0.0.
	<code>IsPositive</code>	Returns true if the value is positive. This includes 0 and +0.0.

Interface	API name	Description
	<code>IsRealNumber</code>	Returns true if the value has a zero imaginary part. This means 0 is real as are all <code>INumber&lt;T&gt;</code> types.
	<code>IsZero</code>	Returns true if the value represents zero. This includes 0, +0.0, and -0.0.
	<code>MaxMagnitude</code>	Returns the value with a greater absolute value, returning <code>NaN</code> if either input is <code>NaN</code> .
	<code>MaxMagnitudeNumber</code>	Returns the value with a greater absolute value, returning the number if one input is <code>NaN</code> .
	<code>MinMagnitude</code>	Returns the value with a lesser absolute value, returning <code>NaN</code> if either input is <code>NaN</code> .
	<code>MinMagnitudeNumber</code>	Returns the value with a lesser absolute value, returning the number if one input is <code>NaN</code> .
<code>ISignedNumber&lt;TSelf&gt;</code>	<code>NegativeOne</code>	Gets the value -1 for the type.

<sup>1</sup>To help understand the behavior of the three `Create*` methods, consider the following examples.

Example when given a value that's too large:

- `byte.CreateChecked(384)` will throw an [OverflowException](#).
- `byte.CreateSaturating(384)` returns 255 because 384 is greater than `Byte.MaxValue` (which is 255).
- `byte.CreateTruncating(384)` returns 128 because it takes the lowest 8 bits (384 has a hex representation of `0x0180`, and the lowest 8 bits is `0x80`, which is 128).

Example when given a value that's too small:

- `byte.CreateChecked(-384)` will throw an [OverflowException](#).
- `byte.CreateSaturating(-384)` returns 0 because -384 is smaller than `Byte.MinValue` (which is 0).
- `byte.CreateTruncating(-384)` returns 128 because it takes the lowest 8 bits (384 has a hex representation of `0xFE80`, and the lowest 8 bits is `0x80`, which is 128).

The `Create*` methods also have some special considerations for IEEE 754 floating-point types, like `float` and `double`, as they have the special values `PositiveInfinity`,

`NegativeInfinity`, and `NaN`. All three `Create*` APIs behave as `CreateSaturating`. Also, while `MinValue` and `MaxValue` represent the largest negative/positive "normal" number, the actual minimum and maximum values are `NegativeInfinity` and `PositiveInfinity`, so they clamp to these values instead.

## Operator interfaces

The operator interfaces correspond to the various operators available to the C# language.

- They explicitly don't pair operations such as multiplication and division since that isn't correct for all types. For example, `Matrix4x4 * Matrix4x4` is valid, but `Matrix4x4 / Matrix4x4` isn't valid.
- They typically allow the input and result types to differ to support scenarios such as dividing two integers to obtain a `double`, for example, `3 / 2 = 1.5`, or calculating the average of a set of integers.

[] [Expand table](#)

Interface name	Defined operators
<a href="#">IAdditionOperators&lt;TSelf,TOther,TResult&gt;</a>	<code>x + y</code>
<a href="#">IBitwiseOperators&lt;TSelf,TOther,TResult&gt;</a>	<code>x &amp; y</code> , ' <code>x   y</code> ', <code>x ^ y</code> , and <code>~x</code>
<a href="#">IComparisonOperators&lt;TSelf,TOther,TResult&gt;</a>	<code>x &lt; y</code> , <code>x &gt; y</code> , <code>x &lt;= y</code> , and <code>x &gt;= y</code>
<a href="#">IDecrementOperators&lt;TSelf&gt;</a>	<code>--x</code> and <code>x--</code>
<a href="#">IDivisionOperators&lt;TSelf,TOther,TResult&gt;</a>	<code>x / y</code>
<a href="#">IEqualityOperators&lt;TSelf,TOther,TResult&gt;</a>	<code>x == y</code> and <code>x != y</code>
<a href="#">IIncrementOperators&lt;TSelf&gt;</a>	<code>++x</code> and <code>x++</code>
<a href="#">IModulusOperators&lt;TSelf,TOther,TResult&gt;</a>	<code>x % y</code>
<a href="#">IMultiplyOperators&lt;TSelf,TOther,TResult&gt;</a>	<code>x * y</code>
<a href="#">IShiftOperators&lt;TSelf,TOther,TResult&gt;</a>	<code>x &lt;&lt; y</code> and <code>x &gt;&gt; y</code>
<a href="#">ISubtractionOperators&lt;TSelf,TOther,TResult&gt;</a>	<code>x - y</code>
<a href="#">IUnaryNegationOperators&lt;TSelf,TResult&gt;</a>	<code>-x</code>
<a href="#">IUnaryPlusOperators&lt;TSelf,TResult&gt;</a>	<code>+x</code>

## ⓘ Note

Some of the interfaces define a checked operator in addition to a regular unchecked operator. Checked operators are called in checked contexts and allow a user-defined type to define overflow behavior. If you implement a checked operator, for example, `CheckedSubtraction(TSelf, TOther)`, you must also implement the unchecked operator, for example, `Subtraction(TSelf, TOther)`.

## Function interfaces

The function interfaces define common mathematical APIs that apply more broadly than to a specific [numeric interface](#). These interfaces are all implemented by `IFloatingPointee754<TSelf>`, and may get implemented by other relevant types in the future.

Expand table

Interface name	Description
<a href="#">IExponentialFunctions&lt;TSelf&gt;</a>	Exposes exponential functions supporting <code>e^x</code> , <code>e^x - 1</code> , <code>2^x</code> , <code>2^x - 1</code> , <code>10^x</code> , and <code>10^x - 1</code> .
<a href="#">IHyperbolicFunctions&lt;TSelf&gt;</a>	Exposes hyperbolic functions supporting <code>acosh(x)</code> , <code>asinh(x)</code> , <code>atanh(x)</code> , <code>cosh(x)</code> , <code>sinh(x)</code> , and <code>tanh(x)</code> .
<a href="#">ILogarithmicFunctions&lt;TSelf&gt;</a>	Exposes logarithmic functions supporting <code>ln(x)</code> , <code>ln(x + 1)</code> , <code>log2(x)</code> , <code>log2(x + 1)</code> , <code>log10(x)</code> , and <code>log10(x + 1)</code> .
<a href="#">IPowerFunctions&lt;TSelf&gt;</a>	Exposes power functions supporting <code>x^y</code> .
<a href="#">IRootFunctions&lt;TSelf&gt;</a>	Exposes root functions supporting <code>cbrt(x)</code> and <code>sqrt(x)</code> .
<a href="#">ITrigonometricFunctions&lt;TSelf&gt;</a>	Exposes trigonometric functions supporting <code>acos(x)</code> , <code>asin(x)</code> , <code>atan(x)</code> , <code>cos(x)</code> , <code>sin(x)</code> , and <code>tan(x)</code> .

## Parsing and formatting interfaces

Parsing and formatting are core concepts in programming. They're commonly used when converting user input to a given type or displaying a type to the user. These interfaces are in the [System](#) namespace.

Expand table

Interface name	Description
<a href="#">IParsable&lt;TSelf&gt;</a>	Exposes support for <code>T.Parse(string, IFormatProvider)</code> and <code>T.TryParse(string, IFormatProvider, out TSelf)</code> .
<a href="#">ISpanParsable&lt;TSelf&gt;</a>	Exposes support for <code>T.Parse(ReadOnlySpan&lt;char&gt;, IFormatProvider)</code> and <code>T.TryParse(ReadOnlySpan&lt;char&gt;, IFormatProvider, out TSelf)</code> .
<a href="#">IFormattable<sup>1</sup></a>	Exposes support for <code>value.ToString(string, IFormatProvider)</code> .
<a href="#">ISpanFormattable<sup>1</sup></a>	Exposes support for <code>value.TryFormat(Span&lt;char&gt;, out int, ReadOnlySpan&lt;char&gt;, IFormatProvider)</code> .

<sup>1</sup>This interface isn't new, nor is it generic. However, it's implemented by all number types and represents the inverse operation of [IParsable](#).

For example, the following program takes two numbers as input, reading them from the console using a generic method where the type parameter is constrained to be [IParsable<TSelf>](#). It calculates the average using a generic method where the type parameters for the input and result values are constrained to be [INumber<TSelf>](#), and then displays the result to the console.

C#

```
using System.Globalization;
using System.Numerics;

static TResult Average<T, TResult>(T first, T second)
    where T : INumber<T>
    where TResult : INumber<TResult>
{
    return TResult.CreateChecked( (first + second) / T.CreateChecked(2) );
}

static T ParseInvariant<T>(string s)
    where T : IParsable<T>
{
    return T.Parse(s, CultureInfo.InvariantCulture);
}

Console.WriteLine("First number: ");
var left = ParseInvariant<float>(Console.ReadLine());

Console.WriteLine("Second number: ");
var right = ParseInvariant<float>(Console.ReadLine());

Console.WriteLine($"Result: {Average<float, float>(left, right)}");

/* This code displays output similar to:
```

```
First number: 5.0
Second number: 6
Result: 5.5
*/
```

## See also

- [Generic math in .NET 7 \(blog post\)](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Generic interfaces in .NET

Article • 08/03/2022

This article provides an overview of .NET's generic interfaces that provide common functionality across families of generic types.

Generic interfaces provide type-safe counterparts to nongeneric interfaces for ordering and equality comparisons, and for functionality that's shared by generic collection types. .NET 7 introduces generic interfaces for number-like types, for example, [System.Numerics.INumber<TSelf>](#). These interfaces let you define generic methods that provide mathematical functionality, where the generic type parameter is constrained to be a type that implements a generic, numeric interface.

## ⓘ Note

The type parameters of several generic interfaces are marked covariant or contravariant, providing greater flexibility in assigning and using types that implement these interfaces. For more information, see [Covariance and Contravariance](#).

## Equality and ordering comparisons

- In the [System](#) namespace, the [System.IComparable<T>](#) and [System.IEquatable<T>](#) generic interfaces, like their nongeneric counterparts, define methods for ordering comparisons and equality comparisons, respectively. Types implement these interfaces to provide the ability to perform such comparisons.
- In the [System.Collections.Generic](#) namespace, the [IComparer<T>](#) and [IEqualityComparer<T>](#) generic interfaces offer a way to define an ordering or equality comparison for types that don't implement the [System.IComparable<T>](#) or [System.IEquatable<T>](#) interface. They also provide a way to redefine those relationships for types that do.

These interfaces are used by methods and constructors of many of the generic collection classes. For example, you can pass a generic [IComparer<T>](#) object to the constructor of the [SortedDictionary<TKey, TValue>](#) class to specify a sort order for a type that does not implement generic [System.IComparable<T>](#). There are overloads of the [Array.Sort](#) generic static method and the [List<T>.Sort](#) instance method for sorting arrays and lists using generic [IComparer<T>](#) implementations.

The [Comparer<T>](#) and [EqualityComparer<T>](#) generic classes provide base classes for implementations of the [IComparer<T>](#) and [IEqualityComparer<T>](#) generic interfaces and also provide default ordering and equality comparisons through their respective [Comparer<T>.Default](#) and [EqualityComparer<T>.Default](#) properties.

## Collection functionality

- The [ICollection<T>](#) generic interface is the basic interface for generic collection types. It provides basic functionality for adding, removing, copying, and enumerating elements. [ICollection<T>](#) inherits from both generic [IEnumerable<T>](#) and nongeneric [IEnumerable](#).
- The [IList<T>](#) generic interface extends the [ICollection<T>](#) generic interface with methods for indexed retrieval.
- The [IDictionary<TKey,TValue>](#) generic interface extends the [ICollection<T>](#) generic interface with methods for keyed retrieval. Generic dictionary types in the .NET base class library also implement the nongeneric [IDictionary](#) interface.
- The [IEnumerable<T>](#) generic interface provides a generic enumerator structure. The [IEnumerator<T>](#) generic interface implemented by generic enumerators inherits the nongeneric [IEnumerator](#) interface; the [MoveNext](#) and [Reset](#) members, which do not depend on the type parameter [T](#), appear only on the nongeneric interface. This means that any consumer of the nongeneric interface can also consume the generic interface.

## Mathematical functionality

.NET 7 introduces generic interfaces in the [System.Numerics](#) namespace that describe number-like types and the functionality available to them. The 20 numeric types that the .NET base class library provides, for example, [Int32](#) and [Double](#), have been updated to implement these interfaces. The most prominent of these interfaces is [INumber<TSelf>](#), which roughly corresponds to a "real" number.

For more information about these interfaces, see [Generic math](#).

## See also

- [System.Collections.Generic](#)
- [System.Collections.ObjectModel](#)

- [Generics](#)
- [Generic collections in .NET](#)
- [Generic delegates for manipulating arrays and lists](#)
- [Covariance and contravariance](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

## **.NET feedback**

.NET is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

# Covariance and contravariance in generics

Article • 09/15/2021

*Covariance* and *contravariance* are terms that refer to the ability to use a more derived type (more specific) or a less derived type (less specific) than originally specified. Generic type parameters support covariance and contravariance to provide greater flexibility in assigning and using generic types.

When you're referring to a type system, covariance, contravariance, and invariance have the following definitions. The examples assume a base class named `Base` and a derived class named `Derived`.

- **Covariance**

Enables you to use a more derived type than originally specified.

You can assign an instance of `IEnumerable<Derived>` to a variable of type `IEnumerable<Base>`.

- **Contravariance**

Enables you to use a more generic (less derived) type than originally specified.

You can assign an instance of `Action<Base>` to a variable of type `Action<Derived>`.

- **Invariance**

Means that you can use only the type originally specified. An invariant generic type parameter is neither covariant nor contravariant.

You cannot assign an instance of `List<Base>` to a variable of type `List<Derived>` or vice versa.

Covariant type parameters enable you to make assignments that look much like ordinary [Polymorphism](#), as shown in the following code.

```
C#
```

```
IEnumerable<Derived> d = new List<Derived>();  
IEnumerable<Base> b = d;
```

The `List<T>` class implements the `IEnumerable<T>` interface, so `List<Derived>` (`List(Of Derived)` in Visual Basic) implements `IEnumerable<Derived>`. The covariant type parameter does the rest.

Contravariance, on the other hand, seems counterintuitive. The following example creates a delegate of type `Action<Base>` (`Action(Of Base)` in Visual Basic), and then assigns that delegate to a variable of type `Action<Derived>`.

C#

```
Action<Base> b = (target) => { Console.WriteLine(target.GetType().Name); };
Action<Derived> d = b;
d(new Derived());
```

This seems backward, but it is type-safe code that compiles and runs. The lambda expression matches the delegate it's assigned to, so it defines a method that takes one parameter of type `Base` and that has no return value. The resulting delegate can be assigned to a variable of type `Action<Derived>` because the type parameter `T` of the `Action<T>` delegate is contravariant. The code is type-safe because `T` specifies a parameter type. When the delegate of type `Action<Base>` is invoked as if it were a delegate of type `Action<Derived>`, its argument must be of type `Derived`. This argument can always be passed safely to the underlying method, because the method's parameter is of type `Base`.

In general, a covariant type parameter can be used as the return type of a delegate, and contravariant type parameters can be used as parameter types. For an interface, covariant type parameters can be used as the return types of the interface's methods, and contravariant type parameters can be used as the parameter types of the interface's methods.

Covariance and contravariance are collectively referred to as *variance*. A generic type parameter that is not marked covariant or contravariant is referred to as *invariant*. A brief summary of facts about variance in the common language runtime:

- Variant type parameters are restricted to generic interface and generic delegate types.
- A generic interface or generic delegate type can have both covariant and contravariant type parameters.
- Variance applies only to reference types; if you specify a value type for a variant type parameter, that type parameter is invariant for the resulting constructed type.

- Variance does not apply to delegate combination. That is, given two delegates of types `Action<Derived>` and `Action<Base>` (`Action(Of Derived)` and `Action(Of Base)` in Visual Basic), you cannot combine the second delegate with the first although the result would be type safe. Variance allows the second delegate to be assigned to a variable of type `Action<Derived>`, but delegates can combine only if their types match exactly.
- Starting in C# 9, covariant return types are supported. An overriding method can declare a more derived return type the method it overrides, and an overriding, read-only property can declare a more derived type.

## Generic interfaces with covariant type parameters

Several generic interfaces have covariant type parameters, for example, `IEnumerable<T>`, `IEnumerator<T>`, `IQueryable<T>`, and `IGrouping<TKey, TElement>`. All the type parameters of these interfaces are covariant, so the type parameters are used only for the return types of the members.

The following example illustrates covariant type parameters. The example defines two types: `Base` has a static method named `PrintBases` that takes an `IEnumerable<Base>` (`IEnumerable(Of Base)` in Visual Basic) and prints the elements. `Derived` inherits from `Base`. The example creates an empty `List<Derived>` (`List(Of Derived)` in Visual Basic) and demonstrates that this type can be passed to `PrintBases` and assigned to a variable of type `IEnumerable<Base>` without casting. `List<T>` implements `IEnumerable<T>`, which has a single covariant type parameter. The covariant type parameter is the reason why an instance of `IEnumerable<Derived>` can be used instead of `IEnumerable<Base>`.

C#

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

class Base
{
    public static void PrintBases(IEnumerable<Base> bases)
    {
        foreach(Base b in bases)
        {
            Console.WriteLine(b);
        }
    }
}
```

```

class Derived : Base
{
    public static void Main()
    {
        List<Derived> dlist = new List<Derived>();

        Derived.PrintBases(dlist);
        IEnumerable<Base> bIEnum = dlist;
    }
}

```

## Generic interfaces with contravariant type parameters

Several generic interfaces have contravariant type parameters; for example:

`IComparer<T>`, `IComparable<T>`, and `IEqualityComparer<T>`. These interfaces have only contravariant type parameters, so the type parameters are used only as parameter types in the members of the interfaces.

The following example illustrates contravariant type parameters. The example defines an abstract (`MustInherit` in Visual Basic) `Shape` class with an `Area` property. The example also defines a `ShapeAreaComparer` class that implements `IComparer<Shape>` (`IComparer(Of Shape)` in Visual Basic). The implementation of the `IComparer<T>.Compare` method is based on the value of the `Area` property, so `ShapeAreaComparer` can be used to sort `Shape` objects by area.

The `Circle` class inherits `Shape` and overrides `Area`. The example creates a `SortedSet<T>` of `Circle` objects, using a constructor that takes an `IComparer<Circle>` (`IComparer(Of Circle)` in Visual Basic). However, instead of passing an `IComparer<Circle>`, the example passes a `ShapeAreaComparer` object, which implements `IComparer<Shape>`. The example can pass a comparer of a less derived type (`Shape`) when the code calls for a comparer of a more derived type (`Circle`), because the type parameter of the `IComparer<T>` generic interface is contravariant.

When a new `Circle` object is added to the `SortedSet<Circle>`, the `IComparer<Shape>.Compare` method (`IComparer(Of Shape).Compare` method in Visual Basic) of the `ShapeAreaComparer` object is called each time the new element is compared to an existing element. The parameter type of the method (`Shape`) is less derived than the type that is being passed (`Circle`), so the call is type safe. Contravariance enables `ShapeAreaComparer` to sort a collection of any single type, as well as a mixed collection of types, that derive from `Shape`.

C#

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

abstract class Shape
{
    public virtual double Area { get { return 0; } }
}

class Circle : Shape
{
    private double r;
    public Circle(double radius) { r = radius; }
    public double Radius { get { return r; } }
    public override double Area { get { return Math.PI * r * r; } }
}

class ShapeAreaComparer : System.Collections.Generic.IComparer<Shape>
{
    int IComparer<Shape>.Compare(Shape a, Shape b)
    {
        if (a == null) return b == null ? 0 : -1;
        return b == null ? 1 : a.Area.CompareTo(b.Area);
    }
}

class Program
{
    static void Main()
    {
        // You can pass ShapeAreaComparer, which implements
        IComparer<Shape>,
        // even though the constructor for SortedSet<Circle> expects
        // IComparer<Circle>, because type parameter T of IComparer<T> is
        // contravariant.
        SortedSet<Circle> circlesByArea =
            new SortedSet<Circle>(new ShapeAreaComparer())
            { new Circle(7.2), new Circle(100), null, new Circle(.01) };

        foreach (Circle c in circlesByArea)
        {
            Console.WriteLine(c == null ? "null" : "Circle with area " +
c.Area);
        }
    }
}

/* This code example produces the following output:
null
Circle with area 0.000314159265358979
Circle with area 162.860163162095
```

```
Circle with area 31415.9265358979
*/
```

## Generic delegates with variant type parameters

The `Func` generic delegates, such as `Func<T,TResult>`, have covariant return types and contravariant parameter types. The `Action` generic delegates, such as `Action<T1,T2>`, have contravariant parameter types. This means that the delegates can be assigned to variables that have more derived parameter types and (in the case of the `Func` generic delegates) less derived return types.

### ① Note

The last generic type parameter of the `Func` generic delegates specifies the type of the return value in the delegate signature. It is covariant (`out` keyword), whereas the other generic type parameters are contravariant (`in` keyword).

The following code illustrates this. The first piece of code defines a class named `Base`, a class named `Derived` that inherits `Base`, and another class with a `static` method (`Shared` in Visual Basic) named `MyMethod`. The method takes an instance of `Base` and returns an instance of `Derived`. (If the argument is an instance of `Derived`, `MyMethod` returns it; if the argument is an instance of `Base`, `MyMethod` returns a new instance of `Derived`.) In `Main()`, the example creates an instance of `Func<Base, Derived>` (`Func(Of Base, Derived)` in Visual Basic) that represents `MyMethod`, and stores it in the variable `f1`.

C#

```
public class Base {}
public class Derived : Base {}

public class Program
{
    public static Derived MyMethod(Base b)
    {
        return b as Derived ?? new Derived();
    }

    static void Main()
    {
        Func<Base, Derived> f1 = MyMethod;
```

The second piece of code shows that the delegate can be assigned to a variable of type `Func<Base, Base>` (`Func(Of Base, Base)` in Visual Basic), because the return type is covariant.

C#

```
// Covariant return type.  
Func<Base, Base> f2 = f1;  
Base b2 = f2(new Base());
```

The third piece of code shows that the delegate can be assigned to a variable of type `Func<Derived, Derived>` (`Func(Of Derived, Derived)` in Visual Basic), because the parameter type is contravariant.

C#

```
// Contravariant parameter type.  
Func<Derived, Derived> f3 = f1;  
Derived d3 = f3(new Derived());
```

The final piece of code shows that the delegate can be assigned to a variable of type `Func<Derived, Base>` (`Func(Of Derived, Base)` in Visual Basic), combining the effects of the contravariant parameter type and the covariant return type.

C#

```
// Covariant return type and contravariant parameter type.  
Func<Derived, Base> f4 = f1;  
Base b4 = f4(new Derived());
```

## Variance in non-generic delegates

In the preceding code, the signature of `MyMethod` exactly matches the signature of the constructed generic delegate: `Func<Base, Derived>` (`Func(Of Base, Derived)` in Visual Basic). The example shows that this generic delegate can be stored in variables or method parameters that have more derived parameter types and less derived return types, as long as all the delegate types are constructed from the generic delegate type `Func<T,TResult>`.

This is an important point. The effects of covariance and contravariance in the type parameters of generic delegates are similar to the effects of covariance and contravariance in ordinary delegate binding (see [Variance in Delegates \(C#\)](#) and [Variance in Delegates \(Visual Basic\)](#)). However, variance in delegate binding works with all

delegate types, not just with generic delegate types that have variant type parameters. Furthermore, variance in delegate binding enables a method to be bound to any delegate that has more restrictive parameter types and a less restrictive return type, whereas the assignment of generic delegates works only if both delegate types are constructed from the same generic type definition.

The following example shows the combined effects of variance in delegate binding and variance in generic type parameters. The example defines a type hierarchy that includes three types, from least derived (`Type1`) to most derived (`Type3`). Variance in ordinary delegate binding is used to bind a method with a parameter type of `Type1` and a return type of `Type3` to a generic delegate with a parameter type of `Type2` and a return type of `Type2`. The resulting generic delegate is then assigned to another variable whose generic delegate type has a parameter of type `Type3` and a return type of `Type1`, using the covariance and contravariance of generic type parameters. The second assignment requires both the variable type and the delegate type to be constructed from the same generic type definition, in this case, `Func<T,TResult>`.

C#

```
using System;

public class Type1 {}
public class Type2 : Type1 {}
public class Type3 : Type2 {}

public class Program
{
    public static Type3 MyMethod(Type1 t)
    {
        return t as Type3 ?? new Type3();
    }

    static void Main()
    {
        Func<Type2, Type2> f1 = MyMethod;

        // Covariant return type and contravariant parameter type.
        Func<Type3, Type1> f2 = f1;
        Type1 t1 = f2(new Type3());
    }
}
```

## Define variant generic interfaces and delegates

Visual Basic and C# have keywords that enable you to mark the generic type parameters of interfaces and delegates as covariant or contravariant.

A covariant type parameter is marked with the `out` keyword (`Out` keyword in Visual Basic). You can use a covariant type parameter as the return value of a method that belongs to an interface, or as the return type of a delegate. You cannot use a covariant type parameter as a generic type constraint for interface methods.

### ⓘ Note

If a method of an interface has a parameter that is a generic delegate type, a covariant type parameter of the interface type can be used to specify a contravariant type parameter of the delegate type.

A contravariant type parameter is marked with the `in` keyword (`In` keyword in Visual Basic). You can use a contravariant type parameter as the type of a parameter of a method that belongs to an interface, or as the type of a parameter of a delegate. You can use a contravariant type parameter as a generic type constraint for an interface method.

Only interface types and delegate types can have variant type parameters. An interface or delegate type can have both covariant and contravariant type parameters.

Visual Basic and C# do not allow you to violate the rules for using covariant and contravariant type parameters, or to add covariance and contravariance annotations to the type parameters of types other than interfaces and delegates.

For information and example code, see [Variance in Generic Interfaces \(C#\)](#) and [Variance in Generic Interfaces \(Visual Basic\)](#).

## List of types

The following interface and delegate types have covariant and/or contravariant type parameters.

Type	Covariant type parameters	Contravariant type parameters
<code>Action&lt;T&gt;</code> to <code>Action&lt;T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16&gt;</code>		Yes
<code>Comparison&lt;T&gt;</code>		Yes
<code>Converter&lt;TInput,TOOutput&gt;</code>	Yes	Yes
<code>Func&lt;TResult&gt;</code>		Yes

Type	Covariant Yes type parameters	Contravariant Yes type parameters
Func<T,TResult> to Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>		
IComparable<T>		Yes
Predicate<T>		Yes
IComparer<T>		Yes
IEnumerable<T>	Yes	
IEnumerator<T>	Yes	
IEqualityComparer<T>		Yes
IGrouping<TKey,TElement>	Yes	
IOrderedEnumerable<TElement>	Yes	
IOrderedQueryable<T>	Yes	
IQueryable<T>	Yes	

## See also

- [Covariance and Contravariance \(C#\)](#)
- [Covariance and Contravariance \(Visual Basic\)](#)
- [Variance in Delegates \(C#\)](#)
- [Variance in Delegates \(Visual Basic\)](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Collections and Data Structures

Article • 08/12/2022

Similar data can often be handled more efficiently when stored and manipulated as a collection. You can use the [System.Array](#) class or the classes in the [System.Collections](#), [System.Collections.Generic](#), [System.Collections.Concurrent](#), and [System.Collections.Immutable](#) namespaces to add, remove, and modify either individual elements or a range of elements in a collection.

There are two main types of collections; generic collections and non-generic collections. Generic collections are type-safe at compile time. Because of this, generic collections typically offer better performance. Generic collections accept a type parameter when they're constructed. They don't require that you cast to and from the [Object](#) type when you add or remove items from the collection. In addition, most generic collections are supported in Windows Store apps. Non-generic collections store items as [Object](#), require casting, and most aren't supported for Windows Store app development. However, you might see non-generic collections in older code.

In .NET Framework 4 and later versions, the collections in the [System.Collections.Concurrent](#) namespace provide efficient thread-safe operations for accessing collection items from multiple threads. The immutable collection classes in the [System.Collections.Immutable](#) namespace ([NuGet package](#)) are inherently thread-safe because operations are performed on a copy of the original collection, and the original collection can't be modified.

## Common collection features

All collections provide methods for adding, removing, or finding items in the collection. In addition, all collections that directly or indirectly implement the [ICollection](#) interface or the [ICollection<T>](#) interface share these features:

- **The ability to enumerate the collection**

.NET collections either implement [System.Collections.IEnumerable](#) or [System.Collections.Generic.IEnumerable<T>](#) to enable the collection to be iterated through. An enumerator can be thought of as a movable pointer to any element in the collection. The `foreach`, `in` statement and the [For Each...Next Statement](#) use the enumerator exposed by the [GetEnumerator](#) method and hide the complexity of manipulating the enumerator. In addition, any collection that implements [System.Collections.Generic.IEnumerable<T>](#) is considered a *queryable type* and can be queried with LINQ. LINQ queries provide a common pattern for accessing data. They're typically more concise and readable than standard `foreach` loops and provide filtering, ordering, and grouping capabilities. LINQ queries can also improve performance. For more information, see [LINQ to Objects \(C#\)](#), [LINQ to Objects \(Visual Basic\)](#), [Parallel LINQ \(PLINQ\)](#), [Introduction to LINQ Queries \(C#\)](#), and [Basic Query Operations \(Visual Basic\)](#).

- **The ability to copy the collection contents to an array**

All collections can be copied to an array using the `CopyTo` method. However, the order of the elements in the new array is based on the sequence in which the enumerator returns them. The resulting array is always one-dimensional with a lower bound of zero.

In addition, many collection classes contain the following features:

- **Capacity and Count properties**

The capacity of a collection is the number of elements it can contain. The count of a collection is the number of elements it actually contains. Some collections hide the capacity or the count or both.

Most collections automatically expand in capacity when the current capacity is reached. The memory is reallocated, and the elements are copied from the old collection to the new one. This design reduces the code required to use the collection. However, the performance of the collection might be negatively affected. For example, for `List<T>`, if `Count` is less than `Capacity`, adding an item is an  $O(1)$  operation. If the capacity needs to be increased to accommodate the new element, adding an item becomes an  $O(n)$  operation, where  $n$  is `Count`. The best way to avoid poor performance caused by multiple reallocations is to set the initial capacity to be the estimated size of the collection.

A `BitArray` is a special case; its capacity is the same as its length, which is the same as its count.

- **A consistent lower bound**

The lower bound of a collection is the index of its first element. All indexed collections in the `System.Collections` namespaces have a lower bound of zero, meaning they're 0-indexed. `Array` has a lower bound of zero by default, but a different lower bound can be defined when creating an instance of the `Array` class using `Array.CreateInstance`.

- **Synchronization for access from multiple threads** (`System.Collections` classes only).

Non-generic collection types in the `System.Collections` namespace provide some thread safety with synchronization; typically exposed through the `SyncRoot` and `IsSynchronized` members. These collections aren't thread-safe by default. If you require scalable and efficient multi-threaded access to a collection, use one of the classes in the `System.Collections.Concurrent` namespace or consider using an immutable collection. For more information, see [Thread-Safe Collections](#).

## Choose a collection

In general, you should use generic collections. The following table describes some common collection scenarios and the collection classes you can use for those scenarios. If you're new to generic collections, the following table will help you choose the generic collection that works best for your task:

I want to...	Generic collection options	Non-generic collection options	Thread-safe or immutable collection options
Store items as key/value pairs for quick look-up by key	<a href="#">Dictionary&lt;TKey, TValue&gt;</a>	Hashtable  (A collection of key/value pairs that are organized based on the hash code of the key.)	<a href="#">ConcurrentDictionary&lt;TKey, TValue&gt;</a>  <a href="#">ReadOnlyDictionary&lt;TKey, TValue&gt;</a>  <a href="#">ImmutableDictionary&lt;TKey, TValue&gt;</a>
Access items by index	<a href="#">List&lt;T&gt;</a>	Array  ArrayList	<a href="#">ImmutableList&lt;T&gt;</a>  <a href="#">ImmutableArray</a>
Use items first-in-first-out (FIFO)	<a href="#">Queue&lt;T&gt;</a>	Queue	<a href="#">ConcurrentQueue&lt;T&gt;</a>  <a href="#">ImmutableQueue&lt;T&gt;</a>
Use data Last-In-First-Out (LIFO)	<a href="#">Stack&lt;T&gt;</a>	Stack	<a href="#">ConcurrentStack&lt;T&gt;</a>  <a href="#">ImmutableStack&lt;T&gt;</a>
Access items sequentially	<a href="#">LinkedList&lt;T&gt;</a>	No recommendation	No recommendation
Receive notifications when items are removed or added to the collection. (implements <a href="#">INotifyPropertyChanged</a> and <a href="#">INotifyCollectionChanged</a> )	<a href="#">ObservableCollection&lt;T&gt;</a>	No recommendation	No recommendation
A sorted collection	<a href="#">SortedList&lt;TKey, TValue&gt;</a>	SortedList	<a href="#">ImmutableSortedDictionary&lt;TKey, TValue&gt;</a>  <a href="#">ImmutableSortedSet&lt;T&gt;</a>
A set for mathematical functions	<a href="#">HashSet&lt;T&gt;</a>  <a href="#">SortedSet&lt;T&gt;</a>	No recommendation	<a href="#">ImmutableHashSet&lt;T&gt;</a>  <a href="#">ImmutableSortedSet&lt;T&gt;</a>

## Algorithmic complexity of collections

When choosing a [collection class](#), it's worth considering potential tradeoffs in performance. Use the following table to reference how various mutable collection types compare in algorithmic complexity to their corresponding immutable counterparts. Often immutable collection types are less performant but provide immutability - which is often a valid comparative benefit.

[Expand table](#)

Mutable	Amortized	Worst Case	Immutable	Complexity
<code>Stack&lt;T&gt;.Push</code>	$O(1)$	$O(n)$	<code>ImmutableStack&lt;T&gt;.Push</code>	$O(1)$
<code>Queue&lt;T&gt;.Enqueue</code>	$O(1)$	$O(n)$	<code>ImmutableQueue&lt;T&gt;.Enqueue</code>	$O(1)$
<code>List&lt;T&gt;.Add</code>	$O(1)$	$O(n)$	<code>ImmutableList&lt;T&gt;.Add</code>	$O(\log n)$
<code>List&lt;T&gt;.Item[Int32]</code>	$O(1)$	$O(1)$	<code>ImmutableList&lt;T&gt;.Item[Int32]</code>	$O(\log n)$
<code>List&lt;T&gt;.Enumerator</code>	$O(n)$	$O(n)$	<code>ImmutableList&lt;T&gt;.Enumerator</code>	$O(n)$
<code>HashSet&lt;T&gt;.Add, lookup</code>	$O(1)$	$O(n)$	<code>ImmutableHashSet&lt;T&gt;.Add</code>	$O(\log n)$
<code>SortedSet&lt;T&gt;.Add</code>	$O(\log n)$	$O(n)$	<code>ImmutableSortedSet&lt;T&gt;.Add</code>	$O(\log n)$
<code>Dictionary&lt;T&gt;.Add</code>	$O(1)$	$O(n)$	<code>ImmutableDictionary&lt;T&gt;.Add</code>	$O(\log n)$
<code>Dictionary&lt;T&gt;.lookup</code>	$O(1)$	$O(1) - \text{or strictly } O(n)$	<code>ImmutableDictionary&lt;T&gt;.lookup</code>	$O(\log n)$
<code>SortedDictionary&lt;T&gt;.Add</code>	$O(\log n)$	$O(n \log n)$	<code>ImmutableSortedDictionary&lt;T&gt;.Add</code>	$O(\log n)$

A `List<T>` can be efficiently enumerated using either a `for` loop or a `foreach` loop. An `ImmutableList<T>`, however, does a poor job inside a `for` loop, due to the  $O(\log n)$  time for its indexer. Enumerating an `ImmutableList<T>` using a `foreach` loop is efficient because `ImmutableList<T>` uses a binary tree to store its data instead of an array like `List<T>` uses. An array can be quickly indexed into, whereas a binary tree must be walked down until the node with the desired index is found.

Additionally, `SortedSet<T>` has the same complexity as `ImmutableSortedSet<T>` because they both use binary trees. The significant difference is that `ImmutableSortedSet<T>` uses an immutable binary tree. Since `ImmutableSortedSet<T>` also offers a [System.Collections.Immutable.ImmutableSortedSet<T>.Builder](#) class that allows mutation, you can have both immutability and performance.

## Related articles

[Expand table](#)

Title	Description
<a href="#">Selecting a Collection Class</a>	Describes the different collections and helps you select one for your scenario.
<a href="#">Commonly Used Collection Types</a>	Describes commonly used generic and non-generic collection types such as <a href="#">System.Array</a> , <a href="#">System.Collections.Generic.List&lt;T&gt;</a> , and <a href="#">System.Collections.Generic.Dictionary&lt;TKey, TValue&gt;</a> .
<a href="#">When to Use Generic Collections</a>	Discusses the use of generic collection types.

Title	Description
<a href="#">Comparisons and Sorts Within Collections</a>	Discusses the use of equality comparisons and sorting comparisons in collections.
<a href="#">Sorted Collection Types</a>	Describes sorted collections performance and characteristics.
<a href="#">Hashtable and Dictionary Collection Types</a>	Describes the features of generic and non-generic hash-based dictionary types.
<a href="#">Thread-Safe Collections</a>	Describes collection types such as <code>System.Collections.Concurrent.BlockingCollection&lt;T&gt;</code> and <code>System.Collections.Concurrent.ConcurrentBag&lt;T&gt;</code> that support safe and efficient concurrent access from multiple threads.
<a href="#">System.Collections.Immutable</a>	Introduces the immutable collections and provides links to the collection types.

## Reference

- [System.Array](#)
- [System.Collections](#)
- [System.Collections.Concurrent](#)
- [System.Collections.Generic](#)
- [System.Collections.Specialized](#)
- [System.Linq](#)
- [System.Collections.Immutable](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

# Selecting a Collection Class

Article • 09/15/2021

Be sure to choose your collection class carefully. Using the wrong type can restrict your use of the collection.

## ⓘ Important

Avoid using the types in the [System.Collections](#) namespace. The generic and concurrent versions of the collections are recommended because of their greater type safety and other improvements.

Consider the following questions:

- Do you need a sequential list where the element is typically discarded after its value is retrieved?
  - If yes, consider using the [Queue](#) class or the [Queue<T>](#) generic class if you need first-in, first-out (FIFO) behavior. Consider using the [Stack](#) class or the [Stack<T>](#) generic class if you need last-in, first-out (LIFO) behavior. For safe access from multiple threads, use the concurrent versions, [ConcurrentQueue<T>](#) and [ConcurrentStack<T>](#). For immutability, consider the immutable versions, [ImmutableQueue<T>](#) and [ImmutableStack<T>](#).
  - If not, consider using the other collections.
- Do you need to access the elements in a certain order, such as FIFO, LIFO, or random?
  - The [Queue](#) class, as well as the [Queue<T>](#), [ConcurrentQueue<T>](#), and [ImmutableQueue<T>](#) generic classes all offer FIFO access. For more information, see [When to Use a Thread-Safe Collection](#).
  - The [Stack](#) class, as well as the [Stack<T>](#), [ConcurrentStack<T>](#), and [ImmutableStack<T>](#) generic classes all offer LIFO access. For more information, see [When to Use a Thread-Safe Collection](#).
  - The [LinkedList<T>](#) generic class allows sequential access either from the head to the tail, or from the tail to the head.
- Do you need to access each element by index?

- The [ArrayList](#) and [StringCollection](#) classes and the [List<T>](#) generic class offer access to their elements by the zero-based index of the element. For immutability, consider the immutable generic versions, [ImmutableArray<T>](#) and [ImmutableList<T>](#).
  - The [Hashtable](#), [SortedList](#), [ListDictionary](#), and [StringDictionary](#) classes, and the [Dictionary<TKey,TValue>](#) and [SortedDictionary<TKey,TValue>](#) generic classes offer access to their elements by the key of the element. Additionally, there are immutable versions of several corresponding types: [ImmutableHashSet<T>](#), [ImmutableDictionary<TKey,TValue>](#), [ImmutableSortedSet<T>](#), and [ImmutableSortedDictionary<TKey,TValue>](#).
  - The [NameObjectCollectionBase](#) and [NameValuePairCollection](#) classes, and the [KeyedCollection<TKey,TItem>](#) and [SortedList<TKey,TValue>](#) generic classes offer access to their elements by either the zero-based index or the key of the element.
- Will each element contain one value, a combination of one key and one value, or a combination of one key and multiple values?
    - One value: Use any of the collections based on the [IList](#) interface or the [IList<T>](#) generic interface. For an immutable option, consider the [ImmutableList<T>](#) generic interface.
    - One key and one value: Use any of the collections based on the [IDictionary](#) interface or the [IDictionary<TKey,TValue>](#) generic interface. For an immutable option, consider the [ImmutableSet<T>](#) or [ImmutableDictionary<TKey,TValue>](#) generic interfaces.
    - One value with embedded key: Use the [KeyedCollection<TKey,TItem>](#) generic class.
    - One key and multiple values: Use the [NameValuePairCollection](#) class.
  - Do you need to sort the elements differently from how they were entered?
    - The [Hashtable](#) class sorts its elements by their hash codes.
    - The [SortedList](#) class, and the [SortedList<TKey,TValue>](#) and [SortedDictionary<TKey,TValue>](#) generic classes sort their elements by the key. The sort order is based on the implementation of the [IComparer](#) interface for the [SortedList](#) class and on the implementation of the [IComparer<T>](#) generic interface for the [SortedList<TKey,TValue>](#) and [SortedDictionary<TKey,TValue>](#) generic classes. Of the two generic types, [SortedDictionary<TKey,TValue>](#) offers

better performance than `SortedList<TKey, TValue>`, while `SortedList<TKey, TValue>` consumes less memory.

- `ArrayList` provides a `Sort` method that takes an `IComparer` implementation as a parameter. Its generic counterpart, the `List<T>` generic class, provides a `Sort` method that takes an implementation of the `IComparer<T>` generic interface as a parameter.
- Do you need fast searches and retrieval of information?
  - `ListDictionary` is faster than `Hashtable` for small collections (10 items or fewer). The `Dictionary<TKey, TValue>` generic class provides faster lookup than the `SortedDictionary<TKey, TValue>` generic class. The multi-threaded implementation is `ConcurrentDictionary<TKey, TValue>`. `ConcurrentBag<T>` provides fast multi-threaded insertion for unordered data. For more information about both multi-threaded types, see [When to Use a Thread-Safe Collection](#).
- Do you need collections that accept only strings?
  - `StringCollection` (based on `IList`) and `StringDictionary` (based on `IDictionary`) are in the `System.Collections.Specialized` namespace.
  - In addition, you can use any of the generic collection classes in the `System.Collections.Generic` namespace as strongly typed string collections by specifying the `String` class for their generic type arguments. For example, you can declare a variable to be of type `List<String>` or `Dictionary<String, String>`.

## LINQ to Objects and PLINQ

LINQ to Objects enables developers to use LINQ queries to access in-memory objects as long as the object type implements `IEnumerable` or `IEnumerable<T>`. LINQ queries provide a common pattern for accessing data, are typically more concise and readable than standard `foreach` loops, and provide filtering, ordering, and grouping capabilities. For more information, see [LINQ to Objects \(C#\)](#) and [LINQ to Objects \(Visual Basic\)](#).

PLINQ provides a parallel implementation of LINQ to Objects that can offer faster query execution in many scenarios, through more efficient use of multi-core computers. For more information, see [Parallel LINQ \(PLINQ\)](#).

## See also

- [System.Collections](#)
- [System.Collections.Specialized](#)

- [System.Collections.Generic](#)
- [Thread-Safe Collections](#)

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Commonly used collection types

Article • 09/15/2021

Collection types represent different ways to collect data, such as hash tables, queues, stacks, bags, dictionaries, and lists.

All collections are based on the [ICollection](#) or [ICollection<T>](#) interfaces, either directly or indirectly. [IList](#) and [IDictionary](#) and their generic counterparts all derive from these two interfaces.

In collections based on [IList](#) or directly on [ICollection](#), every element contains only a value. These types include:

- [Array](#)
- [ArrayList](#)
- [List<T>](#)
- [Queue](#)
- [ConcurrentQueue<T>](#)
- [Stack](#)
- [ConcurrentStack<T>](#)
- [LinkedList<T>](#)

In collections based on the [IDictionary](#) interface, every element contains both a key and a value. These types include:

- [Hashtable](#)
- [SortedList](#)
- [SortedList<TKey,TValue>](#)
- [Dictionary<TKey,TValue>](#)
- [ConcurrentDictionary<TKey,TValue>](#)

The [KeyedCollection<TKey,TItem>](#) class is unique because it is a list of values with keys embedded within the values. As a result, it behaves both like a list and like a dictionary.

When you need efficient multi-threaded collection access, use the generic collections in the [System.Collections.Concurrent](#) namespace.

The [Queue](#) and [Queue<T>](#) classes provide first-in-first-out lists. The [Stack](#) and [Stack<T>](#) classes provide last-in-first-out lists.

## Strong typing

Generic collections are the best solution to strong typing. For example, adding an element of any type other than an `Int32` to a `List<Int32>` collection causes a compile-time error. However, if your language does not support generics, the `System.Collections` namespace includes abstract base classes that you can extend to create collection classes that are strongly typed. These base classes include:

- `CollectionBase`
- `ReadOnlyCollectionBase`
- `DictionaryBase`

## How collections vary

Collections vary in how they store, sort, and compare elements, and how they perform searches.

The `SortedList` class and the `SortedList<TKey,TValue>` generic class provide sorted versions of the `Hashtable` class and the `Dictionary<TKey,TValue>` generic class.

All collections use zero-based indexes except `Array`, which allows arrays that are not zero-based.

You can access the elements of a `SortedList` or a `KeyedCollection<TKey,TItem>` by either the key or the element's index. You can only access the elements of a `Hashtable` or a `Dictionary<TKey,TValue>` by the element's key.

## Use LINQ with collection types

The LINQ to Objects feature provides a common pattern for accessing in-memory objects of any type that implements `IEnumerable` or `IEnumerable<T>`. LINQ queries have several benefits over standard constructs like `foreach` loops:

- They are concise and easier to understand.
- They can filter, order, and group data.
- They can improve performance.

For more information, see [LINQ to Objects \(C#\)](#), [LINQ to Objects \(Visual Basic\)](#), and [Parallel LINQ \(PLINQ\)](#).

## Related topics

Title	Description
<a href="#">Collections and Data Structures</a>	Discusses the various collection types available in .NET, including stacks, queues, lists, arrays, and dictionaries.
<a href="#">Hashtable and Dictionary Collection Types</a>	Describes the features of generic and nongeneric hash-based dictionary types.
<a href="#">Sorted Collection Types</a>	Describes classes that provide sorting functionality for lists and sets.
<a href="#">Generics</a>	Describes the generics feature, including the generic collections, delegates, and interfaces provided by .NET. Provides links to feature documentation for C#, Visual Basic, and Visual C++, and to supporting technologies such as reflection.

## Reference

[System.Collections](#)

[System.Collections.Generic](#)

[System.Collections.ICollection](#)

[System.Collections.Generic.ICollection<T>](#)

[System.Collections.IList](#)

[System.Collections.Generic.IList<T>](#)

[System.Collections.IDictionary](#)

[System.Collections.Generic.IDictionary<TKey, TValue>](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

**.NET feedback**

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# When to use generic collections

Article • 09/15/2021

Using generic collections gives you the automatic benefit of type safety without having to derive from a base collection type and implement type-specific members. Generic collection types also generally perform better than the corresponding nongeneric collection types (and better than types that are derived from nongeneric base collection types) when the collection elements are value types, because with generics, there's no need to box the elements.

For programs that target .NET Standard 1.0 or later, use the generic collection classes in the [System.Collections.Concurrent](#) namespace when multiple threads might be adding or removing items from the collection concurrently. Additionally, when immutability is desired, consider the generic collection classes in the [System.Collections.Immutable](#) namespace.

The following generic types correspond to existing collection types:

- `List<T>` is the generic class that corresponds to [ArrayList](#).
- `Dictionary<TKey,TValue>` and `ConcurrentDictionary<TKey,TValue>` are the generic classes that correspond to [Hashtable](#).
- `Collection<T>` is the generic class that corresponds to [CollectionBase](#). `Collection<T>` can be used as a base class, but unlike `CollectionBase`, it is not abstract, which makes it much easier to use.
- `ReadOnlyCollection<T>` is the generic class that corresponds to [ReadOnlyCollectionBase](#). `ReadOnlyCollection<T>` is not abstract and has a constructor that makes it easy to expose an existing `List<T>` as a read-only collection.
- The `Queue<T>`, `ConcurrentQueue<T>`, `ImmutableQueue<T>`, `ImmutableArray<T>`, `SortedList<TKey,TValue>`, and `ImmutableSortedSet<T>` generic classes correspond to the respective nongeneric classes with the same names.

## Additional Types

Several generic collection types do not have nongeneric counterparts. They include the following:

- `LinkedList<T>` is a general-purpose linked list that provides  $O(1)$  insertion and removal operations.
- `SortedDictionary<TKey, TValue>` is a sorted dictionary with  $O(\log n)$  insertion and retrieval operations, which makes it a useful alternative to `SortedList<TKey, TValue>`.
- `KeyedCollection<TKey, TItem>` is a hybrid between a list and a dictionary, which provides a way to store objects that contain their own keys.
- `BlockingCollection<T>` implements a collection class with bounding and blocking functionality.
- `ConcurrentBag<T>` provides fast insertion and removal of unordered elements.

## Immutable builders

When you desire immutability functionality in your app, the `System.Collections.Immutable` namespace offers generic collection types you can use. All of the immutable collection types offer `Builder` classes that can optimize performance when you're performing multiple mutations. The `Builder` class batches operations in a mutable state. When all mutations have been completed, call the `ToImmutable` method to "freeze" all nodes and create an immutable generic collection, for example, an `ImmutableList<T>`.

The `Builder` object can be created by calling the nongeneric `CreateBuilder()` method. From a `Builder` instance, you can call `ToImmutable()`. Likewise, from the `Immutable*` collection, you can call `ToBuilder()` to create a builder instance from the generic immutable collection. The following are the various `Builder` types.

- `ImmutableArray<T>.Builder`
- `ImmutableDictionary<TKey, TValue>.Builder`
- `ImmutableHashSet<T>.Builder`
- `ImmutableList<T>.Builder`
- `ImmutableSortedDictionary<TKey, TValue>.Builder`
- `ImmutableSortedSet<T>.Builder`

## LINQ to Objects

The LINQ to Objects feature enables you to use LINQ queries to access in-memory objects as long as the object type implements the `System.Collections.IEnumerable` or `System.Collections.Generic.IEnumerable<T>` interface. LINQ queries provide a common

pattern for accessing data; are typically more concise and readable than standard `foreach` loops; and provide filtering, ordering, and grouping capabilities. LINQ queries can also improve performance. For more information, see [LINQ to Objects \(C#\)](#), [LINQ to Objects \(Visual Basic\)](#), and [Parallel LINQ \(PLINQ\)](#).

## Additional Functionality

Some of the generic types have functionality that is not found in the nongeneric collection types. For example, the `List<T>` class, which corresponds to the nongeneric `ArrayList` class, has a number of methods that accept generic delegates, such as the `Predicate<T>` delegate that allows you to specify methods for searching the list, the `Action<T>` delegate that represents methods that act on each element of the list, and the `Converter<TInput,TOutput>` delegate that lets you define conversions between types.

The `List<T>` class allows you to specify your own `IComparer<T>` generic interface implementations for sorting and searching the list. The `SortedDictionary< TKey, TValue >` and `SortedList< TKey, TValue >` classes also have this capability. In addition, these classes let you specify comparers when the collection is created. In similar fashion, the `Dictionary< TKey, TValue >` and `KeyedCollection< TKey, TItem >` classes let you specify your own equality comparers.

## See also

- [Collections and Data Structures](#)
- [Commonly Used Collection Types](#)
- [Generics](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Comparisons and sorts within collections

Article • 09/15/2021

The [System.Collections](#) classes perform comparisons in almost all the processes involved in managing collections, whether searching for the element to remove or returning the value of a key-and-value pair.

Collections typically utilize an equality comparer and/or an ordering comparer. Two constructs are used for comparisons.

## Check for equality

Methods such as `Contains`, `IndexOf`, `LastIndexOf`, and `Remove` use an equality comparer for the collection elements. If the collection is generic, then items are compared for equality according to the following guidelines:

- If type T implements the [IEquatable<T>](#) generic interface, then the equality comparer is the `Equals` method of that interface.
- If type T does not implement [IEquatable<T>](#), `Object.Equals` is used.

In addition, some constructor overloads for dictionary collections accept an [IEqualityComparer<T>](#) implementation, which is used to compare keys for equality. For an example, see the [Dictionary<TKey,TValue>](#) constructor.

## Determine sort order

Methods such as `BinarySearch` and `Sort` use an ordering comparer for the collection elements. The comparisons can be between elements of the collection, or between an element and a specified value. For comparing objects, there is the concept of a `default comparer` and an `explicit comparer`.

The default comparer relies on at least one of the objects being compared to implement the [IComparable](#) interface. It is a good practice to implement [IComparable](#) on all classes which are used as values in a list collection or as keys in a dictionary collection. For a generic collection, equality comparison is determined according to the following:

- If type T implements the [System.IComparable<T>](#) generic interface, then the default comparer is the `IComparable<T>.CompareTo(T)` method of that interface

- If type T implements the non-generic `System.IComparable` interface, then the default comparer is the `IComparable.CompareTo(Object)` method of that interface.
- If type T doesn't implement either interface, then there is no default comparer, and a comparer or comparison delegate must be provided explicitly.

To provide explicit comparisons, some methods accept an `IComparer` implementation as a parameter. For example, the `List<T>.Sort` method accepts an `System.Collections.Generic.IComparer<T>` implementation.

The current culture setting of the system can affect the comparisons and sorts within a collection. By default, the comparisons and sorts in the `Collections` classes are culture-sensitive. To ignore the culture setting and therefore obtain consistent comparison and sorting results, use the `InvariantCulture` with member overloads that accept a `CultureInfo`. For more information, see [Perform culture-insensitive string operations in collections](#) and [Perform culture-insensitive string operations in arrays](#).

## Equality and sort example

The following code demonstrates an implementation of `IEquatable<T>` and `IComparable<T>` on a simple business object. In addition, when the object is stored in a list and sorted, you will see that calling the `Sort()` method results in the use of the default comparer for the `Part` type, and the `Sort(Comparison<T>)` method implemented by using an anonymous method.

C#

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

// Simple business object. A PartId is used to identify the
// type of part but the part name can change.
public class Part : IEquatable<Part>, IComparable<Part>
{
    public string PartName { get; set; }

    public int PartId { get; set; }

    public override string ToString() =>
        $"ID: {PartId}    Name: {PartName}";

    public override bool Equals(object obj) =>
        (obj is Part part)
            ? Equals(part)
            : false;

    public int SortByNameAscending(string name1, string name2) =>

```

```

        name1?.CompareTo(name2) ?? 1;

    // Default comparer for Part type.
    // A null value means that this object is greater.
    public int CompareTo(Part comparePart) =>
        comparePart == null ? 1 : PartId.CompareTo(comparePart.PartId);

    public override int GetHashCode() => PartId;

    public bool Equals(Part other) =>
        other is null ? false : PartId.Equals(other.PartId);

    // Should also override == and != operators.
}

public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        var parts = new List<Part>
        {
            // Add parts to the list.
            new Part { PartName = "regular seat", PartId = 1434 },
            new Part { PartName = "crank arm", PartId = 1234 },
            new Part { PartName = "shift lever", PartId = 1634 },
            // Name intentionally left null.
            new Part { PartId = 1334 },
            new Part { PartName = "banana seat", PartId = 1444 },
            new Part { PartName = "cassette", PartId = 1534 }
        };

        // Write out the parts in the list. This will call the overridden
        // ToString method in the Part class.
        Console.WriteLine("\nBefore sort:");
        parts.ForEach(Console.WriteLine);

        // Call Sort on the list. This will use the
        // default comparer, which is the Compare method
        // implemented on Part.
        parts.Sort();

        Console.WriteLine("\nAfter sort by part number:");
        parts.ForEach(Console.WriteLine);

        // This shows calling the Sort(Comparison<T> comparison) overload
using
        // a lambda expression as the Comparison<T> delegate.
        // This method treats null as the lesser of two values.
        parts.Sort((Part x, Part y) =>
            x.PartName == null && y.PartName == null
            ? 0
            : x.PartName == null
            ? -1
            : y.PartName == null
            ? 1
            : x.PartName.CompareTo(y.PartName));
    }
}

```

```

        ? 1
        : x.PartName.CompareTo(y.PartName));

Console.WriteLine("\nAfter sort by name:");
parts.ForEach(Console.WriteLine);

/*
    Before sort:
ID: 1434  Name: regular seat
ID: 1234  Name: crank arm
ID: 1634  Name: shift lever
ID: 1334  Name:
ID: 1444  Name: banana seat
ID: 1534  Name: cassette

    After sort by part number:
ID: 1234  Name: crank arm
ID: 1334  Name:
ID: 1434  Name: regular seat
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1634  Name: shift lever

    After sort by name:
ID: 1334  Name:
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1234  Name: crank arm
ID: 1434  Name: regular seat
ID: 1634  Name: shift lever

*/
}

}

```

## See also

- [IComparer](#)
- [IEquatable<T>](#)
- [IComparer<T>](#)
- [IComparable](#)
- [IComparable<T>](#)



Collaborate with us on  
GitHub

.NET

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET is an open source project.  
Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

# Sorted Collection Types

Article • 09/15/2021

The [System.Collections.SortedList](#) class, the [System.Collections.Generic.SortedList<TKey,TValue>](#) generic class, and the [System.Collections.Generic.SortedDictionary<TKey,TValue>](#) generic class are similar to the [Hashtable](#) class and the [Dictionary<TKey,TValue>](#) generic class in that they implement the [IDictionary](#) interface, but they maintain their elements in sort order by key, and they do not have the  $O(1)$  insertion and retrieval characteristic of hash tables. The three classes have several features in common:

- All three classes implement the [System.Collections.IDictionary](#) interface. The two generic classes also implement the [System.Collections.Generic.IDictionary<TKey,TValue>](#) generic interface.
- Each element is a key/value pair for enumeration purposes.

## ⓘ Note

The nongeneric [SortedList](#) class returns [DictionaryEntry](#) objects when enumerated, although the two generic types return [KeyValuePair<TKey,TValue>](#) objects.

- Elements are sorted according to a [System.Collections.IComparer](#) implementation (for nongeneric [SortedList](#)) or a [System.Collections.Generic.IComparer<T>](#) implementation (for the two generic classes).
- Each class provides properties that return collections containing only the keys or only the values.

The following table lists some of the differences between the two sorted list classes and the [SortedDictionary<TKey,TValue>](#) class.

<b>SortedList nongeneric class and SortedList&lt;TKey,TValue&gt; generic class</b>	<b>SortedDictionary&lt;TKey,TValue&gt; generic class</b>
The properties that return keys and values are indexed, allowing efficient indexed retrieval.	No indexed retrieval.
Retrieval is $O(\log n)$ .	Retrieval is $O(\log n)$ .
Insertion and removal are generally $O(n)$ ; however, insertion is $O(\log n)$ for data that are already in sort	Insertion and removal are $O(\log n)$ .

<a href="#">SortedList nongeneric class and</a> <a href="#">SortedList&lt;TKey,TValue&gt; generic class</a>	<a href="#">SortedDictionary&lt;TKey,TValue&gt;</a> generic class
order, so that each element is added to the end of the list. (This assumes that a resize is not required.)	
Uses less memory than a <a href="#">SortedDictionary&lt;TKey,TValue&gt;</a> .	Uses more memory than the <a href="#">SortedList nongeneric class</a> and the <a href="#">SortedList&lt;TKey,TValue&gt; generic class</a> .

For sorted lists or dictionaries that must be accessible concurrently from multiple threads, you can add sorting logic to a class that derives from [ConcurrentDictionary<TKey,TValue>](#). When considering immutability, the following corresponding immutable types follow similar sorting semantics: [ImmutableSortedSet<T>](#) and [ImmutableSortedDictionary<TKey,TValue>](#).

#### Note

For values that contain their own keys (for example, employee records that contain an employee ID number), you can create a keyed collection that has some characteristics of a list and some characteristics of a dictionary by deriving from the [KeyedCollection<TKey,TItem>](#) generic class.

Starting with .NET Framework 4, the [SortedSet<T>](#) class provides a self-balancing tree that maintains data in sorted order after insertions, deletions, and searches. This class and the [HashSet<T>](#) class implement the [ISet<T>](#) interface.

## See also

- [System.Collections.IDictionary](#)
- [System.Collections.Generic.IDictionary<TKey,TValue>](#)
- [ConcurrentDictionary<TKey,TValue>](#)
- [Commonly Used Collection Types](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

# Hashtable and Dictionary Collection Types

Article • 09/15/2021

The [System.Collections.Hashtable](#) class, and the [System.Collections.Generic.Dictionary<TKey,TValue>](#) and [System.Collections.Concurrent.ConcurrentDictionary<TKey,TValue>](#) generic classes, implement the [System.Collections.IDictionary](#) interface. The [Dictionary<TKey,TValue>](#) generic class also implements the [IDictionary<TKey,TValue>](#) generic interface. Therefore, each element in these collections is a key-and-value pair.

A [Hashtable](#) object consists of buckets that contain the elements of the collection. A bucket is a virtual subgroup of elements within the [Hashtable](#), which makes searching and retrieving easier and faster than in most collections. Each bucket is associated with a hash code, which is generated using a hash function and is based on the key of the element.

The generic [HashSet<T>](#) class is an unordered collection for containing unique elements.

A hash function is an algorithm that returns a numeric hash code based on a key. The key is the value of some property of the object being stored. A hash function must always return the same hash code for the same key. It is possible for a hash function to generate the same hash code for two different keys, but a hash function that generates a unique hash code for each unique key results in better performance when retrieving elements from the hash table.

Each object that is used as an element in a [Hashtable](#) must be able to generate a hash code for itself by using an implementation of the [GetHashCode](#) method. However, you can also specify a hash function for all elements in a [Hashtable](#) by using a [Hashtable](#) constructor that accepts an [IHashCodeProvider](#) implementation as one of its parameters.

When an object is added to a [Hashtable](#), it is stored in the bucket that is associated with the hash code that matches the object's hash code. When a value is being searched for in the [Hashtable](#), the hash code is generated for that value, and the bucket associated with that hash code is searched.

For example, a hash function for a string might take the ASCII codes of each character in the string and add them together to generate a hash code. The string "picnic" would have a hash code that is different from the hash code for the string "basket"; therefore,

the strings "picnic" and "basket" would be in different buckets. In contrast, "stressed" and "desserts" would have the same hash code and would be in the same bucket.

The [Dictionary<TKey, TValue>](#) and [ConcurrentDictionary<TKey, TValue>](#) classes have the same functionality as the [Hashtable](#) class. A [Dictionary<TKey, TValue>](#) of a specific type (other than [Object](#)) provides better performance than a [Hashtable](#) for value types. This is because the elements of [Hashtable](#) are of type [Object](#); therefore, boxing and unboxing typically occur when you store or retrieve a value type. The [ConcurrentDictionary<TKey, TValue>](#) class should be used when multiple threads might be accessing the collection simultaneously.

## See also

- [Hashtable](#)
- [IDictionary](#)
- [IHashCodeProvider](#)
- [Dictionary<TKey, TValue>](#)
- [System.Collections.Generic.IDictionary<TKey, TValue>](#)
- [System.Collections.Concurrent.ConcurrentDictionary<TKey, TValue>](#)
- [Commonly Used Collection Types](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Thread-safe collections

Article • 01/27/2023

The [System.Collections.Concurrent](#) namespace includes several collection classes that are both thread-safe and scalable. Multiple threads can safely and efficiently add or remove items from these collections, without requiring additional synchronization in user code. When you write new code, use the concurrent collection classes to write multiple threads to the collection concurrently. If you're only reading from a shared collection, then you can use the classes in the [System.Collections.Generic](#) namespace.

## System.Collections and System.Collections.Generic

The collection classes in the [System.Collections](#) namespace include [ArrayList](#) and [Hashtable](#). These classes provide some thread safety through the `Synchronized` property, which returns a thread-safe wrapper around the collection. The wrapper works by locking the entire collection on every add or remove operation. Therefore, each thread that's attempting to access the collection must wait for its turn to take the one lock. This process isn't scalable and can cause significant performance degradation for large collections. Also, the design isn't protected from race conditions. For more information, see [Synchronization in Generic Collections](#).

The collection classes in the [System.Collections.Generic](#) namespace include [List<T>](#) and [Dictionary< TKey, TValue >](#). These classes provide improved type safety and performance compared to the [System.Collections](#) classes. However, the [System.Collections.Generic](#) classes don't provide any thread synchronization; user code must provide all synchronization when items are added or removed on multiple threads concurrently.

We recommend using the concurrent collections classes in the [System.Collections.Concurrent](#) namespace because they provide type safety and also more efficient and complete thread safety.

## Fine-grained locking and lock-free mechanisms

Some of the concurrent collection types use lightweight synchronization mechanisms such as [SpinLock](#), [SpinWait](#), [SemaphoreSlim](#), and [CountdownEvent](#). These synchronization types typically use *busy spinning* for brief periods before they put the thread into a true `Wait` state. When wait times are expected to be short, spinning is far less computationally expensive than waiting, which involves an expensive kernel

transition. For collection classes that use spinning, this efficiency means that multiple threads can add and remove items at a high rate. For more information about spinning versus blocking, see [SpinLock](#) and [SpinWait](#).

The [ConcurrentQueue<T>](#) and [ConcurrentStack<T>](#) classes don't use locks at all. Instead, they rely on [Interlocked](#) operations to achieve thread safety.

### ① Note

Because the concurrent collections classes support [ICollection](#), they provide implementations for the [IsSynchronized](#) and [SyncRoot](#) properties, even though these properties are irrelevant. `IsSynchronized` always returns `false` and, `SyncRoot` is always `null` (`Nothing` in Visual Basic).

The following table lists the collection types in the [System.Collections.Concurrent](#) namespace:

Type	Description
<a href="#">BlockingCollection&lt;T&gt;</a>	Provides bounding and blocking functionality for any type that implements <a href="#">IProducerConsumerCollection&lt;T&gt;</a> . For more information, see <a href="#">BlockingCollection Overview</a> .
<a href="#">ConcurrentDictionary&lt;TKey, TValue&gt;</a>	Thread-safe implementation of a dictionary of key-value pairs.
<a href="#">ConcurrentQueue&lt;T&gt;</a>	Thread-safe implementation of a FIFO (first-in, first-out) queue.
<a href="#">ConcurrentStack&lt;T&gt;</a>	Thread-safe implementation of a LIFO (last-in, first-out) stack.
<a href="#">ConcurrentBag&lt;T&gt;</a>	Thread-safe implementation of an unordered collection of elements.
<a href="#">IProducerConsumerCollection&lt;T&gt;</a>	The interface that a type must implement to be used in a <a href="#">BlockingCollection</a> .

## Related articles

Title	Description
<a href="#">BlockingCollection Overview</a>	Describes the functionality provided by the <a href="#">BlockingCollection&lt;T&gt;</a> type.

Title	Description
<a href="#">How to: Add and Remove Items from a ConcurrentDictionary</a>	Describes how to add and remove elements from a <code>ConcurrentDictionary&lt;TKey, TValue&gt;</code>
<a href="#">How to: Add and Take Items Individually from a BlockingCollection</a>	Describes how to add and retrieve items from a blocking collection without using the read-only enumerator.
<a href="#">How to: Add Bounding and Blocking Functionality to a Collection</a>	Describes how to use any collection class as the underlying storage mechanism for an <code>IProducerConsumerCollection&lt;T&gt;</code> collection.
<a href="#">How to: Use ForEach to Remove Items in a BlockingCollection</a>	Describes how to use <code>foreach</code> ( <code>For Each</code> in Visual Basic) to remove all items in a blocking collection.
<a href="#">How to: Use Arrays of Blocking Collections in a Pipeline</a>	Describes how to use multiple blocking collections at the same time to implement a pipeline.
<a href="#">How to: Create an Object Pool by Using a ConcurrentBag</a>	Shows how to use a concurrent bag to improve performance in scenarios where you can reuse objects instead of continually creating new ones.

## Reference

- [System.Collections.Concurrent](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Delegates and lambdas

Article • 01/05/2022

A delegate defines a type that represents references to methods that have a particular parameter list and return type. A method (static or instance) whose parameter list and return type match can be assigned to a variable of that type, then called directly (with the appropriate arguments) or passed as an argument itself to another method and then called. The following example demonstrates delegate use.

C#

```
using System;
using System.Linq;

public class Program
{
    public delegate string Reverse(string s);

    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Reverse rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

- The `public delegate string Reverse(string s);` line creates a delegate type of a method that takes a string parameter and then returns a string parameter.
- The `static string ReverseString(string s)` method, which has the exact same parameter list and return type as the defined delegate type, implements the delegate.
- The `Reverse rev = ReverseString;` line shows that you can assign a method to a variable of the corresponding delegate type.
- The `Console.WriteLine(rev("a string"));` line demonstrates how to use a variable of a delegate type to invoke the delegate.

In order to streamline the development process, .NET includes a set of delegate types that programmers can reuse and not have to create new types. These types are `Func<>`, `Action<>` and `Predicate<>`, and they can be used without having to define new delegate

types. There are some differences between the three types that have to do with the way they were intended to be used:

- `Action<>` is used when there is a need to perform an action using the arguments of the delegate. The method it encapsulates does not return a value.
- `Func<>` is used usually when you have a transformation on hand, that is, you need to transform the arguments of the delegate into a different result. Projections are a good example. The method it encapsulates returns a specified value.
- `Predicate<>` is used when you need to determine if the argument satisfies the condition of the delegate. It can also be written as a `Func<T, bool>`, which means the method returns a boolean value.

We can now take our example above and rewrite it using the `Func<>` delegate instead of a custom type. The program will continue running exactly the same.

C#

```
using System;
using System.Linq;

public class Program
{
    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Func<string, string> rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

For this simple example, having a method defined outside of the `Main` method seems a bit superfluous. .NET Framework 2.0 introduced the concept of *anonymous delegates*, which let you create "inline" delegates without having to specify any additional type or method.

In the following example, an anonymous delegate filters a list to just the even numbers and then prints them to the console.

C#

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

```

public class Program
{
    public static void Main(string[] args)
    {
        List<int> list = new List<int>();

        for (int i = 1; i <= 100; i++)
        {
            list.Add(i);
        }

        List<int> result = list.FindAll(
            delegate (int no)
            {
                return (no % 2 == 0);
            }
        );

        foreach (var item in result)
        {
            Console.WriteLine(item);
        }
    }
}

```

As you can see, the body of the delegate is just a set of expressions, as any other delegate. But instead of it being a separate definition, we've introduced it *ad hoc* in our call to the `List<T>.FindAll` method.

However, even with this approach, there is still much code that we can throw away. This is where *lambda expressions* come into play. Lambda expressions, or just "lambdas" for short, were introduced in C# 3.0 as one of the core building blocks of Language Integrated Query (LINQ). They are just a more convenient syntax for using delegates. They declare a parameter list and method body, but don't have a formal identity of their own, unless they are assigned to a delegate. Unlike delegates, they can be directly assigned as the right-hand side of event registration or in various LINQ clauses and methods.

Since a lambda expression is just another way of specifying a delegate, we should be able to rewrite the above sample to use a lambda expression instead of an anonymous delegate.

C#

```

using System;
using System.Collections.Generic;

public class Program

```

```
{  
    public static void Main(string[] args)  
    {  
        List<int> list = new List<int>();  
  
        for (int i = 1; i <= 100; i++)  
        {  
            list.Add(i);  
        }  
  
        List<int> result = list.FindAll(i => i % 2 == 0);  
  
        foreach (var item in result)  
        {  
            Console.WriteLine(item);  
        }  
    }  
}
```

In the preceding example, the lambda expression used is `i => i % 2 == 0`. Again, it is just a convenient syntax for using delegates. What happens under the covers is similar to what happens with the anonymous delegate.

Again, lambdas are just delegates, which means that they can be used as an event handler without any problems, as the following code snippet illustrates.

C#

```
public MainWindow()  
{  
    InitializeComponent();  
  
    Loaded += (o, e) =>  
    {  
        this.Title = "Loaded";  
    };  
}
```

The `+=` operator in this context is used to subscribe to an [event](#). For more information, see [How to subscribe to and unsubscribe from events](#).

## Further reading and resources

- [Delegates](#)
- [Lambda expressions](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Delegate.CreateDelegate methods

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

The [CreateDelegate](#) methods create a delegate of a specified type.

## CreateDelegate(Type, MethodInfo) method

This method overload is equivalent to calling the [CreateDelegate\(Type, MethodInfo, Boolean\)](#) method overload and specifying `true` for `throwOnBindFailure`.

### Examples

This section contains two code examples. The first example demonstrates the two kinds of delegates that can be created with this method overload: open over an instance method and open over a static method.

The second code example demonstrates compatible parameter types and return types.

### Example 1

The following code example demonstrates the two ways a delegate can be created using this overload of the [CreateDelegate](#) method.

#### ⓘ Note

There are two overloads of the [CreateDelegate](#) method that specify a [MethodInfo](#) but not a first argument; their functionality is the same except that one allows you to specify whether to throw on failure to bind, and the other always throws. This code example uses both overloads.

The example declares a class `C` with a static method `M2` and an instance method `M1`, and two delegate types: `D1` takes an instance of `C` and a string, and `D2` takes a string.

A second class named `Example` contains the code that creates the delegates.

- A delegate of type `D1`, representing an open instance method, is created for the instance method `M1`. An instance must be passed when the delegate is invoked.
- A delegate of type `D2`, representing an open static method, is created for the static method `M2`.

C#

```

using System;
using System.Reflection;

// Declare three delegate types for demonstrating the combinations
// of static versus instance methods and open versus closed
// delegates.
//
public delegate void D1(C c, string s);
public delegate void D2(string s);
public delegate void D3();

// A sample class with an instance method and a static method.
//
public class C
{
    private int id;
    public C(int id) { this.id = id; }

    public void M1(string s)
    {
        Console.WriteLine("Instance method M1 on C: id = {0}, s = {1}",
                          this.id, s);
    }

    public static void M2(string s)
    {
        Console.WriteLine("Static method M2 on C: s = {0}", s);
    }
}

public class Example2
{
    public static void Main()
    {
        C c1 = new C(42);

        // Get a MethodInfo for each method.
        //
        MethodInfo mi1 = typeof(C).GetMethod("M1",
                                              BindingFlags.Public | BindingFlags.Instance);
        MethodInfo mi2 = typeof(C).GetMethod("M2",
                                              BindingFlags.Public | BindingFlags.Static);

        D1 d1;
        D2 d2;
        D3 d3;
    }
}

```

```
Console.WriteLine("\nAn instance method closed over C.");
// In this case, the delegate and the
// method must have the same list of argument types; use
// delegate type D2 with instance method M1.
//
Delegate test =
    Delegate.CreateDelegate(typeof(D2), c1, mi1, false);

// Because false was specified for throwOnBindFailure
// in the call to CreateDelegate, the variable 'test'
// contains null if the method fails to bind (for
// example, if mi1 happened to represent a method of
// some class other than C).
//
if (test != null)
{
    d2 = (D2)test;

    // The same instance of C is used every time the
    // delegate is invoked.
    d2("Hello, World!");
    d2("Hi, Mom!");
}

Console.WriteLine("\nAn open instance method.");
// In this case, the delegate has one more
// argument than the instance method; this argument comes
// at the beginning, and represents the hidden instance
// argument of the instance method. Use delegate type D1
// with instance method M1.
//
d1 = (D1)Delegate.CreateDelegate(typeof(D1), null, mi1);

// An instance of C must be passed in each time the
// delegate is invoked.
//
d1(c1, "Hello, World!");
d1(new C(5280), "Hi, Mom!");

Console.WriteLine("\nAn open static method.");
// In this case, the delegate and the method must
// have the same list of argument types; use delegate type
// D2 with static method M2.
//
d2 = (D2)Delegate.CreateDelegate(typeof(D2), null, mi2);

// No instances of C are involved, because this is a static
// method.
//
d2("Hello, World!");
d2("Hi, Mom!");

Console.WriteLine("\nA static method closed over the first argument
(String).");
```

```

    // The delegate must omit the first argument of the method.
    // A string is passed as the firstArgument parameter, and
    // the delegate is bound to this string. Use delegate type
    // D3 with static method M2.
    //
    d3 = (D3)Delegate.CreateDelegate(typeof(D3),
        "Hello, World!", mi2);

        // Each time the delegate is invoked, the same string is
        // used.
        d3();
    }

}

/* This code example produces the following output:

An instance method closed over C.
Instance method M1 on C: id = 42, s = Hello, World!
Instance method M1 on C: id = 42, s = Hi, Mom!

An open instance method.
Instance method M1 on C: id = 42, s = Hello, World!
Instance method M1 on C: id = 5280, s = Hi, Mom!

An open static method.
Static method M2 on C: s = Hello, World!
Static method M2 on C: s = Hi, Mom!

A static method closed over the first argument (String).
Static method M2 on C: s = Hello, World!
*/

```

## Example 2

The following code example demonstrates compatibility of parameter types and return types.

The code example defines a base class named `Base` and a class named `Derived` that derives from `Base`. The derived class has a `static` (`Shared` in Visual Basic) method named `MyMethod` with one parameter of type `Base` and a return type of `Derived`. The code example also defines a delegate named `Example` that has one parameter of type `Derived` and a return type of `Base`.

The code example demonstrates that the delegate named `Example` can be used to represent the method `MyMethod`. The method can be bound to the delegate because:

- The parameter type of the delegate (`Derived`) is more restrictive than the parameter type of `MyMethod` (`Base`), so that it is always safe to pass the argument

of the delegate to `MyMethod`.

- The return type of `MyMethod` (`Derived`) is more restrictive than the parameter type of the delegate (`Base`), so that it is always safe to cast the return type of the method to the return type of the delegate.

The code example produces no output.

C#

```
using System;
using System.Reflection;

// Define two classes to use in the demonstration, a base class and
// a class that derives from it.
//
public class Base { }

public class Derived : Base
{
    // Define a static method to use in the demonstration. The method
    // takes an instance of Base and returns an instance of Derived.
    // For the purposes of the demonstration, it is not necessary for
    // the method to do anything useful.
    //
    public static Derived MyMethod(Base arg)
    {
        Base dummy = arg;
        return new Derived();
    }
}

// Define a delegate that takes an instance of Derived and returns an
// instance of Base.
//
public delegate Base Example5(Derived arg);

class Test
{
    public static void Main()
    {
        // The binding flags needed to retrieve MyMethod.
        BindingFlags flags = BindingFlags.Public | BindingFlags.Static;

        // Get a MethodInfo that represents MyMethod.
        MethodInfo minfo = typeof(Derived).GetMethod("MyMethod", flags);

        // Demonstrate contravariance of parameter types and covariance
        // of return types by using the delegate Example5 to represent
        // MyMethod. The delegate binds to the method because the
        // parameter of the delegate is more restrictive than the
        // parameter of the method (that is, the delegate accepts an
        // instance of Derived, which can always be safely passed to
```

```

    // a parameter of type Base), and the return type of MyMethod
    // is more restrictive than the return type of Example5 (that
    // is, the method returns an instance of Derived, which can
    // always be safely cast to type Base).
    //
    Example5 ex =
        (Example5)Delegate.CreateDelegate(typeof(Example5), mInfo);

    // Execute MyMethod using the delegate Example5.
    //
    Base b = ex(new Derived());
}
}

```

## CreateDelegate(Type, Object, MethodInfo) and CreateDelegate(Type, Object, MethodInfo, Boolean) methods

The functionality of these two overloads is the same except that one allows you to specify whether to throw on failure to bind, and the other always throws.

The delegate type and the method must have compatible return types. That is, the return type of `method` must be assignable to the return type of `type`.

`firstArgument`, the second parameter to these overloads, is the first argument of the method the delegate represents. If `firstArgument` is supplied, it is passed to `method` every time the delegate is invoked; `firstArgument` is said to be bound to the delegate, and the delegate is said to be closed over its first argument. If `method` is `static` (Shared in Visual Basic), the argument list supplied when invoking the delegate includes all parameters except the first; if `method` is an instance method, then `firstArgument` is passed to the hidden instance parameter (represented by `this` in C#, or by `Me` in Visual Basic).

If `firstArgument` is supplied, the first parameter of `method` must be a reference type, and `firstArgument` must be compatible with that type.

### ⓘ Important

If `method` is `static` (Shared in Visual Basic) and its first parameter is of type `Object` or `ValueType`, then `firstArgument` can be a value type. In this case `firstArgument` is automatically boxed. Automatic boxing does not occur for any other arguments, as it would in a C# or Visual Basic function call.

If `firstArgument` is a null reference and `method` is an instance method, the result depends on the signatures of the delegate type `type` and of `method`:

- If the signature of `type` explicitly includes the hidden first parameter of `method`, the delegate is said to represent an open instance method. When the delegate is invoked, the first argument in the argument list is passed to the hidden instance parameter of `method`.
- If the signatures of `method` and `type` match (that is, all parameter types are compatible), then the delegate is said to be closed over a null reference. Invoking the delegate is like calling an instance method on a null instance, which is not a particularly useful thing to do.

If `firstArgument` is a null reference and `method` is static, the result depends on the signatures of the delegate type `type` and of `method`:

- If the signature of `method` and `type` match (that is, all parameter types are compatible), the delegate is said to represent an open static method. This is the most common case for static methods. In this case, you can get slightly better performance by using the [CreateDelegate\(Type, MethodInfo\)](#) method overload.
- If the signature of `type` begins with the second parameter of `method` and the rest of the parameter types are compatible, then the delegate is said to be closed over a null reference. When the delegate is invoked, a null reference is passed to the first parameter of `method`.

## Example

The following code example shows all the methods a single delegate type can represent: closed over an instance method, open over an instance method, open over a static method, and closed over a static method.

The code example defines two classes, `C` and `F`, and a delegate type `D` with one argument of type `C`. The classes have matching static and instance methods `M1`, `M3`, and `M4`, and class `C` also has an instance method `M2` that has no arguments.

A third class named `Example` contains the code that creates the delegates.

- Delegates are created for instance method `M1` of type `C` and type `F`; each is closed over an instance of the respective type. Method `M1` of type `C` displays the `ID` properties of the bound instance and of the argument.
- A delegate is created for method `M2` of type `C`. This is an open instance delegate, in which the argument of the delegate represents the hidden first argument on the

instance method. The method has no other arguments. It is called as if it were a static method.

- Delegates are created for static method `M3` of type `C` and type `F`; these are open static delegates.
- Finally, delegates are created for static method `M4` of type `C` and type `F`; each method has the declaring type as its first argument, and an instance of the type is supplied, so the delegates are closed over their first arguments. Method `M4` of type `C` displays the `ID` properties of the bound instance and of the argument.

C#

```
using System;
using System.Reflection;

// Declare a delegate type. The object of this code example
// is to show all the methods this delegate can bind to.
//
public delegate void D(C1 c);

// Declare two sample classes, C1 and F. Class C1 has an ID
// property so instances can be identified.
//
public class C1
{
    private int id;
    public int ID { get { return id; } }
    public C1(int id) { this.id = id; }

    public void M1(C1 c)
    {
        Console.WriteLine("Instance method M1(C1 c) on C1:  this.id = {0}",
c.ID = {1},
            this.id, c.ID);
    }

    public void M2()
    {
        Console.WriteLine("Instance method M2() on C1:  this.id = {0}",
            this.id);
    }

    public static void M3(C1 c)
    {
        Console.WriteLine("Static method M3(C1 c) on C1:  c.ID = {0}",
c.ID);
    }

    public static void M4(C1 c1, C1 c2)
    {
        Console.WriteLine("Static method M4(C1 c1, C1 c2) on C1:  c1.ID =
{0}, c2.ID = {1}",
            c1.ID, c2.ID);
    }
}
```

```

        c1.ID, c2.ID);
    }

}

public class F
{
    public void M1(C1 c)
    {
        Console.WriteLine("Instance method M1(C1 c) on F: c.ID = {0}",
            c.ID);
    }

    public static void M3(C1 c)
    {
        Console.WriteLine("Static method M3(C1 c) on F: c.ID = {0}", c.ID);
    }

    public static void M4(F f, C1 c)
    {
        Console.WriteLine("Static method M4(F f, C1 c) on F: c.ID = {0}",
            c.ID);
    }
}

public class Example
{
    public static void Main()
    {
        C1 c1 = new C1(42);
        C1 c2 = new C1(1491);
        F f1 = new F();

        D d;

        // Instance method with one argument of type C1.
        MethodInfo cmi1 = typeof(C1).GetMethod("M1");
        // Instance method with no arguments.
        MethodInfo cmi2 = typeof(C1).GetMethod("M2");
        // Static method with one argument of type C1.
        MethodInfo cmi3 = typeof(C1).GetMethod("M3");
        // Static method with two arguments of type C1.
        MethodInfo cmi4 = typeof(C1).GetMethod("M4");

        // Instance method with one argument of type C1.
        MethodInfo fmi1 = typeof(F).GetMethod("M1");
        // Static method with one argument of type C1.
        MethodInfo fmi3 = typeof(F).GetMethod("M3");
        // Static method with an argument of type F and an argument
        // of type C1.
        MethodInfo fmi4 = typeof(F).GetMethod("M4");

        Console.WriteLine("\nAn instance method on any type, with an
argument of type C1.");
        // D can represent any instance method that exactly matches its
        // signature. Methods on C1 and F are shown here.
    }
}
```

```

//  

d = (D)Delegate.CreateDelegate(typeof(D), c1, cmi1);  

d(c2);  

d = (D)Delegate.CreateDelegate(typeof(D), f1, fmi1);  

d(c2);  

  

Console.WriteLine("\nAn instance method on C1 with no arguments.");  

// D can represent an instance method on C1 that has no arguments;  

// in this case, the argument of D represents the hidden first  

// argument of any instance method. The delegate acts like a  

// static method, and an instance of C1 must be passed each time  

// it is invoked.  

//  

d = (D)Delegate.CreateDelegate(typeof(D), null, cmi2);  

d(c1);  

  

Console.WriteLine("\nA static method on any type, with an argument  

of type C1.");  

// D can represent any static method with the same signature.  

// Methods on F and C1 are shown here.  

//  

d = (D)Delegate.CreateDelegate(typeof(D), null, cmi3);  

d(c1);  

d = (D)Delegate.CreateDelegate(typeof(D), null, fmi3);  

d(c1);  

  

Console.WriteLine("\nA static method on any type, with an argument  

of");  

Console.WriteLine("    that type and an argument of type C1.");  

// D can represent any static method with one argument of the  

// type the method belongs and a second argument of type C1.  

// In this case, the method is closed over the instance of  

// supplied for the its first argument, and acts like an instance  

// method. Methods on F and C1 are shown here.  

//  

d = (D)Delegate.CreateDelegate(typeof(D), c1, cmi4);  

d(c2);  

Delegate test =  

    Delegate.CreateDelegate(typeof(D), f1, fmi4, false);  

  

// This final example specifies false for throwOnBindFailure  

// in the call to CreateDelegate, so the variable 'test'  

// contains Nothing if the method fails to bind (for  

// example, if fmi4 happened to represent a method of  

// some class other than F).  

//  

if (test != null)  

{  

    d = (D)test;  

    d(c2);  

}  

}  

}  

/* This code example produces the following output:

```

```
An instance method on any type, with an argument of type C1.  
Instance method M1(C1 c) on C1: this.id = 42, c.ID = 1491  
Instance method M1(C1 c) on F: c.ID = 1491  
  
An instance method on C1 with no arguments.  
Instance method M2() on C1: this.id = 42  
  
A static method on any type, with an argument of type C1.  
Static method M3(C1 c) on C1: c.ID = 42  
Static method M3(C1 c) on F: c.ID = 42  
  
A static method on any type, with an argument of  
that type and an argument of type C1.  
Static method M4(C1 c1, C1 c2) on C1: c1.ID = 42, c2.ID = 1491  
Static method M4(F f, C1 c) on F: c.ID = 1491  
*/
```

## Compatible parameter types and return type

The parameter types and return type of a delegate created using this method overload must be compatible with the parameter types and return type of the method the delegate represents; the types do not have to match exactly.

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 with a parameter of type [Hashtable](#) and a return type of [Object](#) can represent a method with a parameter of type [Object](#) and a return value of type [Hashtable](#).

## Determine the methods a delegate can represent

Another useful way to think of the flexibility provided by the [CreateDelegate\(Type, Object, MethodInfo\)](#) overload is that any given delegate can represent four different combinations of method signature and method kind (static versus instance). Consider a

delegate type `D` with one argument of type `C`. The following describes the methods `D` can represent, ignoring the return type since it must match in all cases:

- `D` can represent any instance method that has exactly one argument of type `C`, regardless of what type the instance method belongs to. When `CreateDelegate` is called, `firstArgument` is an instance of the type `method` belongs to, and the resulting delegate is said to be closed over that instance. (Trivially, `D` can also be closed over a null reference if `firstArgument` is a null reference.)
- `D` can represent an instance method of `C` that has no arguments. When `CreateDelegate` is called, `firstArgument` is a null reference. The resulting delegate represents an open instance method, and an instance of `C` must be supplied each time it is invoked.
- `D` can represent a static method that takes one argument of type `C`, and that method can belong to any type. When `CreateDelegate` is called, `firstArgument` is a null reference. The resulting delegate represents an open static method, and an instance of `C` must be supplied each time it is invoked.
- `D` can represent a static method that belongs to type `F` and has two arguments, of type `F` and type `C`. When `CreateDelegate` is called, `firstArgument` is an instance of `F`. The resulting delegate represents a static method that is closed over that instance of `F`. Note that in the case where `F` and `C` are the same type, the static method has two arguments of that type. (In this case, `D` is closed over a null reference if `firstArgument` is a null reference.)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



## .NET feedback

.NET is an open source project.  
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Enum class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

An enumeration is a set of named constants whose underlying type is any integral type. If no underlying type is explicitly declared, [Int32](#) is used. [Enum](#) is the base class for all enumerations in .NET. Enumeration types are defined by the `enum` keyword in C#, the `Enum ... End Enum` construct in Visual Basic, and the `type` keyword in F#.

[Enum](#) provides methods for comparing instances of this class, converting the value of an instance to its string representation, converting the string representation of a number to an instance of this class, and creating an instance of a specified enumeration and value.

You can also treat an enumeration as a bit field. For more information, see the [Non-exclusive members and the Flags attribute](#) section and [FlagsAttribute](#).

## Create an enumeration type

Programming languages typically provide syntax to declare an enumeration that consists of a set of named constants and their values. The following example illustrates the syntax used by C#, F#, and Visual Basic to define an enumeration. It creates an enumeration named `ArrivalStatus` that has three members: `ArrivalStatus.Early`, `ArrivalStatus.OnTime`, and `ArrivalStatus.Late`. Note that in all cases, the enumeration does not explicitly inherit from [Enum](#); the inheritance relationship is handled implicitly by the compiler.

C#

```
public enum ArrivalStatus { Unknown=-3, Late=-1, OnTime=0, Early=1 };
```

### ⚠ Warning

You should never create an enumeration type whose underlying type is non-integral or [Char](#). Although you can create such an enumeration type by using reflection, method calls that use the resulting type are unreliable and may also throw additional exceptions.

# Instantiate an enumeration type

You can instantiate an enumeration type just as you instantiate any other value type: by declaring a variable and assigning one of the enumeration's constants to it. The following example instantiates an `ArrivalStatus` whose value is `ArrivalStatus.OnTime`.

C#

```
public class Example
{
    public static void Main()
    {
        ArrivalStatus status = ArrivalStatus.OnTime;
        Console.WriteLine("Arrival Status: {0} ({0:D})", status);
    }
}
// The example displays the following output:
//      Arrival Status: OnTime (0)
```

You can also instantiate an enumeration value in the following ways:

- By using a particular programming language's features to cast (as in C#) or convert (as in Visual Basic) an integer value to an enumeration value. The following example creates an `ArrivalStatus` object whose value is `ArrivalStatus.Early` in this way.

C#

```
ArrivalStatus status2 = (ArrivalStatus)1;
Console.WriteLine("Arrival Status: {0} ({0:D})", status2);
// The example displays the following output:
//      Arrival Status: Early (1)
```

- By calling its implicit parameterless constructor. As the following example shows, in this case the underlying value of the enumeration instance is 0. However, this is not necessarily the value of a valid constant in the enumeration.

C#

```
ArrivalStatus status1 = new ArrivalStatus();
Console.WriteLine("Arrival Status: {0} ({0:D})", status1);
// The example displays the following output:
//      Arrival Status: OnTime (0)
```

- By calling the `Parse` or `TryParse` method to parse a string that contains the name of a constant in the enumeration. For more information, see the [Parse enumeration](#)

values section.

- By calling the [ToObject](#) method to convert an integral value to an enumeration type. For more information, see the [Perform conversions](#) section.

## Enumeration best practices

We recommend that you use the following best practices when you define enumeration types:

- If you have not defined an enumeration member whose value is 0, consider creating a `None` enumerated constant. By default, the memory used for the enumeration is initialized to zero by the common language runtime. Consequently, if you do not define a constant whose value is zero, the enumeration will contain an illegal value when it is created.
- If there is an obvious default case that your application has to represent, consider using an enumerated constant whose value is zero to represent it. If there is no default case, consider using an enumerated constant whose value is zero to specify the case that is not represented by any of the other enumerated constants.
- Do not specify enumerated constants that are reserved for future use.
- When you define a method or property that takes an enumerated constant as a value, consider validating the value. The reason is that you can cast a numeric value to the enumeration type even if that numeric value is not defined in the enumeration.

Additional best practices for enumeration types whose constants are bit fields are listed in the [Non-exclusive members and the Flags attribute](#) section.

## Perform operations with enumerations

You cannot define new methods when you are creating an enumeration. However, an enumeration type inherits a complete set of static and instance methods from the [Enum](#) class. The following sections survey most of these methods, in addition to several other methods that are commonly used when working with enumeration values.

## Perform conversions

You can convert between an enumeration member and its underlying type by using a casting (in C# and F#), or conversion (in Visual Basic) operator. In F#, the `enum` function

is also used. The following example uses casting or conversion operators to perform conversions both from an integer to an enumeration value and from an enumeration value to an integer.

```
C#
```

```
int value3 = 2;
ArrivalStatus status3 = (ArrivalStatus)value3;

int value4 = (int)status3;
```

The [Enum](#) class also includes a [ToObject](#) method that converts a value of any integral type to an enumeration value. The following example uses the [ToObject\(Type, Int32\)](#) method to convert an [Int32](#) to an [ArrivalStatus](#) value. Note that, because the [ToObject](#) returns a value of type [Object](#), the use of a casting or conversion operator may still be necessary to cast the object to the enumeration type.

```
C#
```

```
int number = -1;
ArrivalStatus arrived =
(ArrivalStatus)ArrivalStatus.ToObject(typeof(ArrivalStatus), number);
```

When converting an integer to an enumeration value, it is possible to assign a value that is not actually a member of the enumeration. To prevent this, you can pass the integer to the [IsDefined](#) method before performing the conversion. The following example uses this method to determine whether the elements in an array of integer values can be converted to [ArrivalStatus](#) values.

```
C#
```

```
using System;

public class Example3
{
    public static void Main()
    {
        int[] values = { -3, -1, 0, 1, 5, Int32.MaxValue };
        foreach (var value in values)
        {
            ArrivalStatus status;
            if (Enum.IsDefined(typeof(ArrivalStatus), value))
                status = (ArrivalStatus)value;
            else
                status = ArrivalStatus.Unknown;
            Console.WriteLine("Converted {0:N0} to {1}", value, status);
        }
    }
}
```

```
        }
    }
    // The example displays the following output:
    //      Converted -3 to Unknown
    //      Converted -1 to Late
    //      Converted 0 to OnTime
    //      Converted 1 to Early
    //      Converted 5 to Unknown
    //      Converted 2,147,483,647 to Unknown
```

Although the [Enum](#) class provides explicit interface implementations of the [IConvertible](#) interface for converting from an enumeration value to an integral type, you should use the methods of the [Convert](#) class, such as [ToInt32](#), to perform these conversions. The following example illustrates how you can use the [GetUnderlyingType](#) method along with the [Convert.ChangeType](#) method to convert an enumeration value to its underlying type. Note that this example does not require the underlying type of the enumeration to be known at compile time.

C#

```
ArrivalStatus status = ArrivalStatus.Early;
var number = Convert.ChangeType(status,
    Enum.GetUnderlyingType(typeof(ArrivalStatus)));
Console.WriteLine("Converted {0} to {1}", status, number);
// The example displays the following output:
//      Converted Early to 1
```

## Parse enumeration values

The [Parse](#) and [TryParse](#) methods allow you to convert the string representation of an enumeration value to that value. The string representation can be either the name or the underlying value of an enumeration constant. Note that the parsing methods will successfully convert string representations of numbers that are not members of a particular enumeration if the strings can be converted to a value of the enumeration's underlying type. To prevent this, the [IsDefined](#) method can be called to ensure that the result of the parsing method is a valid enumeration value. The example illustrates this approach and demonstrates calls to both the [Parse\(Type, String\)](#) and [Enum.TryParse<TEnum>\(String, TEnum\)](#) methods. Note that the non-generic parsing method returns an object that you may have to cast (in C# and F#) or convert (in Visual Basic) to the appropriate enumeration type.

C#

```
string number = "-1";
string name = "Early";
```

```

try
{
    ArrivalStatus status1 = (ArrivalStatus)Enum.Parse(typeof(ArrivalStatus),
number);
    if (!(Enum.IsDefined(typeof(ArrivalStatus), status1)))
        status1 = ArrivalStatus.Unknown;
    Console.WriteLine("Converted '{0}' to {1}", number, status1);
}
catch (FormatException)
{
    Console.WriteLine("Unable to convert '{0}' to an ArrivalStatus value.",
number);
}

ArrivalStatus status2;
if (Enum.TryParse<ArrivalStatus>(name, out status2))
{
    if (!(Enum.IsDefined(typeof(ArrivalStatus), status2)))
        status2 = ArrivalStatus.Unknown;
    Console.WriteLine("Converted '{0}' to {1}", name, status2);
}
else
{
    Console.WriteLine("Unable to convert '{0}' to an ArrivalStatus value.",
number);
}
// The example displays the following output:
//      Converted '-1' to Late
//      Converted 'Early' to Early

```

## Format enumeration values

You can convert enumeration values to their string representations by calling the static [Format](#) method, as well as the overloads of the instance [ToString](#) method. You can use a format string to control the precise way in which an enumeration value is represented as a string. For more information, see [Enumeration Format Strings](#). The following example uses each of the supported enumeration format strings ("G" or "g", "D" or "d", "X" or "x", and "F" or "f" ) to convert a member of the `ArrivalStatus` enumeration to its string representations.

C#

```

string[] formats = { "G", "F", "D", "X" };
ArrivalStatus status = ArrivalStatus.Late;
foreach (var fmt in formats)
    Console.WriteLine(status.ToString(fmt));

// The example displays the following output:
//      Late

```

```
//      Late
//      -1
//      FFFFFFFF
```

## Iterate enumeration members

The `Enum` type does not implement the `IEnumerable` or `IEnumerable<T>` interface, which would enable you to iterate members of a collection by using a `foreach` (in C#), `for..in` (in F#), or `For Each` (in Visual Basic) construct. However, you can enumerate members in either of two ways.

- You can call the `GetNames` method to retrieve a string array containing the names of the enumeration members. Next, for each element of the string array, you can call the `Parse` method to convert the string to its equivalent enumeration value. The following example illustrates this approach.

C#

```
string[] names = Enum.GetNames(typeof(ArrivalStatus));
Console.WriteLine("Members of {0}:", typeof(ArrivalStatus).Name);
Array.Sort(names);
foreach (var name in names)
{
    ArrivalStatus status =
(ArrivalStatus)Enum.Parse(typeof(ArrivalStatus), name);
    Console.WriteLine("    {0} ({0:D})", status);
}
// The example displays the following output:
//      Members of ArrivalStatus:
//          Early (1)
//          Late (-1)
//          OnTime (0)
//          Unknown (-3)
```

- You can call the `GetValues` method to retrieve an array that contains the underlying values in the enumeration. Next, for each element of the array, you can call the `ToObject` method to convert the integer to its equivalent enumeration value. The following example illustrates this approach.

C#

```
var values = Enum.GetValues(typeof(ArrivalStatus));
Console.WriteLine("Members of {0}:", typeof(ArrivalStatus).Name);
foreach (ArrivalStatus status in values)
{
    Console.WriteLine("    {0} ({0:D})", status);
}
```

```
// The example displays the following output:  
//      Members of ArrivalStatus:  
//      OnTime (0)  
//      Early (1)  
//      Unknown (-3)  
//      Late (-1)
```

## Non-exclusive members and the Flags attribute

One common use of an enumeration is to represent a set of mutually exclusive values. For example, an `ArrivalStatus` instance can have a value of `Early`, `OnTime`, or `Late`. It makes no sense for the value of an `ArrivalStatus` instance to reflect more than one enumeration constant.

In other cases, however, the value of an enumeration object can include multiple enumeration members, and each member represents a bit field in the enumeration value. The [FlagsAttribute](#) attribute can be used to indicate that the enumeration consists of bit fields. For example, an enumeration named `Pets` might be used to indicate the kinds of pets in a household. It can be defined as follows.

```
C#  
  
[Flags]  
public enum Pets  
{  
    None = 0, Dog = 1, Cat = 2, Bird = 4, Rodent = 8,  
    Reptile = 16, Other = 32  
};
```

The `Pets` enumeration can then be used as shown in the following example.

```
C#  
  
Pets familyPets = Pets.Dog | Pets.Cat;  
Console.WriteLine("Pets: {0:G} ({0:D})", familyPets);  
// The example displays the following output:  
//      Pets: Dog, Cat (3)
```

The following best practices should be used when defining a bitwise enumeration and applying the [FlagsAttribute](#) attribute.

- Use the [FlagsAttribute](#) custom attribute for an enumeration only if a bitwise operation (AND, OR, EXCLUSIVE OR) is to be performed on a numeric value.

- Define enumeration constants in powers of two, that is, 1, 2, 4, 8, and so on. This means the individual flags in combined enumeration constants do not overlap.
- Consider creating an enumerated constant for commonly used flag combinations. For example, if you have an enumeration used for file I/O operations that contains the enumerated constants `Read = 1` and `Write = 2`, consider creating the enumerated constant `ReadWrite = Read OR Write`, which combines the `Read` and `Write` flags. In addition, the bitwise OR operation used to combine the flags might be considered an advanced concept in some circumstances that should not be required for simple tasks.
- Use caution if you define a negative number as a flag enumerated constant because many flag positions might be set to 1, which might make your code confusing and encourage coding errors.
- A convenient way to test whether a flag is set in a numeric value is to call the instance `HasFlag` method, as shown in the following example.

C#

```
Pets familyPets = Pets.Dog | Pets.Cat;
if (familyPets.HasFlag(Pets.Dog))
    Console.WriteLine("The family has a dog.");
// The example displays the following output:
//      The family has a dog.
```

It is equivalent to performing a bitwise AND operation between the numeric value and the flag enumerated constant, which sets all bits in the numeric value to zero that do not correspond to the flag, and then testing whether the result of that operation is equal to the flag enumerated constant. This is illustrated in the following example.

C#

```
Pets familyPets = Pets.Dog | Pets.Cat;
if ((familyPets & Pets.Dog) == Pets.Dog)
    Console.WriteLine("The family has a dog.");
// The example displays the following output:
//      The family has a dog.
```

- Use `None` as the name of the flag enumerated constant whose value is zero. You cannot use the `None` enumerated constant in a bitwise AND operation to test for a flag because the result is always zero. However, you can perform a logical, not a bitwise, comparison between the numeric value and the `None` enumerated

constant to determine whether any bits in the numeric value are set. This is illustrated in the following example.

```
C#  
  
Pets familyPets = Pets.Dog | Pets.Cat;  
if (familyPets == Pets.None)  
    Console.WriteLine("The family has no pets.");  
else  
    Console.WriteLine("The family has pets.");  
// The example displays the following output:  
//     The family has pets.
```

- Do not define an enumeration value solely to mirror the state of the enumeration itself. For example, do not define an enumerated constant that merely marks the end of the enumeration. If you need to determine the last value of the enumeration, check for that value explicitly. In addition, you can perform a range check for the first and last enumerated constant if all values within the range are valid.

## Add enumeration methods

Because enumeration types are defined by language structures, such as `enum` (C#), and `Enum` (Visual Basic), you cannot define custom methods for an enumeration type other than those methods inherited from the `Enum` class. However, you can use extension methods to add functionality to a particular enumeration type.

In the following example, the `Grades` enumeration represents the possible letter grades that a student may receive in a class. An extension method named `Passing` is added to the `Grades` type so that each instance of that type now "knows" whether it represents a passing grade or not. The `Extensions` class also contains a static read-write variable that defines the minimum passing grade. The return value of the `Passing` extension method reflects the current value of that variable.

```
C#  
  
using System;  
  
// Define an enumeration to represent student grades.  
public enum Grades { F = 0, D = 1, C = 2, B = 3, A = 4 };  
  
// Define an extension method for the Grades enumeration.  
public static class Extensions  
{  
    public static Grades minPassing = Grades.D;
```

```

    public static bool Passing(this Grades grade)
    {
        return grade >= minPassing;
    }
}

class Example8
{
    static void Main()
    {
        Grades g1 = Grades.D;
        Grades g2 = Grades.F;
        Console.WriteLine("{0} {1} a passing grade.", g1, g1.Passing() ? "is" : "is not");
        Console.WriteLine("{0} {1} a passing grade.", g2, g2.Passing() ? "is" : "is not");

        Extensions.minPassing = Grades.C;
        Console.WriteLine("\nRaising the bar!\n");
        Console.WriteLine("{0} {1} a passing grade.", g1, g1.Passing() ? "is" : "is not");
        Console.WriteLine("{0} {1} a passing grade.", g2, g2.Passing() ? "is" : "is not");
    }
}

// The example displays the following output:
//      D is a passing grade.
//      F is not a passing grade.
//
//      Raising the bar!
//
//      D is not a passing grade.
//      F is not a passing grade.

```

## Examples

The following example demonstrates using an enumeration to represent named values and another enumeration to represent named bit fields.

C#

```

using System;

public class EnumTest {
    enum Days { Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday,
Friday };
    enum BoilingPoints { Celsius = 100, Fahrenheit = 212 };
    [Flags]
    enum Colors { Red = 1, Green = 2, Blue = 4, Yellow = 8 };
}

```

```
public static void Main() {  
  
    Type weekdays = typeof(Days);  
    Type boiling = typeof(BoilingPoints);  
  
    Console.WriteLine("The days of the week, and their corresponding  
values in the Days Enum are:");  
  
    foreach ( string s in Enum.GetNames(weekdays) )  
        Console.WriteLine( "{0,-11}= {1}", s, Enum.Format( weekdays,  
Enum.Parse(weekdays, s), "d"));  
  
    Console.WriteLine();  
    Console.WriteLine("Enums can also be created which have values that  
represent some meaningful amount.");  
    Console.WriteLine("The BoilingPoints Enum defines the following  
items, and corresponding values:");  
  
    foreach ( string s in Enum.GetNames(boiling) )  
        Console.WriteLine( "{0,-11}= {1}", s, Enum.Format(boiling,  
Enum.Parse(boiling, s), "d"));  
  
    Colors myColors = Colors.Red | Colors.Blue | Colors.Yellow;  
    Console.WriteLine();  
    Console.WriteLine("myColors holds a combination of colors. Namely:  
{0}", myColors);  
}  
}
```

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.FlagsAttribute class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [FlagsAttribute](#) attribute indicates that an enumeration can be treated as a bit field; that is, a set of flags.

Bit fields are generally used for lists of elements that might occur in combination, whereas enumeration constants are generally used for lists of mutually exclusive elements. Therefore, bit fields are designed to be combined with a bitwise `OR` operation to generate unnamed values, whereas enumerated constants are not. Languages vary in their use of bit fields compared to enumeration constants.

## Attributes of the FlagsAttribute

[AttributeUsageAttribute](#) is applied to this class, and its [Inherited](#) property specifies `false`. This attribute can only be applied to enumerations.

## Guidelines for FlagsAttribute and enum

- Use the [FlagsAttribute](#) custom attribute for an enumeration only if a bitwise operation (AND, OR, EXCLUSIVE OR) is to be performed on a numeric value.
- Define enumeration constants in powers of two, that is, 1, 2, 4, 8, and so on. This means the individual flags in combined enumeration constants do not overlap.
- Consider creating an enumerated constant for commonly used flag combinations. For example, if you have an enumeration used for file I/O operations that contains the enumerated constants `Read = 1` and `Write = 2`, consider creating the enumerated constant `ReadWrite = Read OR Write`, which combines the `Read` and `Write` flags. In addition, the bitwise OR operation used to combine the flags might be considered an advanced concept in some circumstances that should not be required for simple tasks.
- Use caution if you define a negative number as a flag enumerated constant because many flag positions might be set to 1, which might make your code confusing and encourage coding errors.

- A convenient way to test whether a flag is set in a numeric value is to perform a bitwise AND operation between the numeric value and the flag enumerated constant, which sets all bits in the numeric value to zero that do not correspond to the flag, then test whether the result of that operation is equal to the flag enumerated constant.
- Use `None` as the name of the flag enumerated constant whose value is zero. You cannot use the `None` enumerated constant in a bitwise AND operation to test for a flag because the result is always zero. However, you can perform a logical, not a bitwise, comparison between the numeric value and the `None` enumerated constant to determine whether any bits in the numeric value are set.

If you create a value enumeration instead of a flags enumeration, it is still worthwhile to create a `None` enumerated constant. The reason is that by default the memory used for the enumeration is initialized to zero by the common language runtime. Consequently, if you do not define a constant whose value is zero, the enumeration will contain an illegal value when it is created.

If there is an obvious default case your application needs to represent, consider using an enumerated constant whose value is zero to represent the default. If there is no default case, consider using an enumerated constant whose value is zero that means the case that is not represented by any of the other enumerated constants.

- Do not define an enumeration value solely to mirror the state of the enumeration itself. For example, do not define an enumerated constant that merely marks the end of the enumeration. If you need to determine the last value of the enumeration, check for that value explicitly. In addition, you can perform a range check for the first and last enumerated constant if all values within the range are valid.
- Do not specify enumerated constants that are reserved for future use.
- When you define a method or property that takes an enumerated constant as a value, consider validating the value. The reason is that you can cast a numeric value to the enumeration type even if that numeric value is not defined in the enumeration.

## Examples

The following example illustrates the use of the `FlagsAttribute` attribute and shows the effect on the `ToString` method of using `FlagsAttribute` on an `Enum` declaration.

C#

```
using System;

class Example
{
    // Define an Enum without FlagsAttribute.
    enum SingleHue : short
    {
        None = 0,
        Black = 1,
        Red = 2,
        Green = 4,
        Blue = 8
    };

    // Define an Enum with FlagsAttribute.
    [Flags]
    enum MultiHue : short
    {
        None = 0,
        Black = 1,
        Red = 2,
        Green = 4,
        Blue = 8
    };

    static void Main()
    {
        // Display all possible combinations of values.
        Console.WriteLine(
            "All possible combinations of values without FlagsAttribute:");
        for (int val = 0; val <= 16; val++)
            Console.WriteLine("{0,3} - {1:G}", val, (SingleHue)val);

        // Display all combinations of values, and invalid values.
        Console.WriteLine(
            "\nAll possible combinations of values with FlagsAttribute:");
        for (int val = 0; val <= 16; val++)
            Console.WriteLine("{0,3} - {1:G}", val, (MultiHue)val);
    }
}

// The example displays the following output:
//      All possible combinations of values without FlagsAttribute:
//          0 - None
//          1 - Black
//          2 - Red
//          3 - 3
//          4 - Green
//          5 - 5
//          6 - 6
//          7 - 7
//          8 - Blue
//          9 - 9
```

```

//      10 - 10
//      11 - 11
//      12 - 12
//      13 - 13
//      14 - 14
//      15 - 15
//      16 - 16
//
//      All possible combinations of values with FlagsAttribute:
//      0 - None
//      1 - Black
//      2 - Red
//      3 - Black, Red
//      4 - Green
//      5 - Black, Green
//      6 - Red, Green
//      7 - Black, Red, Green
//      8 - Blue
//      9 - Black, Blue
//      10 - Red, Blue
//      11 - Black, Red, Blue
//      12 - Green, Blue
//      13 - Black, Green, Blue
//      14 - Red, Green, Blue
//      15 - Black, Red, Green, Blue
//      16 - 16

```

The preceding example defines two color-related enumerations, `SingleHue` and `MultiHue`. The latter has the `FlagsAttribute` attribute; the former does not. The example shows the difference in behavior when a range of integers, including integers that do not represent underlying values of the enumeration type, are cast to the enumeration type and their string representations displayed. For example, note that 3 cannot be represented as a `SingleHue` value because 3 is not the underlying value of any `SingleHue` member, whereas the `FlagsAttribute` attribute makes it possible to represent 3 as a `MultiHue` value of `Black, Red`.

The following example defines another enumeration with the `FlagsAttribute` attribute and shows how to use bitwise logical and equality operators to determine whether one or more bit fields are set in an enumeration value. You can also use the `Enum.HasFlag` method to do that, but that is not shown in this example.

C#

```

using System;

[Flags]
public enum PhoneService
{
    None = 0,

```

```

        LandLine = 1,
        Cell = 2,
        Fax = 4,
        Internet = 8,
        Other = 16
    }

    public class Example1
    {
        public static void Main()
        {
            // Define three variables representing the types of phone service
            // in three households.
            var household1 = PhoneService.LandLine | PhoneService.Cell |
                PhoneService.Internet;
            var household2 = PhoneService.None;
            var household3 = PhoneService.Cell | PhoneService.Internet;

            // Store the variables in an array for ease of access.
            PhoneService[] households = { household1, household2, household3 };

            // Which households have no service?
            for (int ctr = 0; ctr < households.Length; ctr++)
                Console.WriteLine("Household {0} has phone service: {1}",
                    ctr + 1,
                    households[ctr] == PhoneService.None ?
                        "No" : "Yes");
            Console.WriteLine();

            // Which households have cell phone service?
            for (int ctr = 0; ctr < households.Length; ctr++)
                Console.WriteLine("Household {0} has cell phone service: {1}",
                    ctr + 1,
                    (households[ctr] & PhoneService.Cell) ==
PhoneService.Cell ?
                        "Yes" : "No");
            Console.WriteLine();

            // Which households have cell phones and land lines?
            var cellAndLand = PhoneService.Cell | PhoneService.LandLine;
            for (int ctr = 0; ctr < households.Length; ctr++)
                Console.WriteLine("Household {0} has cell and land line service:
{1}",
                    ctr + 1,
                    (households[ctr] & cellAndLand) == cellAndLand ?
                        "Yes" : "No");
            Console.WriteLine();

            // List all types of service of each household?//
            for (int ctr = 0; ctr < households.Length; ctr++)
                Console.WriteLine("Household {0} has: {1:G}",
                    ctr + 1, households[ctr]);
            Console.WriteLine();
        }
    }

```

```
}

// The example displays the following output:
// Household 1 has phone service: Yes
// Household 2 has phone service: No
// Household 3 has phone service: Yes
//
// Household 1 has cell phone service: Yes
// Household 2 has cell phone service: No
// Household 3 has cell phone service: Yes
//
// Household 1 has cell and land line service: Yes
// Household 2 has cell and land line service: No
// Household 3 has cell and land line service: No
//
// Household 1 has: LandLine, Cell, Internet
// Household 2 has: None
// Household 3 has: Cell, Internet
```

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Handle and raise events

Article • 10/04/2022

Events in .NET are based on the delegate model. The delegate model follows the [observer design pattern](#), which enables a subscriber to register with and receive notifications from a provider. An event sender pushes a notification that an event has happened, and an event receiver receives that notification and defines a response to it. This article describes the major components of the delegate model, how to consume events in applications, and how to implement events in your code.

## Events

An event is a message sent by an object to signal the occurrence of an action. The action can be caused by user interaction, such as a button click, or it can result from some other program logic, such as changing a property's value. The object that raises the event is called the *event sender*. The event sender doesn't know which object or method will receive (handle) the events it raises. The event is typically a member of the event sender; for example, the [Click](#) event is a member of the [Button](#) class, and the [PropertyChanged](#) event is a member of the class that implements the [INotifyPropertyChanged](#) interface.

To define an event, you use the C# [event](#) or the Visual Basic [Event](#) keyword in the signature of your event class, and specify the type of delegate for the event. Delegates are described in the next section.

Typically, to raise an event, you add a method that is marked as `protected` and `virtual` (in C#) or `Protected` and `Overridable` (in Visual Basic). Name this method `On EventName`; for example, `OnDataReceived`. The method should take one parameter that specifies an event data object, which is an object of type [EventArgs](#) or a derived type. You provide this method to enable derived classes to override the logic for raising the event. A derived class should always call the `On EventName` method of the base class to ensure that registered delegates receive the event.

The following example shows how to declare an event named `ThresholdReached`. The event is associated with the [EventHandler](#) delegate and raised in a method named `OnThresholdReached`.

C#

```
class Counter
{
```

```
public event EventHandler ThresholdReached;

protected virtual void OnThresholdReached(EventArgs e)
{
    ThresholdReached?.Invoke(this, e);
}

// provide remaining implementation for the class
}
```

## Delegates

A delegate is a type that holds a reference to a method. A delegate is declared with a signature that shows the return type and parameters for the methods it references, and it can hold references only to methods that match its signature. A delegate is thus equivalent to a type-safe function pointer or a callback. A delegate declaration is sufficient to define a delegate class.

Delegates have many uses in .NET. In the context of events, a delegate is an intermediary (or pointer-like mechanism) between the event source and the code that handles the event. You associate a delegate with an event by including the delegate type in the event declaration, as shown in the example in the previous section. For more information about delegates, see the [Delegate](#) class.

.NET provides the [EventHandler](#) and [EventHandler<TEventArgs>](#) delegates to support most event scenarios. Use the [EventHandler](#) delegate for all events that don't include event data. Use the [EventHandler<TEventArgs>](#) delegate for events that include data about the event. These delegates have no return type value and take two parameters (an object for the source of the event and an object for event data).

Delegates are [multicast](#), which means that they can hold references to more than one event-handling method. For more information, see the [Delegate](#) reference page. Delegates provide flexibility and fine-grained control in event handling. A delegate acts as an event dispatcher for the class that raises the event by maintaining a list of registered event handlers for the event.

For scenarios where the [EventHandler](#) and [EventHandler<TEventArgs>](#) delegates don't work, you can define a delegate. Scenarios that require you to define a delegate are rare, such as when you must work with code that doesn't recognize generics. You mark a delegate with the C# [delegate](#) and Visual Basic [Delegate](#) keyword in the declaration. The following example shows how to declare a delegate named

ThresholdReachedEventHandler:

C#

```
public delegate void ThresholdReachedEventHandler(object sender,  
ThresholdReachedEventArgs e);
```

## Event data

Data that is associated with an event can be provided through an event data class. .NET provides many event data classes that you can use in your applications. For example, the [SerialDataReceivedEventArgs](#) class is the event data class for the [SerialPort.DataReceived](#) event. .NET follows a naming pattern of ending all event data classes with `EventArgs`. You determine which event data class is associated with an event by looking at the delegate for the event. For example, the [SerialDataReceivedEventHandler](#) delegate includes the [SerialDataReceivedEventArgs](#) class as one of its parameters.

The [EventArgs](#) class is the base type for all event data classes. [EventArgs](#) is also the class you use when an event doesn't have any data associated with it. When you create an event that is only meant to notify other classes that something happened and doesn't need to pass any data, include the [EventArgs](#) class as the second parameter in the delegate. You can pass the [EventArgs.Empty](#) value when no data is provided. The [EventHandler](#) delegate includes the [EventArgs](#) class as a parameter.

When you want to create a customized event data class, create a class that derives from [EventArgs](#), and then provide any members needed to pass data that is related to the event. Typically, you should use the same naming pattern as .NET and end your event data class name with `EventArgs`.

The following example shows an event data class named `ThresholdReachedEventArgs`. It contains properties that are specific to the event being raised:

C#

```
public class ThresholdReachedEventArgs : EventArgs  
{  
    public int Threshold { get; set; }  
    public DateTime TimeReached { get; set; }  
}
```

## Event handlers

To respond to an event, you define an event handler method in the event receiver. This method must match the signature of the delegate for the event you're handling. In the

event handler, you perform the actions that are required when the event is raised, such as collecting user input after the user clicks a button. To receive notifications when the event occurs, your event handler method must subscribe to the event.

The following example shows an event handler method named `c_ThresholdReached` that matches the signature for the `EventHandler` delegate. The method subscribes to the `ThresholdReached` event.

```
C#  
  
class ProgramTwo  
{  
    static void Main()  
    {  
        var c = new Counter();  
        c.ThresholdReached += c_ThresholdReached;  
  
        // provide remaining implementation for the class  
    }  
  
    static void c_ThresholdReached(object sender, EventArgs e)  
    {  
        Console.WriteLine("The threshold was reached.");  
    }  
}
```

## Static and dynamic event handlers

.NET allows subscribers to register for event notifications either statically or dynamically. Static event handlers are in effect for the entire life of the class whose events they handle. Dynamic event handlers are explicitly activated and deactivated during program execution, usually in response to some conditional program logic. For example, they can be used if event notifications are needed only under certain conditions or if an application provides multiple event handlers and run-time conditions define the appropriate one to use. The example in the previous section shows how to dynamically add an event handler. For more information, see [Events](#) (in Visual Basic) and [Events](#) (in C#).

## Raising multiple events

If your class raises multiple events, the compiler generates one field per event delegate instance. If the number of events is large, the storage cost of one field per delegate might not be acceptable. For those situations, .NET provides event properties that you can use with another data structure of your choice to store event delegates.

Event properties consist of event declarations accompanied by event accessors. Event accessors are methods that you define to add or remove event delegate instances from the storage data structure.

### Note

The event properties are slower than the event fields because each event delegate must be retrieved before it can be invoked.

The trade-off is between memory and speed. If your class defines many events that are infrequently raised, you'll want to implement event properties. For more information, see [How to: Handle Multiple Events Using Event Properties](#).

## Related articles

Title	Description
<a href="#">How to: Raise and Consume Events</a>	Contains examples of raising and consuming events.
<a href="#">How to: Handle Multiple Events Using Event Properties</a>	Shows how to use event properties to handle multiple events.
<a href="#">Observer Design Pattern</a>	Describes the design pattern that enables a subscriber to register with and receive notifications from a provider.

## See also

- [EventHandler](#)
- [EventHandler<TEventArgs>](#)
- [EventArgs](#)
- [Delegate](#)
- [Events \(Visual Basic\)](#)
- [Events \(C# Programming Guide\)](#)
- [Events and routed events overview \(UWP apps\)](#)
- [Events in Windows Store 8.x apps](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Raise and Consume Events

Article • 10/04/2022

The examples in this article show how to work with events. They include examples of the [EventHandler](#) delegate, the [EventHandler<TEventArgs>](#) delegate, and a custom delegate to illustrate events with and without data.

The examples use concepts described in the [Events](#) article.

## Example 1

The first example shows how to raise and consume an event that doesn't have data. It contains a class named `Counter` that has an event called `ThresholdReached`. This event is raised when a counter value equals or exceeds a threshold value. The [EventHandler](#) delegate is associated with the event because no event data is provided.

C#

```
using System;

namespace ConsoleApplication1
{
    class ProgramOne
    {
        static void Main(string[] args)
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
        }

        static void c_ThresholdReached(object sender, EventArgs e)
        {
            Console.WriteLine("The threshold was reached.");
            Environment.Exit(0);
        }
    }

    class Counter
    {
        private int threshold;
        private int total;
```

```

public Counter(int passedThreshold)
{
    threshold = passedThreshold;
}

public void Add(int x)
{
    total += x;
    if (total >= threshold)
    {
        ThresholdReached?.Invoke(this, EventArgs.Empty);
    }
}

public event EventHandler ThresholdReached;
}
}

```

## Example 2

The second example shows how to raise and consume an event that provides data. The `EventHandler<TEventArgs>` delegate is associated with the event, and an instance of a custom event data object is provided.

C#

```

using System;

namespace ConsoleApplication3
{
    class ProgramThree
    {
        static void Main(string[] args)
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
        }

        static void c_ThresholdReached(object sender,
ThresholdReachedEventArgs e)
        {
            Console.WriteLine("The threshold of {0} was reached at {1}.",
e.Threshold, e.TimeReached);
        }
    }
}

```

```

        Environment.Exit(0);
    }
}

class Counter
{
    private int threshold;
    private int total;

    public Counter(int passedThreshold)
    {
        threshold = passedThreshold;
    }

    public void Add(int x)
    {
        total += x;
        if (total >= threshold)
        {
            ThresholdReachedEventArgs args = new
ThresholdReachedEventArgs();
            args.Threshold = threshold;
            args.TimeReached = DateTime.Now;
            OnThresholdReached(args);
        }
    }

    protected virtual void OnThresholdReached(ThresholdReachedEventArgs
e)
    {
        EventHandler<ThresholdReachedEventArgs> handler =
ThresholdReached;
        if (handler != null)
        {
            handler(this, e);
        }
    }
}

public event EventHandler<ThresholdReachedEventArgs>
ThresholdReached;
}

public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}
}

```

## Example 3

The third example shows how to declare a delegate for an event. The delegate is named `ThresholdReachedEventHandler`. This example is just an illustration. Typically, you don't have to declare a delegate for an event because you can use either the `EventHandler` or the `EventHandler<TEventArgs>` delegate. You should declare a delegate only in rare scenarios, such as making your class available to legacy code that can't use generics.

C#

```
using System;

namespace ConsoleApplication4
{
    class ProgramFour
    {
        static void Main(string[] args)
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
        }

        static void c_ThresholdReached(Object sender,
ThresholdReachedEventArgs e)
        {
            Console.WriteLine("The threshold of {0} was reached at {1}.",
e.Threshold, e.TimeReached);
            Environment.Exit(0);
        }
    }

    class Counter
    {
        private int threshold;
        private int total;

        public Counter(int passedThreshold)
        {
            threshold = passedThreshold;
        }

        public void Add(int x)
        {
            total += x;
            if (total >= threshold)
            {
                ThresholdReachedEventArgs args = new
```

```
ThresholdReachedEventArgs();
    args.Threshold = threshold;
    args.TimeReached = DateTime.Now;
    OnThresholdReached(args);
}
}

protected virtual void OnThresholdReached(ThresholdReachedEventArgs
e)
{
    ThresholdReachedEventHandler handler = ThresholdReached;
    if (handler != null)
    {
        handler(this, e);
    }
}

public event ThresholdReachedEventHandler ThresholdReached;
}

public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}

public delegate void ThresholdReachedEventHandler(Object sender,
ThresholdReachedEventArgs e);
}
```

## See also

- [Events](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Handle Multiple Events Using Event Properties

Article • 09/15/2021

To use event properties, you define the event properties in the class that raises the events, and then set the delegates for the event properties in classes that handle the events. To implement multiple event properties in a class, the class should internally store and maintain the delegate defined for each event. For each field-like-event, a corresponding backing-field reference type is generated. This can lead to unnecessary allocations when the number of events increases. As an alternative, a common approach is to maintain an [EventHandlerList](#) which stores events by key.

To store the delegates for each event, you can use the [EventHandlerList](#) class, or implement your own collection. The collection class must provide methods for setting, accessing, and retrieving the event handler delegate based on the event key. For example, you could use a [Hashtable](#) class, or derive a custom class from the [DictionaryBase](#) class. The implementation details of the delegate collection do not need to be exposed outside your class.

Each event property within the class defines an add accessor method and a remove accessor method. The add accessor for an event property adds the input delegate instance to the delegate collection. The remove accessor for an event property removes the input delegate instance from the delegate collection. The event property accessors use the predefined key for the event property to add and remove instances from the delegate collection.

## To handle multiple events using event properties

1. Define a delegate collection within the class that raises the events.
2. Define a key for each event.
3. Define the event properties in the class that raises the events.
4. Use the delegate collection to implement the add and remove accessor methods for the event properties.
5. Use the public event properties to add and remove event handler delegates in the classes that handle the events.

# Example

The following C# example implements the event properties `MouseDown` and `MouseUp`, using an `EventHandlerList` to store each event's delegate. The keywords of the event property constructs are in bold type.

C#

```
// The class SampleControl defines two event properties, MouseUp and
MouseDown.
class SampleControl : Component
{
    // :
    // Define other control methods and properties.
    // :

    // Define the delegate collection.
protected EventHandlerList listEventDelegates = new EventHandlerList();

    // Define a unique key for each event.
static readonly object mouseDownEventKey = new object();
static readonly object mouseUpEventKey = new object();

    // Define the MouseDown event property.
public event MouseEventHandler MouseDown
{
    // Add the input delegate to the collection.
    add
    {
        listEventDelegates.AddHandler(mouseDownEventKey, value);
    }
    // Remove the input delegate from the collection.
    remove
    {
        listEventDelegates.RemoveHandler(mouseDownEventKey, value);
    }
}

    // Raise the event with the delegate specified by mouseDownEventKey
private void OnMouseDown(MouseEventArgs e)
{
    MouseEventHandler mouseEventDelegate =
        (MouseEventHandler)listEventDelegates[mouseDownEventKey];
    mouseEventDelegate(this, e);
}

    // Define the MouseUp event property.
public event MouseEventHandler MouseUp
{
    // Add the input delegate to the collection.
    add
    {
```

```
        listEventDelegates.AddHandler(mouseUpEventKey, value);
    }
    // Remove the input delegate from the collection.
    remove
    {
        listEventDelegates.RemoveHandler(mouseUpEventKey, value);
    }
}

// Raise the event with the delegate specified by mouseUpEventKey
private void OnMouseUp(MouseEventArgs e)
{
    MouseEventHandler mouseEventDelegate =
        (MouseEventHandler)listEventDelegates[mouseUpEventKey];
    mouseEventDelegate(this, e);
}
}
```

## See also

- [System.ComponentModel.EventHandlerList](#)
- [Events](#)
- [Control.Events](#)
- [How to: Declare Custom Events To Conserve Memory](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Observer design pattern

Article • 05/25/2023

The observer design pattern enables a subscriber to register with and receive notifications from a provider. It's suitable for any scenario that requires push-based notification. The pattern defines a *provider* (also known as a *subject* or an *observable*) and zero, one, or more *observers*. Observers register with the provider, and whenever a predefined condition, event, or state change occurs, the provider automatically notifies all observers by invoking a delegate. In this method call, the provider can also provide current state information to observers. In .NET, the observer design pattern is applied by implementing the generic [System.IObservable<T>](#) and [System.IObserver<T>](#) interfaces. The generic type parameter represents the type that provides notification information.

## When to apply the pattern

The observer design pattern is suitable for distributed push-based notifications, because it supports a clean separation between two different components or application layers, such as a data source (business logic) layer and a user interface (display) layer. The pattern can be implemented whenever a provider uses callbacks to supply its clients with current information.

Implementing the pattern requires that you provide the following details:

- A provider or subject, which is the object that sends notifications to observers. A provider is a class or structure that implements the [IObservable<T>](#) interface. The provider must implement a single method, [IObservable<T>.Subscribe](#), which is called by observers that wish to receive notifications from the provider.
- An observer, which is an object that receives notifications from a provider. An observer is a class or structure that implements the [IObserver<T>](#) interface. The observer must implement three methods, all of which are called by the provider:
  - [IObserver<T>.OnNext](#), which supplies the observer with new or current information.
  - [IObserver<T>.OnError](#), which informs the observer that an error has occurred.
  - [IObserver<T>.OnCompleted](#), which indicates that the provider has finished sending notifications.
- A mechanism that allows the provider to keep track of observers. Typically, the provider uses a container object, such as a [System.Collections.Generic.List<T>](#) object, to hold references to the [IObserver<T>](#) implementations that have subscribed to notifications. Using a storage container for this purpose enables the

provider to handle zero to an unlimited number of observers. The order in which observers receive notifications isn't defined; the provider is free to use any method to determine the order.

- An [IDisposable](#) implementation that enables the provider to remove observers when notification is complete. Observers receive a reference to the [IDisposable](#) implementation from the [Subscribe](#) method, so they can also call the [IDisposable.Dispose](#) method to unsubscribe before the provider has finished sending notifications.
- An object that contains the data that the provider sends to its observers. The type of this object corresponds to the generic type parameter of the [IObservable<T>](#) and [IObserver<T>](#) interfaces. Although this object can be the same as the [IObservable<T>](#) implementation, most commonly it's a separate type.

#### ⓘ Note

In addition to implementing the observer design pattern, you may be interested in exploring libraries that are built using the [IObservable<T>](#) and [IObserver<T>](#) interfaces. For example, [Reactive Extensions for .NET \(Rx\)](#) consist of a set of extension methods and LINQ standard sequence operators to support asynchronous programming.

## Implement the pattern

The following example uses the observer design pattern to implement an airport baggage claim information system. A `BaggageInfo` class provides information about arriving flights and the carousels where baggage from each flight is available for pickup. It's shown in the following example.

C#

```
namespace Observables.Example;

public readonly record struct BaggageInfo(
    int FlightNumber,
    string From,
    int Carousel);
```

A `BaggageHandler` class is responsible for receiving information about arriving flights and baggage claim carousels. Internally, it maintains two collections:

- `_observers`: A collection of clients that observe updated information.
- `_flights`: A collection of flights and their assigned carousels.

The source code for the `BaggageHandler` class is shown in the following example.

C#

```
namespace Observables.Example;

public sealed class BaggageHandler : IObservable<BaggageInfo>
{
    private readonly HashSet<IObserver<BaggageInfo>> _observers = new();
    private readonly HashSet<BaggageInfo> _flights = new();

    public IDisposable Subscribe(IObserver<BaggageInfo> observer)
    {
        // Check whether observer is already registered. If not, add it.
        if (_observers.Add(observer))
        {
            // Provide observer with existing data.
            foreach (BaggageInfo item in _flights)
            {
                observer.OnNext(item);
            }
        }

        return new Unsubscriber<BaggageInfo>(_observers, observer);
    }

    // Called to indicate all baggage is now unloaded.
    public void BaggageStatus(int flightNumber) =>
        BaggageStatus(flightNumber, string.Empty, 0);

    public void BaggageStatus(int flightNumber, string from, int carousel)
    {
        var info = new BaggageInfo(flightNumber, from, carousel);

        // Carousel is assigned, so add new info object to list.
        if (carousel > 0 && _flights.Add(info))
        {
            foreach (IObserver<BaggageInfo> observer in _observers)
            {
                observer.OnNext(info);
            }
        }
        else if (carousel is 0)
        {
            // Baggage claim for flight is done.
            if (_flights.RemoveWhere(
                flight => flight.FlightNumber == info.FlightNumber) > 0)
            {
                foreach (IObserver<BaggageInfo> observer in _observers)
                {

```

```

        observer.OnNext(info);
    }
}
}

public void LastBaggageClaimed()
{
    foreach (I0server<BaggageInfo> observer in _observers)
    {
        observer.OnCompleted();
    }

    _observers.Clear();
}
}

```

Clients that wish to receive updated information call the `BaggageHandler.Subscribe` method. If the client hasn't previously subscribed to notifications, a reference to the client's `I0server<T>` implementation is added to the `_observers` collection.

The overloaded `BaggageHandler.BaggageStatus` method can be called to indicate that baggage from a flight either is being unloaded or is no longer being unloaded. In the first case, the method is passed a flight number, the airport from which the flight originated, and the carousel where baggage is being unloaded. In the second case, the method is passed only a flight number. For baggage that is being unloaded, the method checks whether the `BaggageInfo` information passed to the method exists in the `_flights` collection. If it doesn't, the method adds the information and calls each observer's `OnNext` method. For flights whose baggage is no longer being unloaded, the method checks whether information on that flight is stored in the `_flights` collection. If it is, the method calls each observer's `OnNext` method and removes the `BaggageInfo` object from the `_flights` collection.

When the last flight of the day has landed and its baggage has been processed, the `BaggageHandler.LastBaggageClaimed` method is called. This method calls each observer's `OnCompleted` method to indicate that all notifications have completed, and then clears the `_observers` collection.

The provider's `Subscribe` method returns an `IDisposable` implementation that enables observers to stop receiving notifications before the `OnCompleted` method is called. The source code for this `Unsubscriber<Of BaggageInfo>` class is shown in the following example. When the class is instantiated in the `BaggageHandler.Subscribe` method, it's passed a reference to the `_observers` collection and a reference to the observer that is added to the collection. These references are assigned to local variables. When the

object's `Dispose` method is called, it checks whether the observer still exists in the `_observers` collection, and, if it does, removes the observer.

C#

```
namespace Observables.Example;

internal sealed class Unsubscriber<BaggageInfo> : IDisposable
{
    private readonly ISet<IObserver<BaggageInfo>> _observers;
    private readonly IObserver<BaggageInfo> _observer;

    internal Unsubscriber(
        ISet<IObserver<BaggageInfo>> observers,
        IObserver<BaggageInfo> observer) => (_observers, _observer) =
(observers, observer);

    public void Dispose() => _observers.Remove(_observer);
}
```

The following example provides an `IObserver<T>` implementation named `ArrivalsMonitor`, which is a base class that displays baggage claim information. The information is displayed alphabetically, by the name of the originating city. The methods of `ArrivalsMonitor` are marked as `overridable` (in Visual Basic) or `virtual` (in C#), so they can be overridden in a derived class.

C#

```
namespace Observables.Example;

public class ArrivalsMonitor : IObserver<BaggageInfo>
{
    private readonly string _name;
    private readonly List<string> _flights = new();
    private readonly string _format = "{0,-20} {1,5} {2, 3}";
    private IDisposable? _cancellation;

    public ArrivalsMonitor(string name)
    {
        ArgumentException.ThrowIfNullOrEmpty(name);
        _name = name;
    }

    public virtual void Subscribe(BaggageHandler provider) =>
        _cancellation = provider.Subscribe(this);

    public virtual void Unsubscribe()
    {
        _cancellation?.Dispose();
        _flights.Clear();
    }
}
```

```
}

public virtual void OnCompleted() => _flights.Clear();

// No implementation needed: Method is not called by the BaggageHandler
class.
public virtual void OnError(Exception e)
{
    // No implementation.
}

// Update information.
public virtual void OnNext(BaggageInfo info)
{
    bool updated = false;

    // Flight has unloaded its baggage; remove from the monitor.
    if (info.Carousel is 0)
    {
        string flightNumber = string.Format("{0,5}", info.FlightNumber);
        for (int index = _flights.Count - 1; index >= 0; index--)
        {
            string flightInfo = _flights[index];
            if (flightInfo.Substring(21, 5).Equals(flightNumber))
            {
                updated = true;
                _flights.RemoveAt(index);
            }
        }
    }
    else
    {
        // Add flight if it doesn't exist in the collection.
        string flightInfo = string.Format(_format, info.From,
info.FlightNumber, info.Carousel);
        if (_flights.Contains(flightInfo) is false)
        {
            _flights.Add(flightInfo);
            updated = true;
        }
    }

    if (updated)
    {
        _flights.Sort();
        Console.WriteLine($"Arrivals information from {_name}");
        foreach (string flightInfo in _flights)
        {
            Console.WriteLine(flightInfo);
        }

        Console.WriteLine();
    }
}
}
```

The `ArrivalsMonitor` class includes the `Subscribe` and `Unsubscribe` methods. The `Subscribe` method enables the class to save the `IDisposable` implementation returned by the call to `Subscribe` to a private variable. The `Unsubscribe` method enables the class to unsubscribe from notifications by calling the provider's `Dispose` implementation. `ArrivalsMonitor` also provides implementations of the `OnNext`, `OnError`, and `OnCompleted` methods. Only the `OnNext` implementation contains a significant amount of code. The method works with a private, sorted, generic `List<T>` object that maintains information about the airports of origin for arriving flights and the carousels on which their baggage is available. If the `BaggageHandler` class reports a new flight arrival, the `OnNext` method implementation adds information about that flight to the list. If the `BaggageHandler` class reports that the flight's baggage has been unloaded, the `OnNext` method removes that flight from the list. Whenever a change is made, the list is sorted and displayed to the console.

The following example contains the application entry point that instantiates the `BaggageHandler` class and two instances of the `ArrivalsMonitor` class, and uses the `BaggageHandler.BaggageStatus` method to add and remove information about arriving flights. In each case, the observers receive updates and correctly display baggage claim information.

C#

```
using Observables.Example;

BaggageHandler provider = new();
ArrivalsMonitor observer1 = new("BaggageClaimMonitor1");
ArrivalsMonitor observer2 = new("SecurityExit");

provider.BaggageStatus(712, "Detroit", 3);
observer1.Subscribe(provider);

provider.BaggageStatus(712, "Kalamazoo", 3);
provider.BaggageStatus(400, "New York-Kennedy", 1);
provider.BaggageStatus(712, "Detroit", 3);
observer2.Subscribe(provider);

provider.BaggageStatus(511, "San Francisco", 2);
provider.BaggageStatus(712);
observer2.Unsubscribe();

provider.BaggageStatus(400);
provider.LastBaggageClaimed();

// Sample output:
//   Arrivals information from BaggageClaimMonitor1
//   Detroit          712      3
```

```

// Arrivals information from BaggageClaimMonitor1
// Detroit 712 3
// Kalamazoo 712 3
//
// Arrivals information from BaggageClaimMonitor1
// Detroit 712 3
// Kalamazoo 712 3
// New York-Kennedy 400 1
//
// Arrivals information from SecurityExit
// Detroit 712 3
//
// Arrivals information from SecurityExit
// Detroit 712 3
// Kalamazoo 712 3
//
// Arrivals information from SecurityExit
// Detroit 712 3
// Kalamazoo 712 3
// New York-Kennedy 400 1
//
// Arrivals information from BaggageClaimMonitor1
// Detroit 712 3
// Kalamazoo 712 3
// New York-Kennedy 400 1
// San Francisco 511 2
//
// Arrivals information from SecurityExit
// Detroit 712 3
// Kalamazoo 712 3
// New York-Kennedy 400 1
// San Francisco 511 2
//
// Arrivals information from BaggageClaimMonitor1
// New York-Kennedy 400 1
// San Francisco 511 2
//
// Arrivals information from SecurityExit
// New York-Kennedy 400 1
// San Francisco 511 2
//
// Arrivals information from BaggageClaimMonitor1
// San Francisco 511 2

```

## Related articles

Title	Description
<a href="#">Observer Design Pattern Best Practices</a>	Describes best practices to adopt when developing applications that implement the observer design pattern.

Title	Description
<a href="#">How to: Implement a Provider</a>	Provides a step-by-step implementation of a provider for a temperature monitoring application.
<a href="#">How to: Implement an Observer</a>	Provides a step-by-step implementation of an observer for a temperature monitoring application.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Observer Design Pattern Best Practices

Article • 09/15/2021

In .NET, the observer design pattern is implemented as a set of interfaces. The [System.IObservable<T>](#) interface represents the data provider, which is also responsible for providing an [IDisposable](#) implementation that lets observers unsubscribe from notifications. The [System.IObserver<T>](#) interface represents the observer. This topic describes the best practices that developers should follow when implementing the observer design pattern using these interfaces.

## Threading

Typically, a provider implements the [IObservable<T>.Subscribe](#) method by adding a particular observer to a subscriber list that is represented by some collection object, and it implements the [IDisposable.Dispose](#) method by removing a particular observer from the subscriber list. An observer can call these methods at any time. In addition, because the provider/observer contract does not specify who is responsible for unsubscribing after the [IObserver<T>.OnCompleted](#) callback method, the provider and observer may both try to remove the same member from the list. Because of this possibility, both the [Subscribe](#) and [Dispose](#) methods should be thread-safe. Typically, this involves using a [concurrent collection](#) or a lock. Implementations that are not thread-safe should explicitly document that they are not.

Any additional guarantees have to be specified in a layer on top of the provider/observer contract. Implementers should clearly call out when they impose additional requirements to avoid user confusion about the observer contract.

## Handling Exceptions

Because of the loose coupling between a data provider and an observer, exceptions in the observer design pattern are intended to be informational. This affects how providers and observers handle exceptions in the observer design pattern.

## The Provider—Calling the [OnError](#) Method

The [OnError](#) method is intended as an informational message to observers, much like the [IObserver<T>.OnNext](#) method. However, the [OnNext](#) method is designed to provide an observer with current or updated data, whereas the [OnError](#) method is designed to indicate that the provider is unable to provide valid data.

The provider should follow these best practices when handling exceptions and calling the [OnError](#) method:

- The provider must handle its own exceptions if it has any specific requirements.
- The provider should not expect or require that observers handle exceptions in any particular way.
- The provider should call the [OnError](#) method when it handles an exception that compromises its ability to provide updates. Information on such exceptions can be passed to the observer. In other cases, there is no need to notify observers of an exception.

Once the provider calls the [OnError](#) or [IObserver<T>.OnCompleted](#) method, there should be no further notifications, and the provider can unsubscribe its observers. However, the observers can also unsubscribe themselves at any time, including both before and after they receive an [OnError](#) or [IObserver<T>.OnCompleted](#) notification. The observer design pattern does not dictate whether the provider or the observer is responsible for unsubscribing; therefore, there is a possibility that both may attempt to unsubscribe. Typically, when observers unsubscribe, they are removed from a subscribers collection. In a single-threaded application, the [IDisposable.Dispose](#) implementation should ensure that an object reference is valid and that the object is a member of the subscribers collection before attempting to remove it. In a multithreaded application, a thread-safe collection object, such as a [System.Collections.Concurrent.BlockingCollection<T>](#) object, should be used.

## The Observer—Implementing the OnError Method

When an observer receives an error notification from a provider, the observer should treat the exception as informational and should not be required to take any particular action.

The observer should follow these best practices when responding to an [OnError](#) method call from a provider:

- The observer should not throw exceptions from its interface implementations, such as [OnNext](#) or [OnError](#). However, if the observer does throw exceptions, it should expect these exceptions to go unhandled.
- To preserve the call stack, an observer that wishes to throw an [Exception](#) object that was passed to its [OnError](#) method should wrap the exception before throwing it. A standard exception object should be used for this purpose.

# Additional Best Practices

Attempting to unregister in the `IObservable<T>.Subscribe` method may result in a null reference. Therefore, we recommend that you avoid this practice.

Although it is possible to attach an observer to multiple providers, the recommended pattern is to attach an `IObserver<T>` instance to only one `IObservable<T>` instance.

## See also

- [Observer Design Pattern](#)
- [How to: Implement an Observer](#)
- [How to: Implement a Provider](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Implement a Provider

Article • 09/15/2021

The observer design pattern requires a division between a provider, which monitors data and sends notifications, and one or more observers, which receive notifications (callbacks) from the provider. This topic discusses how to create a provider. A related topic, [How to: Implement an Observer](#), discusses how to create an observer.

## To create a provider

1. Define the data that the provider is responsible for sending to observers. Although the provider and the data that it sends to observers can be a single type, they are generally represented by different types. For example, in a temperature monitoring application, the `Temperature` structure defines the data that the provider (which is represented by the `TemperatureMonitor` class defined in the next step) monitors and to which observers subscribe.

C#

```
using System;

public struct Temperature
{
    private decimal temp;
    private DateTime tempDate;

    public Temperature(decimal temperature, DateTime dateAndTime)
    {
        this.temp = temperature;
        this.tempDate = dateAndTime;
    }

    public decimal Degrees
    { get { return this.temp; } }

    public DateTime Date
    { get { return this.tempDate; } }
}
```

2. Define the data provider, which is a type that implements the `System.IObservable<T>` interface. The provider's generic type argument is the type that the provider sends to observers. The following example defines a `TemperatureMonitor` class, which is a constructed `System.IObservable<T>` implementation with a generic type argument of `Temperature`.

C#

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

public class TemperatureMonitor : IObservable<Temperature>
{
```

3. Determine how the provider will store references to observers so that each observer can be notified when appropriate. Most commonly, a collection object such as a generic `List<T>` object is used for this purpose. The following example defines a private `List<T>` object that is instantiated in the `TemperatureMonitor` class constructor.

C#

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

public class TemperatureMonitor : IObservable<Temperature>
{
    List<IObserver<Temperature>> observers;

    public TemperatureMonitor()
    {
        observers = new List<IObserver<Temperature>>();
    }
```

4. Define an `IDisposable` implementation that the provider can return to subscribers so that they can stop receiving notifications at any time. The following example defines a nested `Unsubscriber` class that is passed a reference to the subscribers collection and to the subscriber when the class is instantiated. This code enables the subscriber to call the object's `IDisposable.Dispose` implementation to remove itself from the subscribers collection.

C#

```
private class Unsubscriber : IDisposable
{
    private List<IObserver<Temperature>> _observers;
    private IObserver<Temperature> _observer;

    public Unsubscriber(List<IObserver<Temperature>> observers,
IObserver<Temperature> observer)
    {
        this._observers = observers;
        this._observer = observer;
    }
```

```
public void Dispose()
{
    if (_observer == null) _observers.Remove(_observer);
}
}
```

5. Implement the `IObservable<T>.Subscribe` method. The method is passed a reference to the `System.IObserver<T>` interface and should be stored in the object designed for that purpose in step 3. The method should then return the `IDisposable` implementation developed in step 4. The following example shows the implementation of the `Subscribe` method in the `TemperatureMonitor` class.

C#

```
public IDisposable Subscribe(IObserver<Temperature> observer)
{
    if (!observers.Contains(observer))
        observers.Add(observer);

    return new Unsubscriber(observers, observer);
}
```

6. Notify observers as appropriate by calling their `IObserver<T>.OnNext`, `IObserver<T>.OnError`, and `IObserver<T>.OnCompleted` implementations. In some cases, a provider may not call the `OnError` method when an error occurs. For example, the following `GetTemperature` method simulates a monitor that reads temperature data every five seconds and notifies observers if the temperature has changed by at least .1 degree since the previous reading. If the device does not report a temperature (that is, if its value is null), the provider notifies observers that the transmission is complete. Note that, in addition to calling each observer's `OnCompleted` method, the `GetTemperature` method clears the `List<T>` collection. In this case, the provider makes no calls to the `OnError` method of its observers.

C#

```
public void GetTemperature()
{
    // Create an array of sample data to mimic a temperature device.
    Nullable<Decimal>[] temps = {14.6m, 14.65m, 14.7m, 14.9m, 14.9m,
    15.2m, 15.25m, 15.2m,
                           15.4m, 15.45m, null };

    // Store the previous temperature, so notification is only sent
    // after at least .1 change.
    Nullable<Decimal> previous = null;
    bool start = true;
```

```

        foreach (var temp in temps) {
            System.Threading.Thread.Sleep(2500);
            if (temp.HasValue) {
                if (start || (Math.Abs(temp.Value - previous.Value) >= 0.1m ))
                {
                    Temperature tempData = new Temperature(temp.Value,
                    DateTime.Now);
                    foreach (var observer in observers)
                        observer.OnNext(tempData);
                    previous = temp;
                    if (start) start = false;
                }
            }
            else {
                foreach (var observer in observers.ToArray())
                    if (observer != null) observer.OnCompleted();

                observers.Clear();
                break;
            }
        }
    }
}

```

## Example

The following example contains the complete source code for defining an `IObservable<T>` implementation for a temperature monitoring application. It includes the `Temperature` structure, which is the data sent to observers, and the `TemperatureMonitor` class, which is the `IObservable<T>` implementation.

C#

```

using System.Threading;
using System;
using System.Collections.Generic;

public class TemperatureMonitor : IObserverable<Temperature>
{
    List<IObserver<Temperature>> observers;

    public TemperatureMonitor()
    {
        observers = new List<IObserver<Temperature>>();
    }

    private class Unsubscriber : IDisposable
    {
        private List<IObserver<Temperature>> _observers;
        private IObserver<Temperature> _observer;
    }
}

```

```

    public Unsubscriber(List<IObserver<Temperature>> observers,
IObserver<Temperature> observer)
{
    this._observers = observers;
    this._observer = observer;
}

    public void Dispose()
{
    if (! (_observer == null)) _observers.Remove(_observer);
}
}

    public IDisposable Subscribe(IObserver<Temperature> observer)
{
    if (! observers.Contains(observer))
        observers.Add(observer);

    return new Unsubscriber(observers, observer);
}

    public void GetTemperature()
{
    // Create an array of sample data to mimic a temperature device.
    Nullable<Decimal>[] temps = {14.6m, 14.65m, 14.7m, 14.9m, 14.9m,
15.2m, 15.25m, 15.2m,
                                         15.4m, 15.45m, null };

    // Store the previous temperature, so notification is only sent after
at least .1 change.
    Nullable<Decimal> previous = null;
    bool start = true;

    foreach (var temp in temps) {
        System.Threading.Thread.Sleep(2500);
        if (temp.HasValue) {
            if (start || (Math.Abs(temp.Value - previous.Value) >= 0.1m )) {
                Temperature tempData = new Temperature(temp.Value,
DateTime.Now);
                foreach (var observer in observers)
                    observer.OnNext(tempData);
                previous = temp;
                if (start) start = false;
            }
        }
        else {
            foreach (var observer in observers.ToArray())
                if (observer != null) observer.OnCompleted();

            observers.Clear();
            break;
        }
    }
}
}

```

## See also

- [IObservable<T>](#)
- [Observer Design Pattern](#)
- [How to: Implement an Observer](#)
- [Observer Design Pattern Best Practices](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### [.NET feedback](#)

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Implement an Observer

Article • 11/10/2021

The observer design pattern requires a division between an observer, which registers for notifications, and a provider, which monitors data and sends notifications to one or more observers. This topic discusses how to create an observer. A related topic, [How to: Implement a Provider](#), discusses how to create a provider.

## To create an observer

1. Define the observer, which is a type that implements the `System.IObserver<T>` interface. For example, the following code defines a type named `TemperatureReporter` that is a constructed `System.IObserver<T>` implementation with a generic type argument of `Temperature`.

C#

```
public class TemperatureReporter : IObserver<Temperature>
```

2. If the observer can stop receiving notifications before the provider calls its `IObserver<T>.OnCompleted` implementation, define a private variable that will hold the `IDisposable` implementation returned by the provider's `IObservable<T>.Subscribe` method. You should also define a subscription method that calls the provider's `Subscribe` method and stores the returned `IDisposable` object. For example, the following code defines a private variable named `unsubscriber` and defines a `Subscribe` method that calls the provider's `Subscribe` method and assigns the returned object to the `unsubscriber` variable.

C#

```
public class TemperatureReporter : IObserver<Temperature>
{
    private IDisposable unsubscriber;
    private bool first = true;
    private Temperature last;

    public virtual void Subscribe(IObservable<Temperature> provider)
    {
        unsubscriber = provider.Subscribe(this);
    }
}
```

3. Define a method that enables the observer to stop receiving notifications before the provider calls its `IObserver<T>.OnCompleted` implementation, if this feature is required. The following example defines an `Unsubscribe` method.

C#

```
public virtual void Unsubscribe()
{
    subscriber.Dispose();
}
```

4. Provide implementations of the three methods defined by the `IObserver<T>` interface: `IObserver<T>.OnNext`, `IObserver<T>.OnError`, and `IObserver<T>.OnCompleted`. Depending on the provider and the needs of the application, the `OnError` and `OnCompleted` methods can be stub implementations. Note that the `OnError` method should not handle the passed `Exception` object as an exception, and the `OnCompleted` method is free to call the provider's `IDisposable.Dispose` implementation. The following example shows the `IObserver<T>` implementation of the `TemperatureReporter` class.

C#

```
public virtual void OnCompleted()
{
    Console.WriteLine("Additional temperature data will not be
transmitted.");
}

public virtual void OnError(Exception error)
{
    // Do nothing.
}

public virtual void OnNext(Temperature value)
{
    Console.WriteLine("The temperature is {0}°C at {1:g}",
value.Degrees, value.Date);
    if (first)
    {
        last = value;
        first = false;
    }
    else
    {
        Console.WriteLine("    Change: {0}° in {1:g}",
value.Degrees - last.Degrees,
value.Date.ToUniversalTime() - last.Date.ToUniversalTime());
    }
}
```

```
    }  
}
```

## Example

The following example contains the complete source code for the `TemperatureReporter` class, which provides the `IObserver<T>` implementation for a temperature monitoring application.

C#

```
public class TemperatureReporter : IObserver<Temperature>  
{  
    private IDisposable unsubscriber;  
    private bool first = true;  
    private Temperature last;  
  
    public virtual void Subscribe(IObservable<Temperature> provider)  
    {  
        unsubscriber = provider.Subscribe(this);  
    }  
  
    public virtual void Unsubscribe()  
    {  
        unsubscriber.Dispose();  
    }  
  
    public virtual void OnCompleted()  
    {  
        Console.WriteLine("Additional temperature data will not be  
transmitted.");  
    }  
  
    public virtual void OnError(Exception error)  
    {  
        // Do nothing.  
    }  
  
    public virtual void OnNext(Temperature value)  
    {  
        Console.WriteLine("The temperature is {0}°C at {1:g}", value.Degrees,  
value.Date);  
        if (first)  
        {  
            last = value;  
            first = false;  
        }  
        else  
        {  
            Console.WriteLine("    Change: {0}° in {1:g}", value.Degrees -  
last.Degrees,
```

```
value.Date.ToUniversalTime() - last.Date.ToUniversalTime());
    }
}
}
```

## See also

- [IObserver<T>](#)
- [Observer Design Pattern](#)
- [How to: Implement a Provider](#)
- [Observer Design Pattern Best Practices](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Handling and throwing exceptions in .NET

Article • 09/15/2021

Applications must be able to handle errors that occur during execution in a consistent manner. .NET provides a model for notifying applications of errors in a uniform way: .NET operations indicate failure by throwing exceptions.

## Exceptions

An exception is any error condition or unexpected behavior that is encountered by an executing program. Exceptions can be thrown because of a fault in your code or in code that you call (such as a shared library), unavailable operating system resources, unexpected conditions that the runtime encounters (such as code that can't be verified), and so on. Your application can recover from some of these conditions, but not from others. Although you can recover from most application exceptions, you can't recover from most runtime exceptions.

In .NET, an exception is an object that inherits from the [System.Exception](#) class. An exception is thrown from an area of code where a problem has occurred. The exception is passed up the stack until the application handles it or the program terminates.

## Exceptions vs. traditional error-handling methods

Traditionally, a language's error-handling model relied on either the language's unique way of detecting errors and locating handlers for them, or on the error-handling mechanism provided by the operating system. The way .NET implements exception handling provides the following advantages:

- Exception throwing and handling works the same for .NET programming languages.
- Doesn't require any particular language syntax for handling exceptions, but allows each language to define its own syntax.
- Exceptions can be thrown across process and even machine boundaries.
- Exception-handling code can be added to an application to increase program reliability.

Exceptions offer advantages over other methods of error notification, such as return codes. Failures don't go unnoticed because if an exception is thrown and you don't handle it, the runtime terminates your application. Invalid values don't continue to propagate through the system as a result of code that fails to check for a failure return code.

## Common exceptions

The following table lists some common exceptions with examples of what can cause them.

Exception type	Description	Example
<a href="#">Exception</a>	Base class for all exceptions.	None (use a derived class of this exception).
<a href="#">IndexOutOfRangeException</a>	Thrown by the runtime only when an array is indexed improperly.	Indexing an array outside its valid range: <code>arr[arr.Length+1]</code>
<a href="#">NullReferenceException</a>	Thrown by the runtime only when a null object is referenced.	<code>object o = null;</code> <code>o.ToString();</code>
<a href="#">InvalidOperationException</a>	Thrown by methods when in an invalid state.	Calling <code>Enumerator.MoveNext()</code> after removing an item from the underlying collection.
<a href="#">ArgumentException</a>	Base class for all argument exceptions.	None (use a derived class of this exception).
<a href="#">ArgumentNullException</a>	Thrown by methods that do not allow an argument to be null.	<code>String s = null;</code> <code>"Calculate".IndexOf(s);</code>
<a href="#">ArgumentOutOfRangeException</a>	Thrown by methods that verify that arguments are in a given range.	<code>String s = "string";</code> <code>s.Substring(s.Length+1);</code>

## See also

- [Exception Class and Properties](#)
- [How to: Use the Try-Catch Block to Catch Exceptions](#)
- [How to: Use Specific Exceptions in a Catch Block](#)
- [How to: Explicitly Throw Exceptions](#)

- [How to: Create User-Defined Exceptions](#)
- [Using User-Filtered Exception Handlers](#)
- [How to: Use Finally Blocks](#)
- [Handling COM Interop Exceptions](#)
- [Best Practices for Exceptions](#)
- [What Every Dev needs to Know About Exceptions in the Runtime ↗](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Exception class and properties

Article • 09/15/2021

The [Exception](#) class is the base class from which exceptions inherit. For example, the [InvalidOperationException](#) class hierarchy is as follows:

```
Object
  |
  Exception
    |
    SystemException
      |
      InvalidOperationException
```

The [Exception](#) class has the following properties that help make understanding an exception easier.

[ ] [Expand table](#)

Property	Description
<b>Name</b>	
<a href="#">Data</a>	An <a href="#">IDictionary</a> that holds arbitrary data in key-value pairs.
<a href="#">HelpLink</a>	Can hold a URL (or URN) to a help file that provides extensive information about the cause of an exception.
<a href="#">InnerException</a>	This property can be used to create and preserve a series of exceptions during exception handling. You can use it to create a new exception that contains previously caught exceptions. The original exception can be captured by the second exception in the <a href="#">InnerException</a> property, allowing code that handles the second exception to examine the additional information. For example, suppose you have a method that receives an argument that's improperly formatted. The code tries to read the argument, but an exception is thrown. The method catches the exception and throws a <a href="#">FormatException</a> . To improve the caller's ability to determine the reason an exception is thrown, it is sometimes desirable for a method to catch an exception thrown by a helper routine and then throw an exception more indicative of the error that has occurred. A new and more meaningful exception can be created, where the inner exception reference can be set to the original exception. This more meaningful exception can then be thrown to the caller. Note that with this functionality, you can create a series of linked exceptions that ends with the exception that was thrown first.
<a href="#">Message</a>	Provides details about the cause of an exception.
<a href="#">Source</a>	Gets or sets the name of the application or the object that causes the error.
<a href="#">StackTrace</a>	Contains a stack trace that can be used to determine where an error occurred. The stack trace includes the source file name and program line number if debugging information is available.

Most of the classes that inherit from [Exception](#) do not implement additional members or provide additional functionality; they simply inherit from [Exception](#). Therefore, the most important information for an exception can be found in the hierarchy of exception classes, the exception name, and the information contained in the exception.

We recommend that you throw and catch only objects that derive from [Exception](#), but you can throw any object that derives from the [Object](#) class as an exception. Note that not all languages support throwing and catching objects that do not derive from [Exception](#).

## See also

- [Exceptions](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to use the try/catch block to catch exceptions

Article • 09/15/2021

Place any code statements that might raise or throw an exception in a `try` block, and place statements used to handle the exception or exceptions in one or more `catch` blocks below the `try` block. Each `catch` block includes the exception type and can contain additional statements needed to handle that exception type.

In the following example, a [StreamReader](#) opens a file called *data.txt* and retrieves a line from the file. Since the code might throw any of three exceptions, it's placed in a `try` block. Three `catch` blocks catch the exceptions and handle them by displaying the results to the console.

C#

```
using System;
using System.IO;

public class ProcessFile
{
    public static void Main()
    {
        try
        {
            using (StreamReader sr = File.OpenText("data.txt"))
            {
                Console.WriteLine($"The first line of this file is
{sr.ReadLine()}");
            }
        }
        catch (FileNotFoundException e)
        {
            Console.WriteLine($"The file was not found: '{e}'");
        }
        catch (DirectoryNotFoundException e)
        {
            Console.WriteLine($"The directory was not found: '{e}'");
        }
        catch (IOException e)
        {
            Console.WriteLine($"The file could not be opened: '{e}'");
        }
    }
}
```

The Common Language Runtime (CLR) catches exceptions not handled by `catch` blocks.

If an exception is caught by the CLR, one of the following results may occur depending on your CLR configuration:

- A **Debug** dialog box appears.
- The program stops execution and a dialog box with exception information appears.
- An error prints out to the [standard error output stream](#).

### Note

Most code can throw an exception, and some exceptions, like [OutOfMemoryException](#), can be thrown by the CLR itself at any time. While applications aren't required to deal with these exceptions, be aware of the possibility when writing libraries to be used by others. For suggestions on when to set code in a `try` block, see [Best Practices for Exceptions](#).

## See also

- [Exceptions](#)
- [Handling I/O errors in .NET](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to use specific exceptions in a catch block

Article • 09/15/2021

In general, it's good programming practice to catch a specific type of exception rather than use a basic `catch` statement.

When an exception occurs, it is passed up the stack and each catch block is given the opportunity to handle it. The order of catch statements is important. Put catch blocks targeted to specific exceptions before a general exception catch block or the compiler might issue an error. The proper catch block is determined by matching the type of the exception to the name of the exception specified in the catch block. If there is no specific catch block, the exception is caught by a general catch block, if one exists.

The following code example uses a `try/catch` block to catch an `InvalidCastException`. The sample creates a class called `Employee` with a single property, employee level (`Emlevel`). A method, `PromoteEmployee`, takes an object and increments the employee level. An `InvalidCastException` occurs when a `DateTime` instance is passed to the `PromoteEmployee` method.

C#

```
using System;

public class Employee
{
    //Create employee level property.
    public int Emlevel
    {
        get
        {
            return(emlevel);
        }
        set
        {
            emlevel = value;
        }
    }

    private int emlevel = 0;
}

public class Ex13
{
    public static void PromoteEmployee(Object emp)
    {
```

```
// Cast object to Employee.
var e = (Employee) emp;
// Increment employee level.
e.Emlevel = e.Emlevel + 1;
}

static void Main()
{
    try
    {
        Object o = new Employee();
        DateTime newYears = new DateTime(2001, 1, 1);
        // Promote the new employee.
        PromoteEmployee(o);
        // Promote DateTime; results in InvalidCastException as newYears
        is not an employee instance.
        PromoteEmployee(newYears);
    }
    catch (InvalidCastException e)
    {
        Console.WriteLine("Error passing data to PromoteEmployee method.
" + e.Message);
    }
}
}
```

## See also

- [Exceptions](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

### Open a documentation issue

### Provide product feedback

# How to explicitly throw exceptions

Article • 04/22/2023

You can explicitly throw an exception using the C# [throw](#) or the Visual Basic [Throw](#) statement. You can also throw a caught exception again using the [throw](#) statement. It's good coding practice to add information to an exception that's rethrown to provide more information when debugging.

The following code example uses a [try/catch](#) block to catch a possible [FileNotFoundException](#). Following the [try](#) block is a [catch](#) block that catches the [FileNotFoundException](#) and writes a message to the console if the data file is not found. The next statement is the [throw](#) statement that throws a new [FileNotFoundException](#) and adds text information to the exception.

```
C#  
  
var fs = default(FileStream);  
try  
{  
    // Opens a text file.  
    fs = new FileStream(@"C:\temp\data.txt", FileMode.Open);  
    var sr = new StreamReader(fs);  
  
    // A value is read from the file and output to the console.  
    string? line = sr.ReadLine();  
    Console.WriteLine(line);  
}  
catch (FileNotFoundException e)  
{  
    Console.WriteLine($"[Data File Missing] {e}");  
    throw new FileNotFoundException($"[data.txt not in c:\temp directory]",  
e);  
}  
finally  
{  
    if (fs != null)  
        fs.Close();  
}
```

## See also

- [Exceptions](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to create user-defined exceptions

Article • 08/12/2022

.NET provides a hierarchy of exception classes ultimately derived from the [Exception](#) base class. However, if none of the predefined exceptions meet your needs, you can create your own exception classes by deriving from the [Exception](#) class.

When creating your own exceptions, end the class name of the user-defined exception with the word "Exception", and implement the three common constructors, as shown in the following example. The example defines a new exception class named

`EmployeeListNotFoundException`. The class is derived from the [Exception](#) base class and includes three constructors.

C#

```
using System;

public class EmployeeListNotFoundException : Exception
{
    public EmployeeListNotFoundException()
    {

    }

    public EmployeeListNotFoundException(string message)
        : base(message)
    {

    }

    public EmployeeListNotFoundException(string message, Exception inner)
        : base(message, inner)
    {

    }
}
```

## ⓘ Note

In situations where you're using remoting, you must ensure that the metadata for any user-defined exceptions is available at the server (callee) and to the client (the proxy object or caller). For more information, see [Best practices for exceptions](#).

## See also

- [Exceptions](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to create user-defined exceptions with localized exception messages

Article • 09/15/2021

In this article, you will learn how to create user-defined exceptions that are inherited from the base [Exception](#) class with localized exception messages using satellite assemblies.

## Create custom exceptions

.NET contains many different exceptions that you can use. However, in some cases when none of them meets your needs, you can create your own custom exceptions.

Let's assume you want to create a `StudentNotFoundException` that contains a `StudentName` property. To create a custom exception, follow these steps:

1. Create a serializable class that inherits from [Exception](#). The class name should end in "Exception":

C#

```
[Serializable]
public class StudentNotFoundException : Exception { }
```

2. Add the default constructors:

C#

```
[Serializable]
public class StudentNotFoundException : Exception
{
    public StudentNotFoundException() { }

    public StudentNotFoundException(string message)
        : base(message) { }

    public StudentNotFoundException(string message, Exception inner)
        : base(message, inner) { }
}
```

3. Define any additional properties and constructors:

C#

```
[Serializable]
public class StudentNotFoundException : Exception
{
    public string StudentName { get; }

    public StudentNotFoundException() { }

    public StudentNotFoundException(string message)
        : base(message) { }

    public StudentNotFoundException(string message, Exception inner)
        : base(message, inner) { }

    public StudentNotFoundException(string message, string studentName)
        : this(message)
    {
        StudentName = studentName;
    }
}
```

## Create localized exception messages

You have created a custom exception, and you can throw it anywhere with code like the following:

C#

```
throw new StudentNotFoundException("The student cannot be found.", "John");
```

The problem with the previous line is that "The student cannot be found." is just a constant string. In a localized application, you want to have different messages depending on user culture. [Satellite assemblies](#) are a good way to do that. A satellite assembly is a .dll that contains resources for a specific language. When you ask for a specific resources at run time, the CLR finds that resource depending on user culture. If no satellite assembly is found for that culture, the resources of the default culture are used.

To create the localized exception messages:

1. Create a new folder named *Resources* to hold the resource files.
2. Add a new resource file to it. To do that in Visual Studio, right-click the folder in **Solution Explorer**, and select **Add > New Item > Resources File**. Name the file *ExceptionMessages.resx*. This is the default resources file.

3. Add a name/value pair for your exception message, like the following image shows:

	Name	Value
▶	StudentNotFound	The student cannot be found.
*		

4. Add a new resource file for French. Name it *ExceptionMessages.fr-FR.resx*.
5. Add a name/value pair for the exception message again, but with a French value:

	Name	Value
▶	StudentNotFound	L'étudiant est introuvable.
*		

6. After you build the project, the build output folder should contain the *fr-FR* folder with a *.dll* file, which is the satellite assembly.
7. You throw the exception with code like the following:

```
C#  
  
var resourceManager = new  
ResourceManager("FULLY_QUALIFIED_NAME_OF_RESOURCE_FILE",  
Assembly.GetExecutingAssembly());  
throw new  
StudentNotFoundException(resourceManager.GetString("StudentNotFound"),  
"John");
```

#### (!) Note

If the project name is `TestProject` and the resource file `ExceptionMessages.resx` resides in the `Resources` folder of the project, the fully qualified name of the resource file is `TestProject.Resources.ExceptionMessages`.

## See also

- [How to create user-defined exceptions](#)
- [Create satellite assemblies](#)
- [base \(C# Reference\)](#)
- [this \(C# Reference\)](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to use finally blocks

Article • 09/15/2021

When an exception occurs, execution stops and control is given to the appropriate exception handler. This often means that lines of code you expect to be executed are bypassed. Some resource cleanup, such as closing a file, needs to be done even if an exception is thrown. To do this, you can use a `finally` block. A `finally` block always executes, regardless of whether an exception is thrown.

The following code example uses a `try/catch` block to catch an [ArgumentOutOfRangeException](#). The `Main` method creates two arrays and attempts to copy one to the other. The action generates an [ArgumentOutOfRangeException](#) and the error is written to the console. The `finally` block executes regardless of the outcome of the copy action.

C#

```
using System;

class ArgumentOutOfRangeExceptionExample
{
    public static void Main()
    {
        int[] array1 = {0, 0};
        int[] array2 = {0, 0};

        try
        {
            Array.Copy(array1, array2, -1);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine("Error: {0}", e);
            throw;
        }
        finally
        {
            Console.WriteLine("This statement is always executed.");
        }
    }
}
```

## See also

- [Exceptions](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Use user-filtered exception handlers

Article • 09/15/2021

User-filtered exception handlers catch and handle exceptions based on requirements you define for the exception. These handlers use the `catch` statement with the `when` keyword (`catch` and `When` in Visual Basic).

This technique is useful when a particular exception object corresponds to multiple errors. In this case, the object typically has a property that contains the specific error code associated with the error. You can use the error code property in the expression to select only the particular error you want to handle in that `catch` clause.

The following example illustrates the `catch/when` statement.

C#

```
try
{
    //Try statements.
}
catch (Exception ex) when (ex.Message.Contains("404"))
{
    //Catch statements.
}
```

The expression of the user-filtered clause is not restricted in any way. If an exception occurs during execution of the user-filtered expression, that exception is discarded and the filter expression is considered to have evaluated to false. In this case, the common language runtime continues the search for a handler for the current exception.

## Combine the specific exception and the user-filtered clauses

A `catch` statement can contain both the specific exception and the user-filtered clauses. The runtime tests the specific exception first. If the specific exception succeeds, the runtime executes the user filter. The generic filter can contain a reference to the variable declared in the class filter. Note that the order of the two filter clauses cannot be reversed.

The following example shows a specific exception in the `catch` statement as well as the user-filtered clause using the `when` keyword.

C#

```
try
{
    //Try statements.
}
catch (System.Net.Http.HttpRequestException ex) when
(ex.Message.Contains("404"))
{
    //Catch statements.
}
```

## See also

- [Exceptions](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Handling COM Interop Exceptions

Article • 09/15/2021

Managed and unmanaged code can work together to handle exceptions. If a method throws an exception in managed code, the common language runtime can pass an HRESULT to a COM object. If a method fails in unmanaged code by returning a failure HRESULT, the runtime throws an exception that can be caught by managed code.

The runtime automatically maps the HRESULT from COM interop to more specific exceptions. For example, E\_ACCESSDENIED becomes [UnauthorizedAccessException](#), E\_OUTOFMEMORY becomes [OutOfMemoryException](#), and so on.

If the HRESULT is a custom result or if it is unknown to the runtime, the runtime passes a generic [COMException](#) to the client. The **ErrorCode** property of the [COMException](#) contains the HRESULT value.

## Working with IErrorInfo

When an error is passed from COM to managed code, the runtime populates the exception object with error information. COM objects that support [IErrorInfo](#) and return HRESULTS provide this information to managed code exceptions. For example, the runtime maps the **Description** from the COM error to the exception's [Message](#) property. If the HRESULT provides no additional error information, the runtime fills many of the exception's properties with default values.

If a method fails in unmanaged code, an exception can be passed to a managed code segment. The topic [HRESULTS and Exceptions](#) contains a table showing how HRESULTS map to runtime exception objects.

## See also

- [Exceptions](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

# Best practices for exceptions

Article • 02/16/2023

A well-designed app handles exceptions and errors to prevent app crashes. This article describes best practices for handling and creating exceptions.

## Use try/catch/finally blocks to recover from errors or release resources

Use `try` / `catch` blocks around code that can potentially generate an exception, and your code can recover from that exception. In `catch` blocks, always order exceptions from the most derived to the least derived. All exceptions derive from the [Exception](#) class. More derived exceptions aren't handled by a catch clause that's preceded by a catch clause for a base exception class. When your code can't recover from an exception, don't catch that exception. Enable methods further up the call stack to recover if possible.

Clean up resources that are allocated with either `using` statements or `finally` blocks. Prefer `using` statements to automatically clean up resources when exceptions are thrown. Use `finally` blocks to clean up resources that don't implement [IDisposable](#). Code in a `finally` clause is almost always executed even when exceptions are thrown.

## Handle common conditions without throwing exceptions

For conditions that are likely to occur but might trigger an exception, consider handling them in a way that will avoid the exception. For example, if you try to close a connection that's already closed, you'll get an `InvalidOperationException`. You can avoid that by using an `if` statement to check the connection state before trying to close it.

C#

```
if (conn.State != ConnectionState.Closed)
{
    conn.Close();
}
```

If you don't check the connection state before closing, you can catch the `InvalidOperationException` exception.

C#

```
try
{
    conn.Close();
}
catch (InvalidOperationException ex)
{
    Console.WriteLine(ex.GetType().FullName);
    Console.WriteLine(ex.Message);
}
```

The method to choose depends on how often you expect the event to occur.

- Use exception handling if the event doesn't occur often, that is, if the event is truly exceptional and indicates an error, such as an unexpected end-of-file. When you use exception handling, less code is executed in normal conditions.
- Check for error conditions in code if the event happens routinely and could be considered part of normal execution. When you check for common error conditions, less code is executed because you avoid exceptions.

## Design classes so that exceptions can be avoided

A class can provide methods or properties that enable you to avoid making a call that would trigger an exception. For example, a [FileStream](#) class provides methods that help determine whether the end of the file has been reached. These methods can be used to avoid the exception that's thrown if you read past the end of the file. The following example shows how to read to the end of a file without triggering an exception:

C#

```
class FileRead
{
    public void ReadAll(FileStream fileToRead)
    {
        // This if statement is optional
        // as it is very unlikely that
        // the stream would ever be null.
        if (fileToRead == null)
        {
            throw new ArgumentNullException();
        }

        int b;
```

```
// Set the stream position to the beginning of the file.  
fileToRead.Seek(0, SeekOrigin.Begin);  
  
// Read each byte to the end of the file.  
for (int i = 0; i < fileToRead.Length; i++)  
{  
    b = fileToRead.ReadByte();  
    Console.WriteLine(b.ToString());  
    // Or do something else with the byte.  
}  
}  
}
```

Another way to avoid exceptions is to return null (or default) for most common error cases instead of throwing an exception. A common error case can be considered a normal flow of control. By returning null (or default) in these cases, you minimize the performance impact to an app.

For value types, whether to use `Nullable<T>` or default as your error indicator is something to consider for your app. By using `Nullable<Guid>`, `default` becomes `null` instead of `Guid.Empty`. Sometimes, adding `Nullable<T>` can make it clearer when a value is present or absent. Other times, adding `Nullable<T>` can create extra cases to check that aren't necessary and only serve to create potential sources of errors.

## Throw exceptions instead of returning an error code

Exceptions ensure that failures don't go unnoticed because the calling code didn't check a return code.

## Use the predefined .NET exception types

Introduce a new exception class only when a predefined one doesn't apply. For example:

- If a property set or method call isn't appropriate given the object's current state, throw an `InvalidOperationException` exception.
- If invalid parameters are passed, throw an `ArgumentException` exception or one of the predefined classes that derive from `ArgumentException`.

## End exception class names with the word `Exception`

When a custom exception is necessary, name it appropriately and derive it from the [Exception](#) class. For example:

```
C#
```

```
public class MyFileNotFoundException : Exception
{
}
```

## Include three constructors in custom exception classes

Use at least the three common constructors when creating your own exception classes: the parameterless constructor, a constructor that takes a string message, and a constructor that takes a string message and an inner exception.

- [Exception\(\)](#), which uses default values.
- [Exception\(String\)](#), which accepts a string message.
- [Exception\(String, Exception\)](#), which accepts a string message and an inner exception.

For an example, see [How to: Create User-Defined Exceptions](#).

## Ensure that exception data is available when code executes remotely

When you create user-defined exceptions, ensure that the metadata for the exceptions is available to code that's executing remotely.

For example, on .NET implementations that support app domains, exceptions might occur across app domains. Suppose app domain A creates app domain B, which executes code that throws an exception. For app domain A to properly catch and handle the exception, it must be able to find the assembly that contains the exception thrown by app domain B. If app domain B throws an exception that is contained in an assembly under its application base, but not under app domain A's application base, app domain A won't be able to find the exception, and the common language runtime will throw a [FileNotFoundException](#) exception. To avoid this situation, you can deploy the assembly that contains the exception information in either of two ways:

- Put the assembly into a common application base shared by both app domains.

- If the domains don't share a common application base, sign the assembly that contains the exception information with a strong name and deploy the assembly into the global assembly cache.

## Use grammatically correct error messages

Write clear sentences and include ending punctuation. Each sentence in the string assigned to the [Exception.Message](#) property should end in a period. For example, "The log table has overflowed." would be an appropriate message string.

## Include a localized string message in every exception

The error message the user sees is derived from the [Exception.Message](#) property of the exception that was thrown, and not from the name of the exception class. Typically, you assign a value to the [Exception.Message](#) property by passing the message string to the `message` argument of an [Exception constructor](#).

For localized applications, you should provide a localized message string for every exception that your application can throw. You use resource files to provide localized error messages. For information on localizing applications and retrieving localized strings, see the following articles:

- [How to: Create user-defined exceptions with localized exception messages](#)
- [Resources in .NET apps](#)
- [System.Resources.ResourceManager](#)

## In custom exceptions, provide additional properties as needed

Provide additional properties for an exception (in addition to the custom message string) only when there's a programmatic scenario where the additional information is useful. For example, the [FileNotFoundException](#) provides the [FileName](#) property.

## Place throw statements so that the stack trace will be helpful

The stack trace begins at the statement where the exception is thrown and ends at the `catch` statement that catches the exception.

## Use exception builder methods

It's common for a class to throw the same exception from different places in its implementation. To avoid excessive code, use helper methods that create the exception and return it. For example:

C#

```
class FileReader
{
    private string fileName;

    public FileReader(string path)
    {
        fileName = path;
    }

    public byte[] Read(int bytes)
    {
        byte[] results = FileUtils.ReadFromFile(fileName, bytes);
        if (results == null)
        {
            throw NewFileIOException();
        }
        return results;
    }

    FileReaderException NewFileIOException()
    {
        string description = "My NewFileIOException Description";

        return new FileReaderException(description);
    }
}
```

In some cases, it's more appropriate to use the exception's constructor to build the exception. An example is a global exception class such as [ArgumentException](#).

## Restore state when methods don't complete due to exceptions

Callers should be able to assume that there are no side effects when an exception is thrown from a method. For example, if you have code that transfers money by

withdrawing from one account and depositing in another account, and an exception is thrown while executing the deposit, you don't want the withdrawal to remain in effect.

C#

```
public void TransferFunds(Account from, Account to, decimal amount)
{
    from.Withdrawal(amount);
    // If the deposit fails, the withdrawal shouldn't remain in effect.
    to.Deposit(amount);
}
```

The preceding method doesn't directly throw any exceptions. However, you must write the method so that the withdrawal is reversed if the deposit operation fails.

One way to handle this situation is to catch any exceptions thrown by the deposit transaction and roll back the withdrawal.

C#

```
private static void TransferFunds(Account from, Account to, decimal amount)
{
    string withdrawalTrxID = from.Withdrawal(amount);
    try
    {
        to.Deposit(amount);
    }
    catch
    {
        from.RollbackTransaction(withdrawalTrxID);
        throw;
    }
}
```

This example illustrates the use of `throw` to rethrow the original exception, making it easier for callers to see the real cause of the problem without having to examine the `InnerException` property. An alternative is to throw a new exception and include the original exception as the inner exception.

C#

```
catch (Exception ex)
{
    from.RollbackTransaction(withdrawalTrxID);
    throw new TransferFundsException("Withdrawal failed.", innerException:
ex)
{
    From = from,
```

```
        To = to,
        Amount = amount
    };
}
```

## Capture exceptions to rethrow later

To capture an exception and preserve its callstack to be able to rethrow it later, use the [System.Runtime.ExceptionServices.ExceptionDispatchInfo](#) class. This class provides the following methods and properties (among others):

- Use [ExceptionDispatchInfo.Capture\(Exception\)](#) to capture an exception and call stack.
- Use [ExceptionDispatchInfo.Throw\(\)](#) to restore the state that was saved when the exception was captured and rethrow the captured exception.
- Use the [ExceptionDispatchInfo.SourceException](#) property to inspect the captured exception.

The following example shows how the [ExceptionDispatchInfo](#) class can be used, and what the output might look like.

C#

```
ExceptionDispatchInfo? edi = null;
try
{
    var txt = File.ReadAllText(@"C:\temp\file.txt");
}
catch (FileNotFoundException e)
{
    edi = ExceptionDispatchInfo.Capture(e);
}

// ...

Console.WriteLine("I was here.");

if (edi is not null)
    edi.Throw();
```

If the file in the example code doesn't exist, the following output is produced:

Output

```
I was here.
Unhandled exception. System.IO.FileNotFoundException: Could not find file
'C:\temp\file.txt'.
```

```
File name: 'C:\temp\file.txt'
  at Microsoft.Win32.SafeHandles.SafeFileHandle.CreateFile(String fullPath,
 FileMode mode, FileAccess access, FileShare share, FileOptions options)
  at Microsoft.Win32.SafeHandles.SafeFileHandle.Open(String fullPath,
 FileMode mode, FileAccess access, FileShare share, FileOptions options,
 Int64 preallocationSize, Nullable`1 unixCreateMode)
  at System.IO.Strategies.OSFileStreamStrategy..ctor(String path, FileMode
 mode, FileAccess access, FileShare share, FileOptions options, Int64
 preallocationSize, Nullable`1 unixCreateMode)
  at System.IO.Strategies.FileStreamHelpers.ChooseStrategyCore(String path,
 FileMode mode, FileAccess access, FileShare share, FileOptions options,
 Int64 preallocationSize, Nullable`1 unixCreateMode)
  at System.IO.StreamReader.ValidateArgsAndOpenPath(String path, Encoding
 encoding, Int32 bufferSize)
  at System.IO.File.ReadAllText(String path, Encoding encoding)
  at Example.ProcessFile.Main() in C:\repos\ConsoleApp1\Program.cs:line 12
--- End of stack trace from previous location ---
  at Example.ProcessFile.Main() in C:\repos\ConsoleApp1\Program.cs:line 24
```

## See also

- [Exceptions](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### [.NET feedback](#)

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.AccessViolationException class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

An access violation occurs in unmanaged or unsafe code when the code attempts to read or write to memory that has not been allocated, or to which it does not have access. This usually occurs because a pointer has a bad value. Not all reads or writes through bad pointers lead to access violations, so an access violation usually indicates that several reads or writes have occurred through bad pointers, and that memory might be corrupted. Thus, access violations almost always indicate serious programming errors. An [AccessViolationException](#) clearly identifies these serious errors.

In programs consisting entirely of verifiable managed code, all references are either valid or null, and access violations are impossible. Any operation that attempts to reference a null reference in verifiable code throws a [NullReferenceException](#) exception. An [AccessViolationException](#) occurs only when verifiable managed code interacts with unmanaged code or with unsafe managed code.

## Troubleshoot AccessViolationException exceptions

An [AccessViolationException](#) exception can occur only in unsafe managed code or when verifiable managed code interacts with unmanaged code:

- An access violation that occurs in unsafe managed code can be expressed as either a [NullReferenceException](#) exception or an [AccessViolationException](#) exception, depending on the platform.
- An access violation in unmanaged code that bubbles up to managed code is always wrapped in an [AccessViolationException](#) exception.

In either case, you can identify and correct the cause of the [AccessViolationException](#) exception as follows:

- Make sure that the memory that you're attempting to access has been allocated. An [AccessViolationException](#) exception is always thrown by an attempt to access protected memory—that is, to access memory that's not allocated or that's not owned by a process.

Automatic memory management is one of the services that the .NET runtime provides. If managed code provides the same functionality as your unmanaged code, you may wish to move to managed code to take advantage of this functionality. For more information, see [Automatic Memory Management](#).

- Make sure that the memory that you're attempting to access has not been corrupted. If several read or write operations have occurred through bad pointers, memory might be corrupted. This typically occurs when reading or writing to addresses outside of a predefined buffer.

## AccessViolationException and try/catch blocks

[AccessViolationException](#) exceptions thrown by the .NET runtime aren't handled by the `catch` statement in a structured exception handler if the exception occurs outside of the memory reserved by the runtime. To handle such an [AccessViolationException](#) exception, apply the [HandleProcessCorruptedStateExceptionsAttribute](#) attribute to the method in which the exception is thrown. This change does not affect [AccessViolationException](#) exceptions thrown by user code, which can continue to be caught by a `catch` statement.

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Exception class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Exception](#) class is the base class for all exceptions. When an error occurs, either the system or the currently executing application reports it by throwing an exception that contains information about the error. After an exception is thrown, it is handled by the application or by the default exception handler.

## Errors and exceptions

Run-time errors can occur for a variety of reasons. However, not all errors should be handled as exceptions in your code. Here are some categories of errors that can occur at run time and the appropriate ways to respond to them.

- **Usage errors.** A usage error represents an error in program logic that can result in an exception. However, the error should be addressed not through exception handling but by modifying the faulty code. For example, the override of the [Object.Equals\(Object\)](#) method in the following example assumes that the `obj` argument must always be non-null.

C#

```
using System;

public class Person1
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public override int GetHashCode()
    {
        return this.Name.GetHashCode();
    }

    public override bool Equals(object obj)
    {
        // This implementation contains an error in program logic:
        // It assumes that the obj argument is not null.
    }
}
```

```

        Person1 p = (Person1) obj;
        return this.Name.Equals(p.Name);
    }
}

public class UsageErrorsEx1
{
    public static void Main()
    {
        Person1 p1 = new Person1();
        p1.Name = "John";
        Person1 p2 = null;

        // The following throws a NullReferenceException.
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}

```

The [NullReferenceException](#) exception that results when `obj` is `null` can be eliminated by modifying the source code to explicitly test for null before calling the [Object.Equals](#) override and then re-compiling. The following example contains the corrected source code that handles a `null` argument.

C#

```

using System;

public class Person2
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public override int GetHashCode()
    {
        return this.Name.GetHashCode();
    }

    public override bool Equals(object obj)
    {
        // This implementation handles a null obj argument.
        Person2 p = obj as Person2;
        if (p == null)
            return false;
        else
            return this.Name.Equals(p.Name);
    }
}

```

```

public class UsageErrorsEx2
{
    public static void Main()
    {
        Person2 p1 = new Person2();
        p1.Name = "John";
        Person2 p2 = null;

        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}

// The example displays the following output:
//      p1 = p2: False

```

Instead of using exception handling for usage errors, you can use the [Debug.Assert](#) method to identify usage errors in debug builds, and the [Trace.Assert](#) method to identify usage errors in both debug and release builds. For more information, see [Assertions in Managed Code](#).

- **Program errors.** A program error is a run-time error that cannot necessarily be avoided by writing bug-free code.

In some cases, a program error may reflect an expected or routine error condition. In this case, you may want to avoid using exception handling to deal with the program error and instead retry the operation. For example, if the user is expected to input a date in a particular format, you can parse the date string by calling the [DateTime.TryParseExact](#) method, which returns a [Boolean](#) value that indicates whether the parse operation succeeded, instead of using the [DateTime.ParseExact](#) method, which throws a [FormatException](#) exception if the date string cannot be converted to a [DateTime](#) value. Similarly, if a user tries to open a file that does not exist, you can first call the [File.Exists](#) method to check whether the file exists and, if it does not, prompt the user whether they want to create it.

In other cases, a program error reflects an unexpected error condition that can be handled in your code. For example, even if you've checked to ensure that a file exists, it may be deleted before you can open it, or it may be corrupted. In that case, trying to open the file by instantiating a [StreamReader](#) object or calling the [Open](#) method may throw a [FileNotFoundException](#) exception. In these cases, you should use exception handling to recover from the error.

- **System failures.** A system failure is a run-time error that cannot be handled programmatically in a meaningful way. For example, any method can throw an [OutOfMemoryException](#) exception if the common language runtime is unable to allocate additional memory. Ordinarily, system failures are not handled by using

exception handling. Instead, you may be able to use an event such as [AppDomain.UnhandledException](#) and call the [Environment.FailFast](#) method to log exception information and notify the user of the failure before the application terminates.

## Try/catch blocks

The common language runtime provides an exception handling model that is based on the representation of exceptions as objects, and the separation of program code and exception handling code into `try` blocks and `catch` blocks. There can be one or more `catch` blocks, each designed to handle a particular type of exception, or one block designed to catch a more specific exception than another block.

If an application handles exceptions that occur during the execution of a block of application code, the code must be placed within a `try` statement and is called a `try` block. Application code that handles exceptions thrown by a `try` block is placed within a `catch` statement and is called a `catch` block. Zero or more `catch` blocks are associated with a `try` block, and each `catch` block includes a type filter that determines the types of exceptions it handles.

When an exception occurs in a `try` block, the system searches the associated `catch` blocks in the order they appear in application code, until it locates a `catch` block that handles the exception. A `catch` block handles an exception of type `T` if the type filter of the catch block specifies `T` or any type that `T` derives from. The system stops searching after it finds the first `catch` block that handles the exception. For this reason, in application code, a `catch` block that handles a type must be specified before a `catch` block that handles its base types, as demonstrated in the example that follows this section. A catch block that handles `System.Exception` is specified last.

If none of the `catch` blocks associated with the current `try` block handle the exception, and the current `try` block is nested within other `try` blocks in the current call, the `catch` blocks associated with the next enclosing `try` block are searched. If no `catch` block for the exception is found, the system searches previous nesting levels in the current call. If no `catch` block for the exception is found in the current call, the exception is passed up the call stack, and the previous stack frame is searched for a `catch` block that handles the exception. The search of the call stack continues until the exception is handled or until no more frames exist on the call stack. If the top of the call stack is reached without finding a `catch` block that handles the exception, the default exception handler handles it and the application terminates.

## F# try..with expression

F# does not use `catch` blocks. Instead, a raised exception is pattern matched using a single `with` block. As this is an expression, rather than a statement, all paths must return the same type. To learn more, see [The try...with Expression](#).

## Exception type features

Exception types support the following features:

- Human-readable text that describes the error. When an exception occurs, the runtime makes a text message available to inform the user of the nature of the error and to suggest action to resolve the problem. This text message is held in the [Message](#) property of the exception object. During the creation of the exception object, you can pass a text string to the constructor to describe the details of that particular exception. If no error message argument is supplied to the constructor, the default error message is used. For more information, see the [Message](#) property.
- The state of the call stack when the exception was thrown. The [StackTrace](#) property carries a stack trace that can be used to determine where the error occurs in the code. The stack trace lists all the called methods and the line numbers in the source file where the calls are made.

## Exception class properties

The [Exception](#) class includes a number of properties that help identify the code location, the type, the help file, and the reason for the exception: [StackTrace](#), [InnerException](#), [Message](#), [HelpLink](#), [HResult](#), [Source](#), [TargetSite](#), and [Data](#).

When a causal relationship exists between two or more exceptions, the [InnerException](#) property maintains this information. The outer exception is thrown in response to this inner exception. The code that handles the outer exception can use the information from the earlier inner exception to handle the error more appropriately. Supplementary information about the exception can be stored as a collection of key/value pairs in the [Data](#) property.

The error message string that is passed to the constructor during the creation of the exception object should be localized and can be supplied from a resource file by using the [ResourceManager](#) class. For more information about localized resources, see the [Creating Satellite Assemblies](#) and [Packaging and Deploying Resources](#) topics.

To provide the user with extensive information about why the exception occurred, the [HelpLink](#) property can hold a URL (or URN) to a help file.

The [Exception](#) class uses the HRESULT `COR_E_EXCEPTION`, which has the value 0x80131500.

For a list of initial property values for an instance of the [Exception](#) class, see the [Exception](#) constructors.

## Performance considerations

Throwing or handling an exception consumes a significant amount of system resources and execution time. Throw exceptions only to handle truly extraordinary conditions, not to handle predictable events or flow control. For example, in some cases, such as when you're developing a class library, it's reasonable to throw an exception if a method argument is invalid, because you expect your method to be called with valid parameters. An invalid method argument, if it is not the result of a usage error, means that something extraordinary has occurred. Conversely, do not throw an exception if user input is invalid, because you can expect users to occasionally enter invalid data. Instead, provide a retry mechanism so users can enter valid input. Nor should you use exceptions to handle usage errors. Instead, use [assertions](#) to identify and correct usage errors.

In addition, do not throw an exception when a return code is sufficient; do not convert a return code to an exception; and do not routinely catch an exception, ignore it, and then continue processing.

## Re-throw an exception

In many cases, an exception handler simply wants to pass the exception on to the caller. This most often occurs in:

- A class library that in turn wraps calls to methods in the .NET class library or other class libraries.
- An application or library that encounters a fatal exception. The exception handler can log the exception and then re-throw the exception.

The recommended way to re-throw an exception is to simply use the [throw](#) statement in C#, the [reraise](#) function in F#, and the [Throw](#) statement in Visual Basic without including an expression. This ensures that all call stack information is preserved when the exception is propagated to the caller. The following example illustrates this. A string

extension method, `FindOccurrences`, wraps one or more calls to `String.IndexOf(String, Int32)` without validating its arguments beforehand.

C#

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

public static class Library1
{
    public static int[] FindOccurrences(this String s, String f)
    {
        var indexes = new List<int>();
        int currentIndex = 0;
        try
        {
            while (currentIndex >= 0 && currentIndex < s.Length)
            {
                currentIndex = s.IndexOf(f, currentIndex);
                if (currentIndex >= 0)
                {
                    indexes.Add(currentIndex);
                    currentIndex++;
                }
            }
        }
        catch (ArgumentNullException)
        {
            // Perform some action here, such as logging this exception.

            throw;
        }
        return indexes.ToArray();
    }
}
```

A caller then calls `FindOccurrences` twice. In the second call to `FindOccurrences`, the caller passes a `null` as the search string, which causes the `String.IndexOf(String, Int32)` method to throw an `ArgumentNullException` exception. This exception is handled by the `FindOccurrences` method and passed back to the caller. Because the `throw` statement is used with no expression, the output from the example shows that the call stack is preserved.

C#

```
public class RethrowEx1
{
    public static void Main()
    {
        String s = "It was a cold day when...";
```

```

        int[] indexes = s.FindOccurrences("a");
        ShowOccurrences(s, "a", indexes);
        Console.WriteLine();

        String toFind = null;
        try
        {
            indexes = s.FindOccurrences(toFind);
            ShowOccurrences(s, toFind, indexes);
        }
        catch (ArgumentNullException e)
        {
            Console.WriteLine("An exception ({0}) occurred.",
                e.GetType().Name);
            Console.WriteLine("Message:\n  {0}\n", e.Message);
            Console.WriteLine("Stack Trace:\n  {0}\n", e.StackTrace);
        }
    }

    private static void ShowOccurrences(String s, String toFind, int[]
indexes)
    {
        Console.Write('{0}' occurs at the following character positions: ",
            toFind);
        for (int ctr = 0; ctr < indexes.Length; ctr++)
            Console.Write("{0}{1}", indexes[ctr],
                ctr == indexes.Length - 1 ? "" : ", ");

        Console.WriteLine();
    }
}

// The example displays the following output:
//      'a' occurs at the following character positions: 4, 7, 15
//
//      An exception (ArgumentNullException) occurred.
//      Message:
//          Value cannot be null.
//      Parameter name: value
//
//      Stack Trace:
//          at System.String.IndexOf(String value, Int32 startIndex, Int32
count, Stri
//      ngComparison comparisonType)
//          at Library.FindOccurrences(String s, String f)
//          at Example.Main()

```

In contrast, if the exception is re-thrown by using this statement:

C#

```
throw e;
```

...then the full call stack is not preserved, and the example would generate the following output:

#### Output

```
'a' occurs at the following character positions: 4, 7, 15

An exception (ArgumentNullException) occurred.
Message:
  Value cannot be null.
Parameter name: value

Stack Trace:
  at Library.FindOccurrences(String s, String f)
  at Example.Main()
```

A slightly more cumbersome alternative is to throw a new exception, and to preserve the original exception's call stack information in an inner exception. The caller can then use the new exception's [InnerException](#) property to retrieve stack frame and other information about the original exception. In this case, the throw statement is:

#### C#

```
throw new ArgumentNullException("You must supply a search string.", e);
```

The user code that handles the exception has to know that the [InnerException](#) property contains information about the original exception, as the following exception handler illustrates.

#### C#

```
try
{
    indexes = s.FindOccurrences(toFind);
    ShowOccurrences(s, toFind, indexes);
}
catch (ArgumentNullException e)
{
    Console.WriteLine("An exception ({0}) occurred.",
                      e.GetType().Name);
    Console.WriteLine("  Message:\n{0}", e.Message);
    Console.WriteLine("  Stack Trace:\n  {0}", e.StackTrace);
    Exception ie = e.InnerException;
    if (ie != null)
    {
        Console.WriteLine("  The Inner Exception:");
        Console.WriteLine("    Exception Name: {0}", ie.GetType().Name);
        Console.WriteLine("    Message: {0}\n", ie.Message);
        Console.WriteLine("    Stack Trace:\n    {0}\n", ie.StackTrace);
    }
}
```

```

    }

// The example displays the following output:
//      'a' occurs at the following character positions: 4, 7, 15
//      An exception (ArgumentNullException) occurred.
//          Message: You must supply a search string.
//
//      Stack Trace:
//          at Library.FindOccurrences(String s, String f)
//          at Example.Main()
//
//      The Inner Exception:
//          Exception Name: ArgumentNullException
//          Message: Value cannot be null.
//          Parameter name: value
//
//      Stack Trace:
//          at System.String.IndexOf(String value, Int32 startIndex, Int32
//count, Stri
//ngComparison comparisonType)
//          at Library.FindOccurrences(String s, String f)

```

## Choose standard exceptions

When you have to throw an exception, you can often use an existing exception type in .NET instead of implementing a custom exception. You should use a standard exception type under these two conditions:

- You're throwing an exception that is caused by a usage error (that is, by an error in program logic made by the developer who is calling your method). Typically, you would throw an exception such as [ArgumentException](#), [ArgumentNullException](#), [InvalidOperationException](#), or [NotSupportedException](#). The string you supply to the exception object's constructor when instantiating the exception object should describe the error so that the developer can fix it. For more information, see the [Message](#) property.
- You're handling an error that can be communicated to the caller with an existing .NET exception. You should throw the most derived exception possible. For example, if a method requires an argument to be a valid member of an enumeration type, you should throw an [InvalidEnumArgumentException](#) (the most derived class) rather than an [ArgumentException](#).

The following table lists common exception types and the conditions under which you would throw them.

Exception	Condition
ArgumentException	A non-null argument that is passed to a method is invalid.
ArgumentNullException	An argument that is passed to a method is <code>null</code> .
ArgumentOutOfRangeException	An argument is outside the range of valid values.
DirectoryNotFoundException	Part of a directory path is not valid.
DivideByZeroException	The denominator in an integer or <code>Decimal</code> division operation is zero.
DriveNotFoundException	A drive is unavailable or does not exist.
FileNotFoundException	A file does not exist.
FormatException	A value is not in an appropriate format to be converted from a string by a conversion method such as <code>Parse</code> .
IndexOutOfRangeException	An index is outside the bounds of an array or collection.
InvalidOperationException	A method call is invalid in an object's current state.
KeyNotFoundException	The specified key for accessing a member in a collection cannot be found.
NotImplementedException	A method or operation is not implemented.
NotSupportedException	A method or operation is not supported.
ObjectDisposedException	An operation is performed on an object that has been disposed.
OverflowException	An arithmetic, casting, or conversion operation results in an overflow.
PathTooLongException	A path or file name exceeds the maximum system-defined length.
PlatformNotSupportedException	The operation is not supported on the current platform.
RankException	An array with the wrong number of dimensions is passed to a method.
TimeoutException	The time interval allotted to an operation has expired.
UriFormatException	An invalid Uniform Resource Identifier (URI) is used.

# Implement custom exceptions

In the following cases, using an existing .NET exception to handle an error condition is not adequate:

- When the exception reflects a unique program error that cannot be mapped to an existing .NET exception.
- When the exception requires handling that is different from the handling that is appropriate for an existing .NET exception, or the exception must be disambiguated from a similar exception. For example, if you throw an [ArgumentOutOfRangeException](#) exception when parsing the numeric representation of a string that is out of range of the target integral type, you would not want to use the same exception for an error that results from the caller not supplying the appropriate constrained values when calling the method.

The [Exception](#) class is the base class of all exceptions in .NET. Many derived classes rely on the inherited behavior of the members of the [Exception](#) class; they do not override the members of [Exception](#), nor do they define any unique members.

To define your own exception class:

1. Define a class that inherits from [Exception](#). If necessary, define any unique members needed by your class to provide additional information about the exception. For example, the [ArgumentException](#) class includes a [ParamName](#) property that specifies the name of the parameter whose argument caused the exception, and the [RegexMatchTimeoutException](#) property includes a [MatchTimeout](#) property that indicates the time-out interval.
2. If necessary, override any inherited members whose functionality you want to change or modify. Note that most existing derived classes of [Exception](#) do not override the behavior of inherited members.
3. Determine whether your custom exception object is serializable. Serialization enables you to save information about the exception and permits exception information to be shared by a server and a client proxy in a remoting context. To make the exception object serializable, mark it with the [SerializableAttribute](#) attribute.
4. Define the constructors of your exception class. Typically, exception classes have one or more of the following constructors:
  - [Exception\(\)](#), which uses default values to initialize the properties of a new exception object.

- `Exception(String)`, which initializes a new exception object with a specified error message.
- `Exception(String, Exception)`, which initializes a new exception object with a specified error message and inner exception.
- `Exception(SerializationInfo, StreamingContext)`, which is a `protected` constructor that initializes a new exception object from serialized data. You should implement this constructor if you've chosen to make your exception object serializable.

The following example illustrates the use of a custom exception class. It defines a `NotPrimeException` exception that is thrown when a client tries to retrieve a sequence of prime numbers by specifying a starting number that is not prime. The exception defines a new property, `NonPrime`, that returns the non-prime number that caused the exception. Besides implementing a protected parameterless constructor and a constructor with `SerializationInfo` and `StreamingContext` parameters for serialization, the `NotPrimeException` class defines three additional constructors to support the `NonPrime` property. Each constructor calls a base class constructor in addition to preserving the value of the non-prime number. The `NotPrimeException` class is also marked with the `SerializableAttribute` attribute.

C#

```
using System;
using System.Runtime.Serialization;

[Serializable()]
public class NotPrimeException : Exception
{
    private int notAPrime;

    protected NotPrimeException()
        : base()
    { }

    public NotPrimeException(int value) :
        base(String.Format("{0} is not a prime number.", value))
    {
        notAPrime = value;
    }

    public NotPrimeException(int value, string message)
        : base(message)
    {
        notAPrime = value;
    }
}
```

```

    public NotPrimeException(int value, string message, Exception
innerException) :
        base(message, innerException)
    {
        notAPrime = value;
    }

    protected NotPrimeException(SerializationInfo info,
                                StreamingContext context)
        : base(info, context)
    { }

    public int NonPrime
    { get { return notAPrime; } }
}

```

The `PrimeNumberGenerator` class shown in the following example uses the Sieve of Eratosthenes to calculate the sequence of prime numbers from 2 to a limit specified by the client in the call to its class constructor. The `GetPrimesFrom` method returns all prime numbers that are greater than or equal to a specified lower limit, but throws a `NotPrimeException` if that lower limit is not a prime number.

C#

```

using System;
using System.Collections.Generic;

[Serializable]
public class PrimeNumberGenerator
{
    private const int START = 2;
    private int maxUpperBound = 10000000;
    private int upperBound;
    private bool[] primeTable;
    private List<int> primes = new List<int>();

    public PrimeNumberGenerator(int upperBound)
    {
        if (upperBound > maxUpperBound)
        {
            string message = String.Format(
                "{0} exceeds the maximum upper bound of {1}.",
                upperBound, maxUpperBound);
            throw new ArgumentOutOfRangeException(message);
        }
        this.upperBound = upperBound;
        // Create array and mark 0, 1 as not prime (True).
        primeTable = new bool[upperBound + 1];
        primeTable[0] = true;
        primeTable[1] = true;
    }
}

```

```

// Use Sieve of Eratosthenes to determine prime numbers.
for (int ctr = START; ctr <= (int)Math.Ceiling(Math.Sqrt(upperBound));
     ctr++)
{
    if (primeTable[ctr]) continue;

    for (int multiplier = ctr; multiplier <= upperBound / ctr;
multiplier++)
        if (ctr * multiplier <= upperBound) primeTable[ctr * multiplier]
= true;
}
// Populate array with prime number information.
int index = START;
while (index != -1)
{
    index = Array.FindIndex(primeTable, index, (flag) => !flag);
    if (index >= 1)
    {
        primes.Add(index);
        index++;
    }
}
}

public int[] GetAllPrimes()
{
    return primes.ToArray();
}

public int[] GetPrimesFrom(int prime)
{
    int start = primes.FindIndex((value) => value == prime);
    if (start < 0)
        throw new NotPrimeException(prime, String.Format("{0} is not a
prime number.", prime));
    else
        return primes.FindAll((value) => value >= prime).ToArray();
}
}

```

The following example makes two calls to the `GetPrimesFrom` method with non-prime numbers, one of which crosses application domain boundaries. In both cases, the exception is thrown and successfully handled in client code.

C#

```

using System;
using System.Reflection;

class Example1
{
    public static void Main()

```

```

{
    int limit = 10000000;
    PrimeNumberGenerator primes = new PrimeNumberGenerator(limit);
    int start = 1000001;
    try
    {
        int[] values = primes.GetPrimesFrom(start);
        Console.WriteLine("There are {0} prime numbers from {1} to {2}",
                          start, limit);
    }
    catch (NotPrimeException e)
    {
        Console.WriteLine("{0} is not prime", e.NonPrime);
        Console.WriteLine(e);
        Console.WriteLine("-----");
    }
}

AppDomain domain = AppDomain.CreateDomain("Domain2");
PrimeNumberGenerator gen =
(PrimeNumberGenerator)domain.CreateInstanceAndUnwrap(
    typeof(Example).Assembly.FullName,
    "PrimeNumberGenerator", true,
    BindingFlags.Default, null,
    new object[] { 1000000 }, null,
    null);
try
{
    start = 100;
    Console.WriteLine(gen.GetPrimesFrom(start));
}
catch (NotPrimeException e)
{
    Console.WriteLine("{0} is not prime", e.NonPrime);
    Console.WriteLine(e);
    Console.WriteLine("-----");
}
}
}

```

## Examples

The following example demonstrates a `catch` (with in F#) block that is defined to handle `ArithmetiException` errors. This `catch` block also catches `DivideByZeroException` errors, because `DivideByZeroException` derives from `ArithmetiException` and there is no `catch` block explicitly defined for `DivideByZeroException` errors.

C#

```
using System;
```

```
class ExceptionTestClass
{
    public static void Main()
    {
        int x = 0;
        try
        {
            int y = 100 / x;
        }
        catch (ArithmetiException e)
        {
            Console.WriteLine($"ArithmetiException Handler: {e}");
        }
        catch (Exception e)
        {
            Console.WriteLine($"Generic Exception Handler: {e}");
        }
    }
}
/*
This code example produces the following results:

ArithmetiException Handler: System.DivideByZeroException: Attempted to
divide by zero.
    at ExceptionTestClass.Main()

*/

```

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Exception.Data property

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

Use the [System.Collections.IDictionary](#) object returned by the [Data](#) property to store and retrieve supplementary information relevant to the exception. The information is in the form of an arbitrary number of user-defined key/value pairs. The key component of each key/value pair is typically an identifying string, whereas the value component of the pair can be any type of object.

## Key/value pair security

The key/value pairs stored in the collection returned by the [Data](#) property are not secure. If your application calls a nested series of routines, and each routine contains exception handlers, the resulting call stack contains a hierarchy of those exception handlers. If a lower-level routine throws an exception, any upper-level exception handler in the call stack hierarchy can read and/or modify the key/value pairs stored in the collection by any other exception handler. This means you must guarantee that the information in the key/value pairs is not confidential and that your application will operate correctly if the information in the key/value pairs is corrupted.

## Key conflicts

A key conflict occurs when different exception handlers specify the same key to access a key/value pair. Use caution when developing your application because the consequence of a key conflict is that lower-level exception handlers can inadvertently communicate with higher-level exception handlers, and this communication might cause subtle program errors. However, if you are cautious you can use key conflicts to enhance your application.

## Avoid key conflicts

Avoid key conflicts by adopting a naming convention to generate unique keys for key/value pairs. For example, a naming convention might yield a key that consists of the period-delimited name of your application, the method that provides supplementary information for the pair, and a unique identifier.

Suppose two applications, named Products and Suppliers, each has a method named Sales. The Sales method in the Products application provides the identification number (the stock keeping unit or SKU) of a product. The Sales method in the Suppliers application provides the identification number, or SID, of a supplier. Consequently, the naming convention for this example yields the keys, "Products.Sales.SKU" and "Suppliers.Sales.SID".

## Exploit key conflicts

Exploit key conflicts by using the presence of one or more special, prearranged keys to control processing. Suppose, in one scenario, the highest level exception handler in the call stack hierarchy catches all exceptions thrown by lower-level exception handlers. If a key/value pair with a special key exists, the high-level exception handler formats the remaining key/value pairs in the [IDictionary](#) object in some nonstandard way; otherwise, the remaining key/value pairs are formatted in some normal manner.

Now suppose, in another scenario, the exception handler at each level of the call stack hierarchy catches the exception thrown by the next lower-level exception handler. In addition, each exception handler knows the collection returned by the [Data](#) property contains a set of key/value pairs that can be accessed with a prearranged set of keys.

Each exception handler uses the prearranged set of keys to update the value component of the corresponding key/value pair with information unique to that exception handler. After the update process is complete, the exception handler throws the exception to the next higher-level exception handler. Finally, the highest level exception handler accesses the key/value pairs and displays the consolidated update information from all the lower-level exception handlers.

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Exception.Message property

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

Error messages target the developer who is handling the exception. The text of the [Message](#) property should completely describe the error and, when possible, should also explain how to correct the error. Top-level exception handlers may display the message to end-users, so you should ensure that it is grammatically correct and that each sentence of the message ends with a period. Do not use question marks or exclamation points. If your application uses localized exception messages, you should ensure that they are accurately translated.

## ⓘ Important

Do not disclose sensitive information in exception messages without checking for the appropriate permissions.

The value of the [Message](#) property is included in the information returned by [ToString](#). The [Message](#) property is set only when creating an [Exception](#). If no message was supplied to the constructor for the current instance, the system supplies a default message that is formatted using the current system culture.

## Examples

The following code example throws and then catches an [Exception](#) exception and displays the exception's text message using the [Message](#) property.

C#

```
using System;

namespace NDP_UE_CS
{
    // Derive an exception; the constructor sets the HelpLink and
    // Source properties.
    class LogTableOverflowException : Exception
    {
        const string overflowMessage = "The log table has overflowed.";

        public LogTableOverflowException(
            string auxMessage, Exception inner) :

```

```

        base(String.Format("{0} - {1}",
                           overflowMessage, auxMessage), inner)
    {
        this.HelpLink = "https://learn.microsoft.com";
        this.Source = "Exception_Class_Samples";
    }
}

class LogTable
{
    public LogTable(int numElements)
    {
        logArea = new string[numElements];
        elemInUse = 0;
    }

    protected string[] logArea;
    protected int elemInUse;

    // The AddRecord method throws a derived exception if
    // the array bounds exception is caught.
    public int AddRecord(string newRecord)
    {
        try
        {
            logArea[elemInUse] = newRecord;
            return elemInUse++;
        }
        catch (Exception e)
        {
            throw new LogTableOverflowException(
                String.Format("Record \"{0}\" was not logged.",
                             newRecord), e);
        }
    }
}

class OverflowDemo
{
    // Create a log table and force an overflow.
    public static void Main()
    {
        LogTable log = new LogTable(4);

        Console.WriteLine(
            "This example of \n    Exception.Message, \n" +
            "    Exception.HelpLink, \n    Exception.Source, \n" +
            "    Exception.StackTrace, and \n    Exception." +
            "TargetSite \ngenerates the following output.");
    }

    try
    {
        for (int count = 1; ; count++)
        {
            log.AddRecord(

```

```
        String.Format(
            "Log record number {0}", count));
    }
}
catch (Exception ex)
{
    Console.WriteLine("\nMessage ---\n{0}", ex.Message);
    Console.WriteLine(
        "\nHelpLink ---\n{0}", ex.HelpLink);
    Console.WriteLine("\nSource ---\n{0}", ex.Source);
    Console.WriteLine(
        "\nStackTrace ---\n{0}", ex.StackTrace);
    Console.WriteLine(
        "\nTargetSite ---\n{0}", ex.TargetSite);
}
}
}
```

```
/*
This example of
Exception.Message,
Exception.HelpLink,
Exception.Source,
Exception.StackTrace, and
Exception.TargetSite
generates the following output.
```

```
Message ---
The log table has overflowed. - Record "Log record number 5" was not logged.
```

```
HelpLink ---
https://learn.microsoft.com
```

```
Source ---
Exception_Class_Samples
```

```
StackTrace ---
    at NDP_UE_CS.LogTable.AddRecord(String newRecord)
    at NDP_UE_CS.OverflowDemo.Main()
```

```
TargetSite ---
Int32 AddRecord(System.String)
*/
```

 Collaborate with us on  
GitHub

The source for this content can  
be found on GitHub, where you  
can also create and review

.NET

.NET feedback

.NET is an open source project.  
Select a link to provide feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.InvalidCastException class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

.NET supports automatic conversion from derived types to their base types and back to the derived type, as well as from types that present interfaces to interface objects and back. It also includes a variety of mechanisms that support custom conversions. For more information, see [Type Conversion in .NET](#).

An [InvalidCastException](#) exception is thrown when the conversion of an instance of one type to another type is not supported. For example, attempting to convert a [Char](#) value to a [DateTime](#) value throws an [InvalidCastException](#) exception. It differs from an [OverflowException](#) exception, which is thrown when a conversion of one type to another is supported, but the value of the source type is outside the range of the target type. An [InvalidCastException](#) exception is caused by developer error and should not be handled in a `try/catch` block. Instead, the cause of the exception should be eliminated.

For information about conversions supported by the system, see the [Convert](#) class. For errors that occur when the destination type can store source type values but is not large enough to store a specific source value, see the [OverflowException](#) exception.

## ⓘ Note

In many cases, your language compiler detects that no conversion exists between the source type and the target type and issues a compiler error.

Some of the conditions under which an attempted conversion throws an [InvalidCastException](#) exception are discussed in the following sections.

For an explicit reference conversion to be successful, the source value must be `null`, or the object type referenced by the source argument must be convertible to the destination type by an implicit reference conversion.

The following intermediate language (IL) instructions throw an [InvalidCastException](#) exception:

- `castclass`
- `refanyval`
- `unbox`

[InvalidOperationException](#) uses the HRESULT `COR_E_INVALIDCAST`, which has the value 0x80004002.

For a list of initial property values for an instance of [InvalidOperationException](#), see the [InvalidOperationException](#) constructors.

## Primitive types and [IConvertible](#)

You directly or indirectly call a primitive type's [IConvertible](#) implementation that does not support a particular conversion. For example, trying to convert a [Boolean](#) value to a [Char](#) or a [DateTime](#) value to an [Int32](#) throws an [InvalidOperationException](#) exception. The following example calls both the [Boolean.IConvertible.ToChar](#) and [Convert.ToChar\(Boolean\)](#) methods to convert a [Boolean](#) value to a [Char](#). In both cases, the method call throws an [InvalidOperationException](#) exception.

C#

```
using System;

public class IConvertibleEx
{
    public static void Main()
    {
        bool flag = true;
        try
        {
            IConvertible conv = flag;
            Char ch = conv.ToChar(null);
            Console.WriteLine("Conversion succeeded.");
        }
        catch (InvalidOperationException)
        {
            Console.WriteLine("Cannot convert a Boolean to a Char.");
        }

        try
        {
            Char ch = Convert.ToChar(flag);
            Console.WriteLine("Conversion succeeded.");
        }
        catch (InvalidOperationException)
        {
            Console.WriteLine("Cannot convert a Boolean to a Char.");
        }
    }
}

// The example displays the following output:
```

```
//      Cannot convert a Boolean to a Char.  
//      Cannot convert a Boolean to a Char.
```

Because the conversion is not supported, there is no workaround.

## The Convert.ChangeType method

You've called the [Convert.ChangeType](#) method to convert an object from one type to another, but one or both types don't implement the [IConvertible](#) interface.

In most cases, because the conversion is not supported, there is no workaround. In some cases, a possible workaround is to manually assign property values from the source type to similar properties of the target type.

## Narrowing conversions and IConvertible implementations

Narrowing operators define the explicit conversions supported by a type. A casting operator in C# or the  `CType` conversion method in Visual Basic (if  `Option Strict` is on) is required to perform the conversion.

However, if neither the source type nor the target type defines an explicit or narrowing conversion between the two types, and the [IConvertible](#) implementation of one or both types doesn't support a conversion from the source type to the target type, an [InvalidOperationException](#) exception is thrown.

In most cases, because the conversion is not supported, there is no workaround.

## Downcasting

You're downcasting, that is, trying to convert an instance of a base type to one of its derived types. In the following example, trying to convert a  `Person` object to a  `PersonWithID` object fails.

C#

```
using System;  
  
public class Person  
{  
    String _name;
```

```
public String Name
{
    get { return _name; }
    set { _name = value; }
}

public class PersonWithId : Person
{
    String _id;

    public string Id
    {
        get { return _id; }
        set { _id = value; }
    }
}

public class Example
{
    public static void Main()
    {
        Person p = new Person();
        p.Name = "John";
        try {
            PersonWithId pid = (PersonWithId) p;
            Console.WriteLine("Conversion succeeded.");
        }
        catch (InvalidCastException) {
            Console.WriteLine("Conversion failed.");
        }

        PersonWithId pid1 = new PersonWithId();
        pid1.Name = "John";
        pid1.Id = "246";
        Person p1 = pid1;
        try {
            PersonWithId pid1a = (PersonWithId) p1;
            Console.WriteLine("Conversion succeeded.");
        }
        catch (InvalidCastException) {
            Console.WriteLine("Conversion failed.");
        }

        Person p2 = null;
        try {
            PersonWithId pid2 = (PersonWithId) p2;
            Console.WriteLine("Conversion succeeded.");
        }
        catch (InvalidCastException) {
            Console.WriteLine("Conversion failed.");
        }
    }
}

// The example displays the following output:
```

```
//      Conversion failed.
//      Conversion succeeded.
//      Conversion succeeded.
```

As the example shows, the downcast succeeds only if the `Person` object was created by an upcast from a `PersonWithId` object to a `Person` object, or if the `Person` object is `null`.

## Conversion from an interface object

You're attempting to convert an interface object to a type that implements that interface, but the target type is not the same type or a base class of the type from which the interface object was originally derived. The following example throws an `InvalidOperationException` exception when it attempts to convert an `IFormatProvider` object to a `DateTimeFormatInfo` object. The conversion fails because although the `DateTimeFormatInfo` class implements the `IFormatProvider` interface, the `DateTimeFormatInfo` object is not related to the `CultureInfo` class from which the interface object was derived.

C#

```
using System;
using System.Globalization;

public class InterfaceEx
{
    public static void Main()
    {
        var culture = CultureInfo.InvariantCulture;
        IFormatProvider provider = culture;

        DateTimeFormatInfo dt = (DateTimeFormatInfo)provider;
    }
}

// The example displays the following output:
//      Unhandled Exception: System.InvalidCastException:
//          Unable to cast object of type //System.Globalization.CultureInfo//
//          to
//          type //System.Globalization.DateTimeFormatInfo//.
//          at Example.Main()
```

As the exception message indicates, the conversion would succeed only if the interface object is converted back to an instance of the original type, in this case a `CultureInfo`. The conversion would also succeed if the interface object is converted to an instance of a base type of the original type.

# String conversions

You're trying to convert a value or an object to its string representation by using a casting operator in C#. In the following example, both the attempt to cast a `Char` value to a string and the attempt to cast an integer to a string throw an `InvalidCastException` exception.

C#

```
public class StringEx
{
    public static void Main()
    {
        object value = 12;
        // Cast throws an InvalidCastException exception.
        string s = (string)value;
    }
}
```

## ① Note

Using the Visual Basic `cstr` operator to convert a value of a primitive type to a string succeeds. The operation does not throw an `InvalidCastException` exception.

To successfully convert an instance of any type to its string representation, call its `ToString` method, as the following example does. The `ToString` method is always present, since the `ToString` method is defined by the `Object` class and therefore is either inherited or overridden by all managed types.

C#

```
using System;

public class ToStringEx2
{
    public static void Main()
    {
        object value = 12;
        string s = value.ToString();
        Console.WriteLine(s);
    }
}
// The example displays the following output:
//      12
```

# Visual Basic 6.0 migration

You're upgrading a Visual Basic 6.0 application with a call to a custom event in a user control to Visual Basic .NET, and an `InvalidCastException` exception is thrown with the message, "Specified cast is not valid." To eliminate this exception, change the line of code in your form (such as `Form1`)

VB

```
Call UserControl11_MyCustomEvent(UserControl11, New
UserControl1.MyCustomEventArgs(5))
```

and replace it with the following line of code:

VB

```
Call UserControl11_MyCustomEvent(UserControl11(0), New
UserControl1.MyCustomEventArgs(5))
```

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.InvalidOperationException class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

[InvalidOperationException](#) is used in cases when the failure to invoke a method is caused by reasons other than invalid arguments. Typically, it's thrown when the state of an object cannot support the method call. For example, an [InvalidOperationException](#) exception is thrown by methods such as:

- [IEnumerator.MoveNext](#) if objects of a collection are modified after the enumerator is created. For more information, see [Changing a collection while iterating it](#).
- [ResourceSet.GetString](#) if the resource set is closed before the method call is made.
- [XContainer.Add](#), if the object or objects to be added would result in an incorrectly structured XML document.
- A method that attempts to manipulate the UI from a thread that is not the main or UI thread.

## Important

Because the [InvalidOperationException](#) exception can be thrown in a wide variety of circumstances, it is important to read the exception message returned by the [Message](#) property.

[InvalidOperationException](#) uses the HRESULT `COR_E_INVALIDOPERATION`, which has the value 0x80131509.

For a list of initial property values for an instance of [InvalidOperationException](#), see the [InvalidOperationException](#) constructors.

## Common causes of InvalidOperationException exceptions

The following sections show how some common cases in which an [InvalidOperationException](#) exception is thrown in an app. How you handle the issue depends on the specific situation. Most commonly, however, the exception results from developer error, and the [InvalidOperationException](#) exception can be anticipated and avoided.

# Updating a UI thread from a non-UI thread

Often, worker threads are used to perform some background work that involves gathering data to be displayed in an application's user interface. However, most GUI (graphical user interface) application frameworks for .NET, such as Windows Forms and Windows Presentation Foundation (WPF), let you access GUI objects only from the thread that creates and manages the UI (the Main or UI thread). An [InvalidOperationException](#) is thrown when you try to access a UI element from a thread other than the UI thread. The text of the exception message is shown in the following table.

[] [Expand table](#)

Application Type	Message
WPF app	The calling thread cannot access this object because a different thread owns it.
UWP app	The application called an interface that was marshaled for a different thread.
Windows Forms app	Cross-thread operation not valid: Control 'TextBox1' accessed from a thread other than the thread it was created on.

UI frameworks for .NET implement a *dispatcher* pattern that includes a method to check whether a call to a member of a UI element is being executed on the UI thread, and other methods to schedule the call on the UI thread:

- In WPF apps, call the [Dispatcher.CheckAccess](#) method to determine if a method is running on a non-UI thread. It returns `true` if the method is running on the UI thread and `false` otherwise. Call one of the overloads of the [Dispatcher.Invoke](#) method to schedule the call on the UI thread.
- In UWP apps, check the [CoreDispatcher.HasThreadAccess](#) property to determine if a method is running on a non-UI thread. Call the [CoreDispatcher.RunAsync](#) method to execute a delegate that updates the UI thread.
- In Windows Forms apps, use the [Control.InvokeRequired](#) property to determine if a method is running on a non-UI thread. Call one of the overloads of the [Control.Invoke](#) method to execute a delegate that updates the UI thread.

The following examples illustrate the [InvalidOperationException](#) exception that is thrown when you attempt to update a UI element from a thread other than the thread that created it. Each example requires that you create two controls:

- A text box control named `textBox1`. In a Windows Forms app, you should set its `Multiline` property to `true`.
- A button control named `threadExampleBtn`. The example provides a handler, `ThreadsExampleBtn_Click`, for the button's `Click` event.

In each case, the `threadExampleBtn_Click` event handler calls the `DoSomeWork` method twice. The first call runs synchronously and succeeds. But the second call, because it runs asynchronously on a thread pool thread, attempts to update the UI from a non-UI thread. This results in a `InvalidOperationException` exception.

## WPF apps

C#

```
private async void threadExampleBtn_Click(object sender, RoutedEventArgs e)
{
    textBox1.Text = String.Empty;

    textBox1.Text = "Simulating work on UI thread.\n";
    DoSomeWork(20);
    textBox1.Text += "Work completed...\n";

    textBox1.Text += "Simulating work on non-UI thread.\n";
    await Task.Run(() => DoSomeWork(1000));
    textBox1.Text += "Work completed...\n";
}

private async void DoSomeWork(int milliseconds)
{
    // Simulate work.
    await Task.Delay(milliseconds);

    // Report completion.
    var msg = String.Format("Some work completed in {0} ms.\n",
    milliseconds);
    textBox1.Text += msg;
}
```

The following version of the `DoSomeWork` method eliminates the exception in a WPF app.

C#

```
private async void DoSomeWork(int milliseconds)
{
    // Simulate work.
    await Task.Delay(milliseconds);

    // Report completion.
```

```

    bool uiAccess = textBox1.Dispatcher.CheckAccess();
    String msg = String.Format("Some work completed in {0} ms. on {1}UI
thread\n",
                                milliseconds, uiAccess ? String.Empty : "non-
");
    if (uiAccess)
        textBox1.Text += msg;
    else
        textBox1.Dispatcher.Invoke(() => { textBox1.Text += msg; });
}

```

## Windows Forms apps

C#

```

List<String> lines = new List<String>();

private async void threadExampleBtn_Click(object sender, EventArgs e)
{
    textBox1.Text = String.Empty;
    lines.Clear();

    lines.Add("Simulating work on UI thread.");
    textBox1.Lines = lines.ToArray();
    DoSomeWork(20);

    lines.Add("Simulating work on non-UI thread.");
    textBox1.Lines = lines.ToArray();
    await Task.Run(() => DoSomeWork(1000));

    lines.Add("ThreadsExampleBtn_Click completes. ");
    textBox1.Lines = lines.ToArray();
}

private async void DoSomeWork(int milliseconds)
{
    // simulate work
    await Task.Delay(milliseconds);

    // report completion
    lines.Add(String.Format("Some work completed in {0} ms on UI thread.",
milliseconds));
    textBox1.Lines = lines.ToArray();
}

```

The following version of the `DoSomeWork` method eliminates the exception in a Windows Forms app.

C#

```

private async void DoSomeWork(int milliseconds)
{
    // simulate work
    await Task.Delay(milliseconds);

    // Report completion.
    bool uiMarshal = textBox1.InvokeRequired;
    String msg = String.Format("Some work completed in {0} ms. on {1}UI
thread\n",
                                milliseconds, uiMarshal ? String.Empty :
"non-");
    lines.Add(msg);

    if (uiMarshal) {
        textBox1.Invoke(new Action(() => { textBox1.Lines = lines.ToArray();
}));
    }
    else {
        textBox1.Lines = lines.ToArray();
    }
}

```

## Changing a collection while iterating it

The `foreach` statement in C#, `for...in` in F#, or `For Each` statement in Visual Basic is used to iterate the members of a collection and to read or modify its individual elements. However, it can't be used to add or remove items from the collection. Doing this throws an `InvalidOperationException` exception with a message that is similar to, "Collection was modified; enumeration operation may not execute."

The following example iterates a collection of integers attempts to add the square of each integer to the collection. The example throws an `InvalidOperationException` with the first call to the `List<T>.Add` method.

C#

```

using System;
using System.Collections.Generic;

public class IteratingEx1
{
    public static void Main()
    {
        var numbers = new List<int>() { 1, 2, 3, 4, 5 };
        foreach (var number in numbers)
        {
            int square = (int)Math.Pow(number, 2);
            Console.WriteLine("{0}^{1}", number, square);
            Console.WriteLine("Adding {0} to the collection...\n", square);
        }
    }
}

```

```

        numbers.Add(square);
    }
}
// The example displays the following output:
//    1^1
//    Adding 1 to the collection...
//
//
//    Unhandled Exception: System.InvalidOperationException: Collection was
modified;
//        enumeration operation may not execute.
//        at
System.ThrowHelper.ThrowInvalidOperationException(ExceptionResource
resource)
//        at System.Collections.Generic.List`1.Enumerator.MoveNextRare()
//        at Example.Main()

```

You can eliminate the exception in one of two ways, depending on your application logic:

- If elements must be added to the collection while iterating it, you can iterate it by index using the `for (for..to in F#)` statement instead of `foreach`, `for...in`, or `For Each`. The following example uses the `for` statement to add the square of numbers in the collection to the collection.

C#

```

using System;
using System.Collections.Generic;

public class IteratingEx2
{
    public static void Main()
    {
        var numbers = new List<int>() { 1, 2, 3, 4, 5 };

        int upperBound = numbers.Count - 1;
        for (int ctr = 0; ctr <= upperBound; ctr++)
        {
            int square = (int)Math.Pow(numbers[ctr], 2);
            Console.WriteLine("{0}^{1}", numbers[ctr], square);
            Console.WriteLine("Adding {0} to the collection...\n",
square);
            numbers.Add(square);
        }

        Console.WriteLine("Elements now in the collection: ");
        foreach (var number in numbers)
            Console.Write("{0}    ", number);
    }
}

```

```

}
// The example displays the following output:
//      1^1
//      Adding 1 to the collection...
//
//      2^4
//      Adding 4 to the collection...
//
//      3^9
//      Adding 9 to the collection...
//
//      4^16
//      Adding 16 to the collection...
//
//      5^25
//      Adding 25 to the collection...
//
//      Elements now in the collection:
//      1      2      3      4      5      1      4      9      16      25

```

Note that you must establish the number of iterations before iterating the collection either by using a counter inside the loop that will exit the loop appropriately, by iterating backward, from `Count - 1` to 0, or, as the example does, by assigning the number of elements in the array to a variable and using it to establish the upper bound of the loop. Otherwise, if an element is added to the collection on every iteration, an endless loop results.

- If it is not necessary to add elements to the collection while iterating it, you can store the elements to be added in a temporary collection that you add when iterating the collection has finished. The following example uses this approach to add the square of numbers in a collection to a temporary collection, and then to combine the collections into a single array object.

C#

```

using System;
using System.Collections.Generic;

public class IteratingEx3
{
    public static void Main()
    {
        var numbers = new List<int>() { 1, 2, 3, 4, 5 };
        var temp = new List<int>();

        // Square each number and store it in a temporary collection.
        foreach (var number in numbers)
        {
            int square = (int)Math.Pow(number, 2);
            temp.Add(square);
        }

        // Print the temporary collection.
        foreach (var square in temp)
        {
            Console.WriteLine(square);
        }
    }
}

```

```

        }

        // Combine the numbers into a single array.
        int[] combined = new int[numbers.Count + temp.Count];
        Array.Copy(numbers.ToArray(), 0, combined, 0, numbers.Count);
        Array.Copy(temp.ToArray(), 0, combined, numbers.Count,
temp.Count);

        // Iterate the array.
        foreach (var value in combined)
            Console.Write("{0}    ", value);
    }
}

// The example displays the following output:
//      1    2    3    4    5    1    4    9    16    25

```

## Sorting an array or collection whose objects cannot be compared

General-purpose sorting methods, such as the [Array.Sort\(Array\)](#) method or the [List<T>.Sort\(\)](#) method, usually require that at least one of the objects to be sorted implement the [IComparable<T>](#) or the [IComparable](#) interface. If not, the collection or array cannot be sorted, and the method throws an [InvalidOperationException](#) exception. The following example defines a `Person` class, stores two `Person` objects in a generic `List<T>` object, and attempts to sort them. As the output from the example shows, the call to the [List<T>.Sort\(\)](#) method throws an [InvalidOperationException](#).

C#

```

using System;
using System.Collections.Generic;

public class Person1
{
    public Person1(string fName, string lName)
    {
        FirstName = fName;
        LastName = lName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class ListSortEx1
{
    public static void Main()
    {
        var people = new List<Person1>();

```

```

    people.Add(new Person1("John", "Doe"));
    people.Add(new Person1("Jane", "Doe"));
    people.Sort();
    foreach (var person in people)
        Console.WriteLine("{0} {1}", person.FirstName, person.LastName);
}
}

// The example displays the following output:
//      Unhandled Exception: System.InvalidOperationException: Failed to
//      compare two elements in the array. --->
//      System.ArgumentException: At least one object must implement
//      IComparable.
//          at System.Collections.Comparer.Compare(Object a, Object b)
//          at System.Collections.Generic.ArraySortHelper`1.SwapIfGreater(T[]
keys, IComparer`1 comparer, Int32 a, Int32 b)
//          at
System.Collections.Generic.ArraySortHelper`1.DepthLimitedQuickSort(T[] keys,
Int32 left, Int32 right, IComparer`1 comparer, Int32 depthLimit)
//          at System.Collections.Generic.ArraySortHelper`1.Sort(T[] keys,
Int32 index, Int32 length, IComparer`1 comparer)
//          --- End of inner exception stack trace ---
//          at System.Collections.Generic.ArraySortHelper`1.Sort(T[] keys,
Int32 index, Int32 length, IComparer`1 comparer)
//          at System.Array.Sort[T](T[] array, Int32 index, Int32 length,
IComparer`1 comparer)
//          at System.Collections.Generic.List`1.Sort(Int32 index, Int32 count,
IComparer`1 comparer)
//          at Example.Main()

```

You can eliminate the exception in any of three ways:

- If you can own the type that you are trying to sort (that is, if you control its source code), you can modify it to implement the `IComparable<T>` or the `IComparable` interface. This requires that you implement either the `IComparable<T>.CompareTo` or the `CompareTo` method. Adding an interface implementation to an existing type is not a breaking change.

The following example uses this approach to provide an `IComparable<T>` implementation for the `Person` class. You can still call the collection or array's general sorting method and, as the output from the example shows, the collection sorts successfully.

C#

```

using System;
using System.Collections.Generic;

public class Person2 : IComparable<Person>
{

```

```

public Person2(String fName, String lName)
{
    FirstName = fName;
    LastName = lName;
}

public String FirstName { get; set; }
public String LastName { get; set; }

public int CompareTo(Person other)
{
    return String.Format("{0} {1}", LastName, FirstName).
        CompareTo(String.Format("{0} {1}", other.LastName,
other.FirstName));
}

public class ListSortEx2
{
    public static void Main()
    {
        var people = new List<Person2>();

        people.Add(new Person2("John", "Doe"));
        people.Add(new Person2("Jane", "Doe"));
        people.Sort();
        foreach (var person in people)
            Console.WriteLine("{0} {1}", person.FirstName,
person.LastName);
    }
}
// The example displays the following output:
//      Jane Doe
//      John Doe

```

- If you cannot modify the source code for the type you are trying to sort, you can define a special-purpose sorting class that implements the `IComparer<T>` interface. You can call an overload of the `Sort` method that includes an `IComparer<T>` parameter. This approach is especially useful if you want to develop a specialized sorting class that can sort objects based on multiple criteria.

The following example uses the approach by developing a custom `PersonComparer` class that is used to sort `Person` collections. It then passes an instance of this class to the `List<T>.Sort(IComparer<T>)` method.

C#

```

using System;
using System.Collections.Generic;

public class Person3

```

```

{
    public Person3(String fName, String lName)
    {
        FirstName = fName;
        LastName = lName;
    }

    public String FirstName { get; set; }
    public String LastName { get; set; }
}

public class PersonComparer : IComparer<Person3>
{
    public int Compare(Person3 x, Person3 y)
    {
        return String.Format("{0} {1}", x.LastName, x.FirstName).
            CompareTo(String.Format("{0} {1}", y.LastName,
y.FirstName));
    }
}

public class ListSortEx3
{
    public static void Main()
    {
        var people = new List<Person3>();

        people.Add(new Person3("John", "Doe"));
        people.Add(new Person3("Jane", "Doe"));
        people.Sort(new PersonComparer());
        foreach (var person in people)
            Console.WriteLine("{0} {1}", person.FirstName,
person.LastName);
    }
}

// The example displays the following output:
//      Jane Doe
//      John Doe

```

- If you cannot modify the source code for the type you are trying to sort, you can create a `Comparison<T>` delegate to perform the sorting. The delegate signature is

C#

```
int Comparison<T>(T x, T y)
```

The following example uses the approach by defining a `PersonComparison` method that matches the `Comparison<T>` delegate signature. It then passes this delegate to the `List<T>.Sort(Comparison<T>)` method.

C#

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

public class Person
{
    public Person(String fName, String lName)
    {
        FirstName = fName;
        LastName = lName;
    }

    public String FirstName { get; set; }
    public String LastName { get; set; }
}

public class ListSortEx4
{
    public static void Main()
    {
        var people = new List<Person>();

        people.Add(new Person("John", "Doe"));
        people.Add(new Person("Jane", "Doe"));
        people.Sort(PersonComparison);
        foreach (var person in people)
            Console.WriteLine("{0} {1}", person.FirstName,
person.LastName);
    }

    public static int PersonComparison(Person x, Person y)
    {
        return String.Format("{0} {1}", x.LastName, x.FirstName).
            CompareTo(String.Format("{0} {1}", y.LastName,
y.FirstName));
    }
}
// The example displays the following output:
//      Jane Doe
//      John Doe
```

## Casting a Nullable<T> that's null to its underlying type

Attempting to cast a `Nullable<T>` value that is `null` to its underlying type throws an `InvalidOperationException` exception and displays the error message, "Nullable object must have a value.

The following example throws an `InvalidOperationException` exception when it attempts to iterate an array that includes a `Nullable(Of Integer)` value.

C#

```
using System;
using System.Linq;

public class NullableEx1
{
    public static void Main()
    {
        var queryResult = new int?[] { 1, 2, null, 4 };
        var map = queryResult.Select(nullableInt => (int)nullableInt);

        // Display list.
        foreach (var num in map)
            Console.Write("{0} ", num);
        Console.WriteLine();
    }
}

// The example displays the following output:
//    1 2
//    Unhandled Exception: System.InvalidOperationException: Nullable object
//    must have a value.
//        at
System.ThrowHelper.ThrowInvalidOperationException(ExceptionResource
resource)
//        at Example.<Main>b__0(Nullable`1 nullableInt)
//        at System.Linq.Enumerable.WhereSelectArrayIterator`2.MoveNext()
//        at Example.Main()
```

To prevent the exception:

- Use the `Nullable<T>.HasValue` property to select only those elements that are not `null`.
- Call one of the `Nullable<T>.GetValueOrDefault` overloads to provide a default value for a `null` value.

The following example does both to avoid the `InvalidOperationException` exception.

C#

```
using System;
using System.Linq;

public class NullableEx2
{
    public static void Main()
    {
        var queryResult = new int?[] { 1, 2, null, 4 };
        var numbers = queryResult.Select(nullableInt =>
(int)nullableInt.GetValueOrDefault());
```

```

// Display list using Nullable<int>.HasValue.
foreach (var number in numbers)
    Console.Write("{0} ", number);
Console.WriteLine();

numbers = queryResult.Select(nullableInt => (int)
(nullableInt.HasValue ? nullableInt : -1));
// Display list using Nullable<int>.GetValueOrDefault.
foreach (var number in numbers)
    Console.Write("{0} ", number);
Console.WriteLine();
}
}
// The example displays the following output:
//      1 2 0 4
//      1 2 -1 4

```

## Call a System.Linq.Enumerable method on an empty collection

The `Enumerable.Average`, `Enumerable.Average`, `Enumerable.First`, `Enumerable.Last`, `Enumerable.Max`, `Enumerable.Min`, `Enumerable.Single`, and `Enumerable.SingleOrDefault` methods perform operations on a sequence and return a single result. Some overloads of these methods throw an `InvalidOperationException` exception when the sequence is empty, while other overloads return `null`. The `Enumerable.SingleOrDefault` method also throws an `InvalidOperationException` exception when the sequence contains more than one element.

### ⓘ Note

Most of the methods that throw an `InvalidOperationException` exception are overloads. Be sure that you understand the behavior of the overload that you choose.

The following table lists the exception messages from the `InvalidOperationException` exception objects thrown by calls to some `System.Linq.Enumerable` methods.

[ ] [Expand table](#)

Method	Message
<code>Aggregate</code>	Sequence contains no elements
<code>Average</code>	
<code>Last</code>	

Method	Message
Max	
Min	
First	Sequence contains no matching element
Single	Sequence contains more than one matching element
SingleOrDefault	

How you eliminate or handle the exception depends on your application's assumptions and on the particular method you call.

- When you deliberately call one of these methods without checking for an empty sequence, you are assuming that the sequence is not empty, and that an empty sequence is an unexpected occurrence. In this case, catching or rethrowing the exception is appropriate.
- If your failure to check for an empty sequence was inadvertent, you can call one of the overloads of the [Enumerable.Any](#) overload to determine whether a sequence contains any elements.

 **Tip**

Calling the [Enumerable.Any<TSource>\(IQueryable<TSource>, Func<TSource, Boolean>\)](#) method before generating a sequence can improve performance if the data to be processed might contain a large number of elements or if operation that generates the sequence is expensive.

- If you've called a method such as [Enumerable.First](#), [Enumerable.Last](#), or [Enumerable.Single](#), you can substitute an alternate method, such as [Enumerable.FirstOrDefault](#), [Enumerable.LastOrDefault](#), or [Enumerable.SingleOrDefault](#), that returns a default value instead of a member of the sequence.

The examples provide additional detail.

The following example uses the [Enumerable.Average](#) method to compute the average of a sequence whose values are greater than 4. Since no values from the original array exceed 4, no values are included in the sequence, and the method throws an [InvalidOperationException](#) exception.

C#

```

using System;
using System.Linq;

public class Example
{
    public static void Main()
    {
        int[] data = { 1, 2, 3, 4 };
        var average = data.Where(num => num > 4).Average();
        Console.WriteLine("The average of numbers greater than 4 is {0}",
                        average);
    }
}
// The example displays the following output:
//      Unhandled Exception: System.InvalidOperationException: Sequence
contains no elements
//          at System.Linq.Enumerable.Average(IEnumerable`1 source)
//          at Example.Main()

```

The exception can be eliminated by calling the [Any](#) method to determine whether the sequence contains any elements before calling the method that processes the sequence, as the following example shows.

C#

```

using System;
using System.Linq;

public class EnumerableEx2
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };
        var moreThan4 = dbQueryResults.Where(num => num > 4);

        if (moreThan4.Any())
            Console.WriteLine("Average value of numbers greater than 4:
{0}:",
                            moreThan4.Average());
        else
            // handle empty collection
            Console.WriteLine("The dataset has no values greater than 4.");
    }
}
// The example displays the following output:
//      The dataset has no values greater than 4.

```

The [Enumerable.First](#) method returns the first item in a sequence or the first element in a sequence that satisfies a specified condition. If the sequence is empty and therefore does not have a first element, it throws an [InvalidOperationException](#) exception.

In the following example, the `Enumerable.First<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)` method throws an `InvalidOperationException` exception because the `dbQueryResults` array doesn't contain an element greater than 4.

C#

```
using System;
using System.Linq;

public class EnumerableEx3
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };

        var firstNum = dbQueryResults.First(n => n > 4);

        Console.WriteLine("The first value greater than 4 is {0}",
                          firstNum);
    }
}

// The example displays the following output:
//     Unhandled Exception: System.InvalidOperationException:
//         Sequence contains no matching element
//         at System.Linq.Enumerable.First[TSource](IEnumerable`1 source,
Func`2 predicate)
//         at Example.Main()
```

You can call the `Enumerable.FirstOrDefault` method instead of `Enumerable.First` to return a specified or default value. If the method does not find a first element in the sequence, it returns the default value for that data type. The default value is `null` for a reference type, zero for a numeric data type, and `DateTime.MinValue` for the `DateTime` type.

### ⓘ Note

Interpreting the value returned by the `Enumerable.FirstOrDefault` method is often complicated by the fact that the default value of the type can be a valid value in the sequence. In this case, you can call the `Enumerable.Any` method to determine whether the sequence has valid members before calling the `Enumerable.First` method.

The following example calls the `Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)` method to prevent the `InvalidOperationException` exception thrown in the previous example.

C#

```

using System;
using System.Linq;

public class EnumerableEx4
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };

        var firstNum = dbQueryResults.FirstOrDefault(n => n > 4);

        if (firstNum == 0)
            Console.WriteLine("No value is greater than 4.");
        else
            Console.WriteLine("The first value greater than 4 is {0}",
                firstNum);
    }
}

// The example displays the following output:
//      No value is greater than 4.

```

## Call `Enumerable.Single` or `Enumerable.SingleOrDefault` on a sequence without one element

The `Enumerable.Single` method returns the only element of a sequence, or the only element of a sequence that meets a specified condition. If there are no elements in the sequence, or if there is more than one element , the method throws an `InvalidOperationException` exception.

You can use the `Enumerable.SingleOrDefault` method to return a default value instead of throwing an exception when the sequence contains no elements. However, the `Enumerable.SingleOrDefault` method still throws an `InvalidOperationException` exception when the sequence contains more than one element.

The following table lists the exception messages from the `InvalidOperationException` exception objects thrown by calls to the `Enumerable.Single` and `Enumerable.SingleOrDefault` methods.

[+] Expand table

Method	Message
<code>Single</code>	Sequence contains no matching element
<code>Single</code>	Sequence contains more than one matching element
<code>SingleOrDefault</code>	

In the following example, the call to the `Enumerable.Single` method throws an `InvalidOperationException` exception because the sequence doesn't have an element greater than 4.

C#

```
using System;
using System.Linq;

public class EnumerableEx5
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };

        var singleObject = dbQueryResults.Single(value => value > 4);

        // Display results.
        Console.WriteLine("{0} is the only value greater than 4",
singleObject);
    }
}

// The example displays the following output:
//     Unhandled Exception: System.InvalidOperationException:
//         Sequence contains no matching element
//         at System.Linq.Enumerable.Single[TSource](IEnumerable`1 source,
Func`2 predicate)
//         at Example.Main()
```

The following example attempts to prevent the `InvalidOperationException` exception thrown when a sequence is empty by instead calling the `Enumerable.SingleOrDefault` method. However, because this sequence returns multiple elements whose value is greater than 2, it also throws an `InvalidOperationException` exception.

C#

```
using System;
using System.Linq;

public class EnumerableEx6
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };

        var singleObject = dbQueryResults.SingleOrDefault(value => value >
2);

        if (singleObject != 0)
            Console.WriteLine("{0} is the only value greater than 2",
singleObject);
```

```
        else
            // Handle an empty collection.
            Console.WriteLine("No value is greater than 2");
    }
}

// The example displays the following output:
//      Unhandled Exception: System.InvalidOperationException:
//          Sequence contains more than one matching element
//          at System.Linq.Enumerable.SingleOrDefault[TSource](IEnumerable`1
// source, Func`2 predicate)
//          at Example.Main()
```

Calling the [Enumerable.Single](#) method assumes that either a sequence or the sequence that meets specified criteria contains only one element. [Enumerable.SingleOrDefault](#) assumes a sequence with zero or one result, but no more. If this assumption is a deliberate one on your part and these conditions are not met, rethrowing or catching the resulting [InvalidOperationException](#) is appropriate. Otherwise, or if you expect that invalid conditions will occur with some frequency, you should consider using some other [Enumerable](#) method, such as [FirstOrDefault](#) or [Where](#).

## Dynamic cross-application domain field access

The [OpCodes.Ldflda](#) Microsoft intermediate language (MSIL) instruction throws an [InvalidOperationException](#) exception if the object containing the field whose address you are trying to retrieve is not within the application domain in which your code is executing. The address of a field can only be accessed from the application domain in which it resides.

## Throw an [InvalidOperationException](#) exception

You should throw an [InvalidOperationException](#) exception only when the state of your object for some reason does not support a particular method call. That is, the method call is valid in some circumstances or contexts, but is invalid in others.

If the method invocation failure is due to invalid arguments, then [ArgumentException](#) or one of its derived classes, [ArgumentNullException](#) or [ArgumentOutOfRangeException](#), should be thrown instead.

 Collaborate with us on  
GitHub

.NET

[.NET feedback](#)

.NET is an open source project.  
Select a link to provide feedback:

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.NotImplementedException class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

The [NotImplementedException](#) exception is thrown when a particular method, get accessor, or set accessor is present as a member of a type but is not implemented.

[NotImplementedException](#) uses the default [Object.Equals](#) implementation, which supports reference equality. For a list of initial values for an instance of [NotImplementedException](#), see the [NotImplementedException](#) constructors.

## Throw the exception

You might choose to throw a [NotImplementedException](#) exception in properties or methods in your own types when the that member is still in development and will only later be implemented in production code. In other words, a [NotImplementedException](#) exception should be synonymous with "still in development."

## Handle the exception

The [NotImplementedException](#) exception indicates that the method or property that you are attempting to invoke has no implementation and therefore provides no functionality. As a result, you should not handle this error in a `try/catch` block. Instead, you should remove the member invocation from your code. You can include a call to the member when it is implemented in the production version of a library.

In some cases, a [NotImplementedException](#) exception may not be used to indicate functionality that is still in development in a pre-production library. However, this still indicates that the functionality is unavailable, and you should remove the member invocation from your code.

## NotImplementedException and other exception types

.NET also includes two other exception types, [NotSupportedException](#) and [PlatformNotSupportedException](#), that indicate that no implementation exists for a

particular member of a type. You should throw one of these instead of a [NotImplementedException](#) exception under the following conditions:

- Throw a [PlatformNotSupportedException](#) exception on platforms on which the functionality is not supported if you've designed a type with one or more members that are available on some platforms or versions but not others.
- Throw a [NotSupportedException](#) exception if the implementation of an interface member or an override to an abstract base class method is not possible.

For example, the [Convert.ToInt32\(DateTime\)](#) method throws a [NotSupportedException](#) exception because no meaningful conversion between a date and time and a 32-bit signed integer exists. The method must be present in this case because the [Convert](#) class implements the [IConvertible](#) interface.

You should also throw a [NotSupportedException](#) exception if you've implemented an abstract base class and add a new member to it that must be overridden by derived classes. In that case, making the member abstract causes existing subclasses to fail to load.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

## [.NET feedback](#)

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.NotSupportedException class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

[NotSupportedException](#) indicates that no implementation exists for an invoked method or property.

[NotSupportedException](#) uses the HRESULT `COR_E_NOTSUPPORTED`, which has the value 0x80131515.

For a list of initial property values for an instance of [NotSupportedException](#), see the [NotSupportedException](#) constructors.

## Throw a NotSupportedException exception

You might consider throwing a [NotSupportedException](#) exception in the following cases:

- You're implementing a general-purpose interface, and number of the methods have no meaningful implementation. For example, if you are creating a date and time type that implements the [IConvertible](#) interface, you would throw a [NotSupportedException](#) exception for most of the conversions.
- You've inherited from an abstract class that requires that you override a number of methods. However, you're only prepared to provide an implementation for a subset of these. For the methods that you decide not to implement, you can choose to throw a [NotSupportedException](#).
- You're defining a general-purpose type with a state that enables operations conditionally. For example, your type can be either read-only or read-write. In that case:
  - If the object is read-only, attempting to assign values to the properties of an instance or call methods that modify instance state should throw a [NotSupportedException](#) exception.
  - You should implement a property that returns a [Boolean](#) value that indicates whether particular functionality is available. For example, for a type that can be either read-only or read-write, you could implement a `IsReadOnly` property that indicates whether the set of read-write methods are available or unavailable.

# Handle a NotSupportedException exception

The [NotSupportedException](#) exception indicates that a method has no implementation and that you should not call it. You should not handle the exception. Instead, what you should do depends on the cause of the exception: whether an implementation is completely absent, or the member invocation is inconsistent with the purpose of an object (such as a call to the [FileStream.Write](#) method on a read-only [FileStream](#) object).

**An implementation has not been provided because the operation cannot be performed in a meaningful way.** This is a common exception when you are calling methods on an object that provides implementations for the methods of an abstract base class, or that implements a general-purpose interface, and the method has no meaningful implementation.

For example, the [Convert](#) class implements the [IConvertible](#) interface, which means that it must include a method to convert every primitive type to every other primitive type. Many of those conversions, however, are not possible. As a result, a call to the [Convert.ToBoolean\(DateTime\)](#) method, for instance, throws a [NotSupportedException](#) exception because there is no possible conversion between a [DateTime](#) and a [Boolean](#) value

To eliminate the exception, you should eliminate the method call.

**The method call is not supported given the state of the object.** You're attempting to invoke a member whose functionality is unavailable because of the object's state. You can eliminate the exception in one of three ways:

- You know the state of the object in advance, but you've invoked an unsupported method or property. In this case, the member invocation is an error, and you can eliminate it.
- You know the state of the object in advance (usually because your code has instantiated it), but the object is mis-configured. The following example illustrates this issue. It creates a read-only [FileStream](#) object and then attempts to write to it.

C#

```
using System;
using System.IO;
using System.Text;
using System.Threading.Tasks;

public class Example
{
    public static async Task Main()
```

```

    {
        Encoding enc = Encoding.Unicode;
        String value = "This is a string to persist.";
        Byte[] bytes = enc.GetBytes(value);

        FileStream fs = new FileStream(@"..\TestFile.dat",
                                         FileMode.Open,
                                         FileAccess.Read);
        Task t = fs.WriteAsync(enc.GetPreamble(), 0,
                               enc.GetPreamble().Length);
        Task t2 = t.ContinueWith((a) => fs.WriteAsync(bytes, 0,
                               bytes.Length));
        await t2;
        fs.Close();
    }
}

// The example displays the following output:
//    Unhandled Exception: System.NotSupportedException: Stream does
//    not support writing.
//        at System.IO.Stream.BeginWriteInternal(Byte[] buffer, Int32
//        offset, Int32 count, AsyncCallback callback, Object state
//        , Boolean serializeAsynchronously)
//        at System.IO.FileStream.BeginWrite(Byte[] array, Int32 offset,
//        Int32 numBytes, AsyncCallback userCallback, Object sta
//        teObject)
//        at System.IO.Stream.<>c.<BeginEndWriteAsync>b__53_0(Stream
//        stream, ReadWriteParameters args, AsyncCallback callback,
//        Object state)
//        at
System.Threading.Tasks.TaskFactory`1.FromAsyncTrim[TInstance, TArgs]
(TInstance thisRef, TArgs args, Func`5 beginMet
//    hod, Func`3 endMethod)
//        at System.IO.Stream.BeginEndWriteAsync(Byte[] buffer, Int32
//        offset, Int32 count)
//        at System.IO.FileStream.WriteAsync(Byte[] buffer, Int32
//        offset, Int32 count, CancellationToken cancellationToken)
//        at System.IO.Stream.WriteAsync(Byte[] buffer, Int32 offset,
//        Int32 count)
//        at Example.Main()

```

You can eliminate the exception by ensuring that the instantiated object supports the functionality you intend. The following example addresses the problem of the read-only [FileStream](#) object by providing the correct arguments to the [FileStream\(Stream, FileMode, FileAccess\)](#) constructor.

- You don't know the state of the object in advance, and the object doesn't support a particular operation. In most cases, the object should include a property or method that indicates whether it supports a particular set of operations. You can eliminate the exception by checking the value of the object and invoking the member only if appropriate.

The following example defines a `DetectEncoding` method that throws a `NotSupportedException` exception when it attempts to read from the beginning of a stream that does not support read access.

C#

```
using System;
using System.IO;
using System.Threading.Tasks;

public class TestPropEx1
{
    public static async Task Main()
    {
        String name = @"\TestFile.dat";
        var fs = new FileStream(name,
                               FileMode.Create,
                               FileAccess.Write);
        Console.WriteLine("Filename: {0}, Encoding: {1}",
                          name, await
        FileUtilities1.GetEncodingType(fs));
    }
}

public class FileUtilities1
{
    public enum EncodingType
    { None = 0, Unknown = -1, Utf8 = 1, Utf16 = 2, Utf32 = 3 }

    public async static Task<EncodingType> GetEncodingType(FileStream
fs)
    {
        Byte[] bytes = new Byte[4];
        int bytesRead = await fs.ReadAsync(bytes, 0, 4);
        if (bytesRead < 2)
            return EncodingType.None;

        if (bytesRead >= 3 & (bytes[0] == 0xEF && bytes[1] == 0xBB &&
bytes[2] == 0xBF))
            return EncodingType.Utf8;

        if (bytesRead == 4)
        {
            var value = BitConverter.ToInt32(bytes, 0);
            if (value == 0x0000FEFF | value == 0xFEFF0000)
                return EncodingType.Utf32;
        }

        var value16 = BitConverter.ToInt16(bytes, 0);
        if (value16 == (ushort)0xFEFF | value16 == (ushort)0xFFFF)
            return EncodingType.Utf16;

        return EncodingType.Unknown;
    }
}
```

```

    }
// The example displays the following output:
//      Unhandled Exception: System.NotSupportedException: Stream does
//      not support reading.
//          at System.IO.FileStream.BeginRead(Byte[] array, Int32 offset,
//      Int32 numBytes, AsyncCallback callback, Object state)
//          at System.IO.Stream.<>c.<BeginEndReadAsync>b__46_0(Stream
//      stream, ReadWriteParameters args, AsyncCallback callback, Object state)
//          at
System.Threading.Tasks.TaskFactory`1.FromAsyncTrim[TInstance, TArgs]
(TInstance thisRef, TArgs args, Func`5 beginMethod, Func`3 endMethod)
//          at System.IO.Stream.BeginEndReadAsync(Byte[] buffer, Int32
offset, Int32 count)
//          at System.IO.FileStream.ReadAsync(Byte[] buffer, Int32 offset,
Int32 count, CancellationToken cancellationToken)
//          at System.IO.Stream.ReadAsync(Byte[] buffer, Int32 offset,
Int32 count)
//          at FileUtilities.GetEncodingType(FileStream fs) in
C:\Work\docs\program.cs:line 26
//          at Example.Main() in C:\Work\docs\program.cs:line 13
//          at Example.<Main>()

```

You can eliminate the exception by examining the value of the [FileStream.CanRead](#) property and exiting the method if the stream is read-only.

C#

```

public static async Task<EncodingType> GetEncodingType(FileStream
fs)
{
    if (!fs.CanRead)
        return EncodingType.Unknown;

    Byte[] bytes = new Byte[4];
    int bytesRead = await fs.ReadAsync(bytes, 0, 4);
    if (bytesRead < 2)
        return EncodingType.None;

    if (bytesRead >= 3 & (bytes[0] == 0xEF && bytes[1] == 0xBB &&
bytes[2] == 0xBF))
        return EncodingType.Utf8;

    if (bytesRead == 4)
    {
        var value = BitConverter.ToInt32(bytes, 0);
        if (value == 0x0000FEFF | value == 0xFEFF0000)
            return EncodingType.Utf32;
    }

    var value16 = BitConverter.ToInt16(bytes, 0);
    if (value16 == (ushort)0xFEFF | value16 == (ushort)0xFFFF)
        return EncodingType.Utf16;

```

```
        return EncodingType.Unknown;
    }
}
// The example displays the following output:
//     Filename: .\TestFile.dat, Encoding: Unknown
```

## Related exception types

The [NotSupportedException](#) exception is closely related to two other exception types;

- [NotImplementedException](#)

This exception is thrown when a method could be implemented but is not, either because the member will be implemented in a later version, the member is not available on a particular platform, or the member belongs to an abstract class and a derived class must provide an implementation.

- [InvalidOperationException](#)

This exception is thrown in scenarios in which it is generally sometimes possible for the object to perform the requested operation, and the object state determines whether the operation can be performed.

## .NET Compact Framework notes

When working with the .NET Compact Framework and using P/Invoke on a native function, this exception may be thrown if:

- The declaration in managed code is incorrect.
- The .NET Compact Framework does not support what you are trying to do.
- The DLL names are mangled on export.

If a [NotSupportedException](#) exception is thrown, check:

- For any violations of the .NET Compact Framework P/Invoke restrictions.
- For any arguments that require pre-allocated memory. If these exist, you should pass a reference to an existing variable.
- That the names of the exported functions are correct. This can be verified with [DumpBin.exe](#).
- That you are not attempting to pass too many arguments.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.TypeInitializationException class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

When a class initializer fails to initialize a type, a [TypeInitializationException](#) is created and passed a reference to the exception thrown by the type's class initializer. The [InnerException](#) property of [TypeInitializationException](#) holds the underlying exception.

Typically, the [TypeInitializationException](#) exception reflects a catastrophic condition (the runtime is unable to instantiate a type) that prevents an application from continuing. Most commonly, the [TypeInitializationException](#) is thrown in response to some change in the executing environment of the application. Consequently, other than possibly for troubleshooting debug code, the exception should not be handled in a `try/catch` block. Instead, the cause of the exception should be investigated and eliminated.

[TypeInitializationException](#) uses the HRESULT `COR_E_TYPEINITIALIZATION`, which has the value 0x80131534.

For a list of initial property values for an instance of [TypeInitializationException](#), see the [TypeInitializationException](#) constructors.

The following sections describe some of the situations in which a [TypeInitializationException](#) exception is thrown.

## Static constructors

A static constructor, if one exists, is called automatically by the runtime before creating a new instance of a type. Static constructors can be explicitly defined by a developer. If a static constructor is not explicitly defined, compilers automatically create one to initialize any `static` (in C# or F#) or `Shared` (in Visual Basic) members of the type. For more information on static constructors, see [Static Constructors](#).

Most commonly, a [TypeInitializationException](#) exception is thrown when a static constructor is unable to instantiate a type. The [InnerException](#) property indicates why the static constructor was unable to instantiate the type. Some of the more common causes of a [TypeInitializationException](#) exception are:

- An unhandled exception in a static constructor

If an exception is thrown in a static constructor, that exception is wrapped in a [TypeInitializationException](#) exception, and the type cannot be instantiated.

What often makes this exception difficult to troubleshoot is that static constructors are not always explicitly defined in source code. A static constructor exists in a type if:

- It has been explicitly defined as a member of a type.
- The type has `static` (in C# or F#) or `Shared` (in Visual Basic) variables that are declared and initialized in a single statement. In this case, the language compiler generates a static constructor for the type. You can inspect it by using a utility such as [IL Disassembler](#). For instance, when the C# and VB compilers compile the following example, they generate the IL for a static constructor that is similar to this:

```
il

.method private specialname rtspecialname static
    void .cctor() cil managed
{
    // Code size      12 (0xc)
    .maxstack 8
    IL_0000: ldc.i4.3
    IL_0001: newobj    instance void TestClass::ctor(int32)
    IL_0006: stsfld    class TestClass Example::test
    IL_000b: ret
} // end of method Example::cctor
```

The following example shows a [TypeInitializationException](#) exception thrown by a compiler-generated static constructor. The `Example` class includes a `static` (in C#) or `Shared` (in Visual Basic) field of type `TestClass` that is instantiated by passing a value of 3 to its class constructor. That value, however, is illegal; only values of 0 or 1 are permitted. As a result, the `TestClass` class constructor throws an [ArgumentOutOfRangeException](#). Since this exception is not handled, it is wrapped in a [TypeInitializationException](#) exception.

```
C#

using System;

public class Example
{
    private static TestClass test = new TestClass(3);

    public static void Main()
    {
```

```

        Example ex = new Example();
        Console.WriteLine(test.Value);
    }

}

public class TestClass
{
    public readonly int Value;

    public TestClass(int value)
    {
        if (value < 0 || value > 1) throw new
ArgumentOutOfRangeException(nameof(value));
        Value = value;
    }
}

// The example displays the following output:
//     Unhandled Exception: System.TypeInitializationException:
//         The type initializer for 'Example' threw an exception. --->
//     System.ArgumentOutOfRangeException: Specified argument was out
of the range of valid values.
//         at TestClass..ctor(Int32 value)
//         at Example..cctor()
//         --- End of inner exception stack trace ---
//         at Example.Main()

```

Note that the exception message displays information about the [InnerException](#) property.

- A missing assembly or data file

A common cause of a [TypeInitializationException](#) exception is that an assembly or data file that was present in an application's development and test environments is missing from its runtime environment. For example, you can compile the following example to an assembly named Missing1a.dll by using this command-line syntax:

C#

```
csc -t:library Missing1a.cs
```

C#

```

using System;

public class InfoModule
{
    private DateTime firstUse;
    private int ctr = 0;

    public InfoModule(DateTime dat)

```

```
        firstUse = dat;
    }

    public int Increment()
    {
        return ++ctr;
    }

    public DateTime GetInitializationTime()
    {
        return firstUse;
    }
}
```

You can then compile the following example to an executable named Missing1.exe by including a reference to Missing1a.dll:

C#

```
csc Missing1.cs /r:Missing1a.dll
```

However, if you rename, move, or delete Missing1a.dll and run the example, it throws a [TypeInitializationException](#) exception and displays the output shown in the example. Note that the exception message includes information about the [InnerException](#) property. In this case, the inner exception is a [FileNotFoundException](#) that is thrown because the runtime cannot find the dependent assembly.

C#

```
using System;

public class MissingEx1
{
    public static void Main()
    {
        Person p = new Person("John", "Doe");
        Console.WriteLine(p);
    }
}

public class Person
{
    static readonly InfoModule s_infoModule;

    readonly string _fName;
    readonly string _lName;

    static Person()
```

```

    {
        s_infoModule = new InfoModule(DateTime.UtcNow);
    }

    public Person(string fName, string lName)
    {
        _fName = fName;
        _lName = lName;
        s_infoModule.Increment();
    }

    public override string ToString()
    {
        return string.Format("{0} {1}", _fName, _lName);
    }
}

// The example displays the following output if missing1a.dll is
// renamed or removed:
//    Unhandled Exception: System.TypeInitializationException:
//        The type initializer for 'Person' threw an exception. --->
//        System.IO.FileNotFoundException: Could not load file or
//        assembly
//            'Missing1a, Version=0.0.0.0, Culture=neutral,
//            PublicKeyToken=null'
//            or one of its dependencies. The system cannot find the file
//            specified.
//        at Person..cctor()
//        --- End of inner exception stack trace ---
//        at Person..ctor(String fName, String lName)
//        at Example.Main()

```

### ⓘ Note

In this example, a `TypeInitializationException` exception was thrown because an assembly could not be loaded. The exception can also be thrown if a static constructor attempts to open a data file, such as a configuration file, an XML file, or a file containing serialized data, that it cannot find.

## Regular expression match timeout values

You can set the default timeout value for a regular expression pattern matching operation on a per-application domain basis. The timeout is defined by specifying a `TimeSpan` value for the "REGEX\_DEFAULT\_MATCH\_TIMEOUT" property to the `AppDomain.SetData` method. The time interval must be a valid `TimeSpan` object that is greater than zero and less than approximately 24 days. If these requirements are not met, the attempt to set the default timeout value throws an

[ArgumentOutOfRangeException](#), which in turn is wrapped in a [TypeInitializationException](#) exception.

The following example shows the [TypeInitializationException](#) that is thrown when the value assigned to the "REGEX\_DEFAULT\_MATCH\_TIMEOUT" property is invalid. To eliminate the exception, set the "REGEX\_DEFAULT\_MATCH\_TIMEOUT" property to a [TimeSpan](#) value that is greater than zero and less than approximately 24 days.

C#

```
using System;
using System.Text.RegularExpressions;

public class RegexEx1
{
    public static void Main()
    {
        AppDomain domain = AppDomain.CurrentDomain;
        // Set a timeout interval of -2 seconds.
        domain.SetData("REGEX_DEFAULT_MATCH_TIMEOUT",
TimeSpan.FromSeconds(-2));

        Regex rgx = new Regex("[aeiou]");
        Console.WriteLine("Regular expression pattern: {0}",
rgx.ToString());
        Console.WriteLine("Timeout interval for this regex: {0} seconds",
rgx.MatchTimeout.TotalSeconds);
    }
}
// The example displays the following output:
//     Unhandled Exception: System.TypeInitializationException:
//         The type initializer for 'System.Text.RegularExpressions.Regex'
threw an exception. --->
//         System.ArgumentOutOfRangeException: Specified argument was out of
the range of valid values.
//         Parameter name: AppDomain data 'REGEX_DEFAULT_MATCH_TIMEOUT'
contains an invalid value or
//         object for specifying a default matching timeout for
System.Text.RegularExpressions.Regex.
//         at System.Text.RegularExpressions.Regex.InitDefaultMatchTimeout()
//         at System.Text.RegularExpressions.Regex..cctor()
//         --- End of inner exception stack trace ---
//         at System.Text.RegularExpressions.Regex..ctor(String pattern)
//         at Example.Main()
```

## Calendars and cultural data

If you attempt to instantiate a calendar but the runtime is unable to instantiate the [CultureInfo](#) object that corresponds to that calendar, it throws a

`TypeInitializationException` exception. This exception can be thrown by the following calendar class constructors:

- The parameterless constructor of the [JapaneseCalendar](#) class.
- The parameterless constructor of the [KoreanCalendar](#) class.
- The parameterless constructor of the [TaiwanCalendar](#) class.

Since cultural data for these cultures should be available on all systems, you should rarely, if ever, encounter this exception.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

**.NET feedback**

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Numerics in .NET

Article • 09/01/2023

.NET provides a range of numeric integer and floating-point primitives, as well as:

- [System.Half](#), which represents a half-precision floating-point number.
- [System.Decimal](#), which represents a decimal floating-point number.
- [System.Numerics.BigInteger](#), which is an integral type with no theoretical upper or lower bound.
- [System.Numerics.Complex](#), which represents complex numbers.
- A set of SIMD-enabled types in the [System.Numerics](#) namespace.

## Integer types

.NET supports both signed and unsigned 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit integer types, which are listed in the following tables.

### Signed integer types

Expand table

Type	Size (in bytes)	Minimum value	Maximum value
<a href="#">System.Int16</a>	2	-32,768	32,767
<a href="#">System.Int32</a>	4	-2,147,483,648	2,147,483,647
<a href="#">System.Int64</a>	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
<a href="#">System.Int128</a>	16	-170,141,183,460,469,231,731,687,303,715,884,105,728	170,141,183,460,469,231,731,687,303,715,884,105,727
<a href="#">System.SByte</a>	1	-128	127
<a href="#">System.IntPtr</a> (in 32-bit process)	4	-2,147,483,648	2,147,483,647
<a href="#">System.IntPtr</a> (in 64-bit process)	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

### Unsigned integer types

Expand table

Type	Size (in bytes)	Minimum value	Maximum value
<a href="#">System.Byte</a>	1	0	255
<a href="#">System.UInt16</a>	2	0	65,535
<a href="#">System.UInt32</a>	4	0	4,294,967,295
<a href="#">System.UInt64</a>	8	0	18,446,744,073,709,551,615
<a href="#">System.UInt128</a>	16	0	340,282,366,920,938,463,463,374,607,431,768,211,455
<a href="#">System.UIntPtr</a> (in 32-bit process)	4	0	4,294,967,295

Type	Size (in bytes)	Minimum value	Maximum value
<a href="#">System.UIntPtr</a> (in 64-bit process)	8	0	18,446,744,073,709,551,615

Each integer type supports a set of standard arithmetic operators. The [System.Math](#) class provides methods for a broader set of mathematical functions.

You can also work with the individual bits in an integer value by using the [System.BitConverter](#) class.

 **Note**

The unsigned integer types are not CLS-compliant. For more information, see [Language independence and language-independent components](#).

## BigInteger

The [System.Numerics.BigInteger](#) structure is an immutable type that represents an arbitrarily large integer whose value in theory has no upper or lower bounds. The methods of the [BigInteger](#) type closely parallel those of the other integral types.

## Floating-point types

.NET includes the following floating-point types:

 [Expand table](#)

Type	Size (in bytes)	Approximate range	Primitive?	Notes
<a href="#">System.Half</a>	2	$\pm 65504$	No	Introduced in .NET 5
<a href="#">System.Single</a>	4	$\pm 3.4 \times 10^{38}$	Yes	
<a href="#">System.Double</a>	8	$\pm 1.7 \times 10^{308}$	Yes	
<a href="#">System.Decimal</a>	16	$\pm 7.9228 \times 10^{28}$	No	

The [Half](#), [Single](#), and [Double](#) types support special values that represent not-a-number and infinity. For example, the [Double](#) type provides the following values: [Double.NaN](#), [Double.NegativeInfinity](#), and [Double.PositiveInfinity](#). You use the [Double.IsNaN](#), [Double.IsInfinity](#), [Double.IsPositiveInfinity](#), and [Double.IsNegativeInfinity](#) methods to test for these special values.

Each floating-point type supports a set of standard arithmetic operators. The [System.Math](#) class provides methods for a broader set of mathematical functions. .NET Core 2.0 and later includes the [System.MathF](#) class, which provides methods that accept arguments of the [Single](#) type.

You can also work with the individual bits in [Double](#), [Single](#), and [Half](#) values by using the [System.BitConverter](#) class. The [System.Decimal](#) structure has its own methods, [Decimal.GetBits](#) and [Decimal\(Int32\[\]\)](#), for working with a decimal value's individual bits, as well as its own set of methods for performing some additional mathematical operations.

The [Double](#), [Single](#), and [Half](#) types are intended to be used for values that, by their nature, are imprecise (for example, the distance between two stars) and for applications in which a high degree of precision and small rounding error is not required. Use the [System.Decimal](#) type for cases in which greater precision is required and rounding errors should be minimized.

## ⓘ Note

The `Decimal` type doesn't eliminate the need for rounding. Rather, it minimizes errors due to rounding.

## Complex

The `System.Numerics.Complex` structure represents a complex number, that is, a number with a real number part and an imaginary number part. It supports a standard set of arithmetic, comparison, equality, explicit and implicit conversion operators, as well as mathematical, algebraic, and trigonometric methods.

## SIMD-enabled types

The `System.Numerics` namespace includes a set of .NET SIMD-enabled types. SIMD (Single Instruction Multiple Data) operations can be parallelized at the hardware level. That increases the throughput of the vectorized computations, which are common in mathematical, scientific, and graphics apps.

The .NET SIMD-enabled types include the following:

- The `Vector2`, `Vector3`, and `Vector4` types, which represent vectors with 2, 3, and 4 `Single` values.
- Two matrix types, `Matrix3x2`, which represents a 3x2 matrix, and `Matrix4x4`, which represents a 4x4 matrix.
- The `Plane` type, which represents a plane in three-dimensional space.
- The `Quaternion` type, which represents a vector that is used to encode three-dimensional physical rotations.
- The `Vector<T>` type, which represents a vector of a specified numeric type and provides a broad set of operators that benefit from SIMD support. The count of a `Vector<T>` instance is fixed, but its value `Vector<T>.Count` depends on the CPU of the machine, on which code is executed.

## ⓘ Note

The `Vector<T>` type is included with .NET Core and .NET 5+, but not .NET Framework. If you're using .NET Framework, install the `System.Numerics.Vectors` NuGet package to get access to this type.

The SIMD-enabled types are implemented in such a way that they can be used with non-SIMD-enabled hardware or JIT compilers. To take advantage of SIMD instructions, your 64-bit apps must be run by the runtime that uses the RyuJIT compiler, which is included in .NET Core and in .NET Framework 4.6 and later versions. It adds SIMD support when targeting 64-bit processors.

For more information, see [Use SIMD-accelerated numeric types](#).

## See also

- [Standard numeric format strings](#)
- [Floating-point numeric types in C#](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can

 .NET

[.NET feedback](#)

.NET is an open source project. Select a link to provide feedback:

also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Boolean struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

A [Boolean](#) instance can have either of two values: `true` or `false`.

The [Boolean](#) structure provides methods that support the following tasks:

- Converting Boolean values to strings: [ToString](#)
- Parsing strings to convert them to Boolean values: [Parse](#) and [TryParse](#)
- Comparing values: [CompareTo](#) and [Equals](#)

This article explains these tasks and other usage details.

## Format Boolean values

The string representation of a [Boolean](#) is either "True" for a `true` value or "False" for a `false` value. The string representation of a [Boolean](#) value is defined by the read-only [TrueString](#) and [FalseString](#) fields.

You use the [ToString](#) method to convert Boolean values to strings. The Boolean structure includes two [ToString](#) overloads: the parameterless [ToString\(\)](#) method and the [ToString\(IFormatProvider\)](#) method, which includes a parameter that controls formatting. However, because this parameter is ignored, the two overloads produce identical strings. The [ToString\(IFormatProvider\)](#) method does not support culture-sensitive formatting.

The following example illustrates formatting with the [ToString](#) method. Note that the C# and VB examples use the [composite formatting](#) feature, while the F# example uses [string interpolation](#). In both cases the [ToString](#) method is called implicitly.

C#

```
using System;

public class Example10
{
    public static void Main()
    {
        bool raining = false;
        bool busLate = true;
```

```

        Console.WriteLine("It is raining: {0}", raining);
        Console.WriteLine("The bus is late: {0}", busLate);
    }
}
// The example displays the following output:
//      It is raining: False
//      The bus is late: True

```

Because the [Boolean](#) structure can have only two values, it is easy to add custom formatting. For simple custom formatting in which other string literals are substituted for "True" and "False", you can use any conditional evaluation feature supported by your language, such as the [conditional operator](#) in C# or the [If operator](#) in Visual Basic. The following example uses this technique to format [Boolean](#) values as "Yes" and "No" rather than "True" and "False".

C#

```

using System;

public class Example11
{
    public static void Main()
    {
        bool raining = false;
        bool busLate = true;

        Console.WriteLine("It is raining: {0}",
                          raining ? "Yes" : "No");
        Console.WriteLine("The bus is late: {0}",
                          busLate ? "Yes" : "No");
    }
}
// The example displays the following output:
//      It is raining: No
//      The bus is late: Yes

```

For more complex custom formatting operations, including culture-sensitive formatting, you can call the [String.Format\(IFormatProvider, String, Object\[\]\)](#) method and provide an [ICustomFormatter](#) implementation. The following example implements the [ICustomFormatter](#) and [IFormatProvider](#) interfaces to provide culture-sensitive Boolean strings for the English (United States), French (France), and Russian (Russia) cultures.

C#

```

using System;
using System.Globalization;

public class Example4

```

```
{  
    public static void Main()  
    {  
        String[] cultureNames = { "", "en-US", "fr-FR", "ru-RU" };  
        foreach (var cultureName in cultureNames) {  
            bool value = true;  
            CultureInfo culture =  
CultureInfo.CreateSpecificCulture(cultureName);  
            BooleanFormatter formatter = new BooleanFormatter(culture);  
  
                string result = string.Format(formatter, "Value for '{0}': {1}",  
culture.Name, value);  
                Console.WriteLine(result);  
        }  
    }  
}  
  
public class BooleanFormatter : ICustomFormatter, IFormatProvider  
{  
    private CultureInfo culture;  
  
    public BooleanFormatter() : this(CultureInfo.CurrentCulture)  
    { }  
  
    public BooleanFormatter(CultureInfo culture)  
    {  
        this.culture = culture;  
    }  
  
    public Object GetFormat(Type formatType)  
    {  
        if (formatType == typeof(ICustomFormatter))  
            return this;  
        else  
            return null;  
    }  
  
    public string Format(string fmt, Object arg, IFormatProvider  
formatProvider)  
    {  
        // Exit if another format provider is used.  
        if (! formatProvider.Equals(this)) return null;  
  
        // Exit if the type to be formatted is not a Boolean  
        if (! (arg is Boolean)) return null;  
  
        bool value = (bool) arg;  
        switch (culture.Name) {  
            case "en-US":  
                return value.ToString();  
            case "fr-FR":  
                if (value)  
                    return "vrai";  
                else  
                    return "faux";  
        }  
    }  
}
```

```

        case "ru-RU":
            if (value)
                return "верно";
            else
                return "неверно";
        default:
            return value.ToString();
    }
}
// The example displays the following output:
//      Value for '': True
//      Value for 'en-US': True
//      Value for 'fr-FR': vrai
//      Value for 'ru-RU': верно

```

Optionally, you can use [resource files](#) to define culture-specific Boolean strings.

## Convert to and from Boolean values

The [Boolean](#) structure implements the [IConvertible](#) interface. As a result, you can use the [Convert](#) class to perform conversions between a [Boolean](#) value and any other primitive type in .NET, or you can call the [Boolean](#) structure's explicit implementations. However, conversions between a [Boolean](#) and the following types are not supported, so the corresponding conversion methods throw an [InvalidOperationException](#) exception:

- Conversion between [Boolean](#) and [Char](#) (the [Convert.ToBoolean\(Char\)](#) and [Convert.ToChar\(Boolean\)](#) methods).
- Conversion between [Boolean](#) and [DateTime](#) (the [Convert.ToBoolean\(DateTime\)](#) and [Convert.ToDateTime\(Boolean\)](#) methods).

All conversions from integral or floating-point numbers to Boolean values convert non-zero values to `true` and zero values to `false`. The following example illustrates this by calling selected overloads of the [Convert.ToBoolean](#) class.

C#

```

using System;

public class Example2
{
    public static void Main()
    {
        Byte byteValue = 12;
        Console.WriteLine(Convert.ToBoolean(byteValue));
        Byte byteValue2 = 0;
        Console.WriteLine(Convert.ToBoolean(byteValue2));
    }
}

```

```

        int intValue = -16345;
        Console.WriteLine(Convert.ToBoolean(intValue));
        long longValue = 945;
        Console.WriteLine(Convert.ToBoolean(longValue));
        SByte sbyteValue = -12;
        Console.WriteLine(Convert.ToBoolean(sbyteValue));
        double dblValue = 0;
        Console.WriteLine(Convert.ToBoolean(dblValue));
        float sngValue = .0001f;
        Console.WriteLine(Convert.ToBoolean(sngValue));
    }
}

// The example displays the following output:
//      True
//      False
//      True
//      True
//      True
//      False
//      True

```

When converting from Boolean to numeric values, the conversion methods of the [Convert](#) class convert `true` to 1 and `false` to 0. However, Visual Basic conversion functions convert `true` to either 255 (for conversions to [Byte](#) values) or -1 (for all other numeric conversions). The following example converts `true` to numeric values by using a [Convert](#) method, and, in the case of the Visual Basic example, by using the Visual Basic language's own conversion operator.

C#

```

using System;

public class Example3
{
    public static void Main()
    {
        bool flag = true;

        byte byteValue;
        byteValue = Convert.ToByte(flag);
        Console.WriteLine("{0} -> {1}", flag, byteValue);

        sbyte sbyteValue;
        sbyteValue = Convert.ToSByte(flag);
        Console.WriteLine("{0} -> {1}", flag, sbyteValue);

        double dblValue;
        dblValue = Convert.ToDouble(flag);
        Console.WriteLine("{0} -> {1}", flag, dblValue);

        int intValue;

```

```
        intValue = Convert.ToInt32(flag);
        Console.WriteLine("{0} -> {1}", flag, intValue);
    }
}
// The example displays the following output:
//      True -> 1
//      True -> 1
//      True -> 1
//      True -> 1
```

For conversions from [Boolean](#) to string values, see the [Format Boolean values](#) section. For conversions from strings to [Boolean](#) values, see the [Parse Boolean values](#) section.

## Parse Boolean values

The [Boolean](#) structure includes two static parsing methods, [Parse](#) and [TryParse](#), that convert a string to a Boolean value. The string representation of a Boolean value is defined by the case-insensitive equivalents of the values of the [TrueString](#) and [FalseString](#) fields, which are "True" and "False", respectively. In other words, the only strings that parse successfully are "True", "False", "true", "false", or some mixed-case equivalent. You cannot successfully parse numeric strings such as "0" or "1". Leading or trailing white-space characters are not considered when performing the string comparison.

The following example uses the [Parse](#) and [TryParse](#) methods to parse a number of strings. Note that only the case-insensitive equivalents of "True" and "False" can be successfully parsed.

C#

```
using System;

public class Example7
{
    public static void Main()
    {
        string[] values = { null, String.Empty, "True", "False",
                           "true", "false", "    true    ",
                           "TrUe", "fAlSe", "fa lse", "0",
                           "1", "-1", "string" };
        // Parse strings using the Boolean.Parse method.
        foreach (var value in values) {
            try {
                bool flag = Boolean.Parse(value);
                Console.WriteLine("'{}' --> {}", value, flag);
            }
            catch (ArgumentException) {
                Console.WriteLine("Cannot parse a null string.");
            }
        }
    }
}
```

```

        }
        catch (FormatException) {
            Console.WriteLine("Cannot parse '{0}'.", value);
        }
    }
    Console.WriteLine();
    // Parse strings using the Boolean.TryParse method.
    foreach (var value in values) {
        bool flag = false;
        if (Boolean.TryParse(value, out flag))
            Console.WriteLine('{0}' --> {1}, value, flag);
        else
            Console.WriteLine("Unable to parse '{0}'", value);
    }
}
// The example displays the following output:
//      Cannot parse a null string.
//      Cannot parse ''.
//      'True' --> True
//      'False' --> False
//      'true' --> True
//      'false' --> False
//      '    true    ' --> True
//      'TrUe' --> True
//      'fAlSe' --> False
//      Cannot parse 'fa lse'.
//      Cannot parse '0'.
//      Cannot parse '1'.
//      Cannot parse '-1'.
//      Cannot parse 'string'.

//      Unable to parse ''
//      Unable to parse ''
//      'True' --> True
//      'False' --> False
//      'true' --> True
//      'false' --> False
//      '    true    ' --> True
//      'TrUe' --> True
//      'fAlSe' --> False
//      Cannot parse 'fa lse'.
//      Unable to parse '0'
//      Unable to parse '1'
//      Unable to parse '-1'
//      Unable to parse 'string'

```

If you're programming in Visual Basic, you can use the `CBool` function to convert the string representation of a number to a Boolean value. "0" is converted to `false`, and the string representation of any non-zero value is converted to `true`. If you're not programming in Visual Basic, you must convert your numeric string to a number before

converting it to a Boolean. The following example illustrates this by converting an array of integers to Boolean values.

```
C#  
  
using System;  
  
public class Example8  
{  
    public static void Main()  
    {  
        String[] values = { "09", "12.6", "0", "-13 " };  
        foreach (var value in values) {  
            bool success, result;  
            int number;  
            success = Int32.TryParse(value, out number);  
            if (success) {  
                // The method throws no exceptions.  
                result = Convert.ToBoolean(number);  
                Console.WriteLine("Converted '{0}' to {1}", value, result);  
            }  
            else {  
                Console.WriteLine("Unable to convert '{0}'", value);  
            }  
        }  
    }  
}  
  
// The example displays the following output:  
//     Converted '09' to True  
//     Unable to convert '12.6'  
//     Converted '0' to False  
//     Converted '-13 ' to True
```

## Compare Boolean values

Because Boolean values are either `true` or `false`, there is little reason to explicitly call the [CompareTo](#) method, which indicates whether an instance is greater than, less than, or equal to a specified value. Typically, to compare two Boolean variables, you call the [Equals](#) method or use your language's equality operator.

However, when you want to compare a Boolean variable with the literal Boolean value `true` or `false`, it's not necessary to do an explicit comparison, because the result of evaluating a Boolean value is that Boolean value. For example, the following two expressions are equivalent, but the second is more compact. However, both techniques offer comparable performance.

```
C#
```

```
if (booleanValue == true) {
```

C#

```
if (booleanValue) {
```

## Work with Booleans as binary values

A Boolean value occupies one byte of memory, as the following example shows. The C# example must be compiled with the `/unsafe` switch.

C#

```
using System;

public struct BoolStruct
{
    public bool flag1;
    public bool flag2;
    public bool flag3;
    public bool flag4;
    public bool flag5;
}

public class Example9
{
    public static void Main()
    {
        unsafe {
            BoolStruct b = new BoolStruct();
            bool* addr = (bool*) &b;
            Console.WriteLine("Size of BoolStruct: {0}", sizeof(BoolStruct));
            Console.WriteLine("Field offsets:");
            Console.WriteLine("    flag1: {0}", (bool*) &b.flag1 - addr);
            Console.WriteLine("    flag1: {0}", (bool*) &b.flag2 - addr);
            Console.WriteLine("    flag1: {0}", (bool*) &b.flag3 - addr);
            Console.WriteLine("    flag1: {0}", (bool*) &b.flag4 - addr);
            Console.WriteLine("    flag1: {0}", (bool*) &b.flag5 - addr);
        }
    }
}

// The example displays the following output:
//      Size of BoolStruct: 5
//      Field offsets:
//          flag1: 0
//          flag1: 1
//          flag1: 2
```

```
//          flag1: 3
//          flag1: 4
```

The byte's low-order bit is used to represent its value. A value of 1 represents `true`; a value of 0 represents `false`.

### Tip

You can use the `System.Collections.Specialized.BitVector32` structure to work with sets of Boolean values.

You can convert a Boolean value to its binary representation by calling the `BitConverter.GetBytes(Boolean)` method. The method returns a byte array with a single element. To restore a Boolean value from its binary representation, you can call the `BitConverter.ToBoolean(Byte[], Int32)` method.

The following example calls the `BitConverter.GetBytes` method to convert a Boolean value to its binary representation and displays the individual bits of the value, and then calls the `BitConverter.ToBoolean` method to restore the value from its binary representation.

C#

```
using System;

public class Example1
{
    public static void Main()
    {
        bool[] flags = { true, false };
        foreach (var flag in flags)
        {
            // Get binary representation of flag.
            Byte value = BitConverter.GetBytes(flag)[0];
            Console.WriteLine("Original value: {0}", flag);
            Console.WriteLine("Binary value: {0} ({1})", value,
                GetBinaryString(value));
            // Restore the flag from its binary representation.
            bool newFlag = BitConverter.ToBoolean(new Byte[] { value }, 0);
            Console.WriteLine("Restored value: {0}\n", flag);
        }
    }

    private static string GetBinaryString(Byte value)
    {
        string retVal = Convert.ToString(value, 2);
        return new string('0', 8 - retVal.Length) + retVal;
    }
}
```

```
}

// The example displays the following output:
//      Original value: True
//      Binary value:  1 (00000001)
//      Restored value: True
//
//      Original value: False
//      Binary value:  0 (00000000)
//      Restored value: False
```

## Perform operations with Boolean values

This section illustrates how Boolean values are used in apps. The first section discusses its use as a flag. The second illustrates its use for arithmetic operations.

### Boolean values as flags

Boolean variables are most commonly used as flags, to signal the presence or absence of some condition. For example, in the [String.Compare\(String, String, Boolean\)](#) method, the final parameter, `ignoreCase`, is a flag that indicates whether the comparison of two strings is case-insensitive (`ignoreCase` is `true`) or case-sensitive (`ignoreCase` is `false`). The value of the flag can then be evaluated in a conditional statement.

The following example uses a simple console app to illustrate the use of Boolean variables as flags. The app accepts command-line parameters that enable output to be redirected to a specified file (the `/f` switch), and that enable output to be sent both to a specified file and to the console (the `/b` switch). The app defines a flag named `isRedirected` to indicate whether output is to be sent to a file, and a flag named `isBoth` to indicate that output should be sent to the console. The F# example uses a [recursive function](#) to parse the arguments.

C#

```
using System;
using System.IO;
using System.Threading;

public class Example5
{
    public static void Main()
    {
        // Initialize flag variables.
        bool isRedirected = false;
        bool isBoth = false;
        String fileName = "";
```

```

StreamWriter sw = null;

// Get any command line arguments.
String[] args = Environment.GetCommandLineArgs();
// Handle any arguments.
if (args.Length > 1) {
    for (int ctr = 1; ctr < args.Length; ctr++) {
        String arg = args[ctr];
        if (arg.StartsWith("/") || arg.StartsWith("-")) {
            switch (arg.Substring(1).ToLower())
            {
                case "f":
                    isRedirected = true;
                    if (args.Length < ctr + 2) {
                        ShowSyntax("The /f switch must be followed by a
filename.");
                        return;
                    }
                    fileName = args[ctr + 1];
                    ctr++;
                    break;
                case "b":
                    isBoth = true;
                    break;
                default:
                    ShowSyntax(String.Format("The {0} switch is not
supported",
args[ctr]));
                    return;
            }
        }
    }
}

// If isBoth is True, isRedirected must be True.
if (isBoth && !isRedirected) {
    ShowSyntax("The /f switch must be used if /b is used.");
    return;
}

// Handle output.
if (isRedirected) {
    sw = new StreamWriter(fileName);
    if (!isBoth)
        Console.SetOut(sw);
}
String msg = String.Format("Application began at {0}", DateTime.Now);
Console.WriteLine(msg);
if (isBoth) sw.WriteLine(msg);
Thread.Sleep(5000);
msg = String.Format("Application ended normally at {0}",
DateTime.Now);
Console.WriteLine(msg);
if (isBoth) sw.WriteLine(msg);
if (isRedirected) sw.Close();

```

```
}

private static void ShowSyntax(String errMsg)
{
    Console.WriteLine(errMsg);
    Console.WriteLine("\nSyntax: Example [[/f <filename> [/b]]\n");
}

}
```

## Booleans and arithmetic operations

A Boolean value is sometimes used to indicate the presence of a condition that triggers a mathematical calculation. For example, a `hasShippingCharge` variable might serve as a flag to indicate whether to add shipping charges to an invoice amount.

Because an operation with a `false` value has no effect on the result of an operation, it's not necessary to convert the Boolean to an integral value to use in the mathematical operation. Instead, you can use conditional logic.

The following example computes an amount that consists of a subtotal, a shipping charge, and an optional service charge. The `hasServiceCharge` variable determines whether the service charge is applied. Instead of converting `hasServiceCharge` to a numeric value and multiplying it by the amount of the service charge, the example uses conditional logic to add the service charge amount if it is applicable.

C#

```
using System;

public class Example6
{
    public static void Main()
    {
        bool[] hasServiceCharges = { true, false };
        Decimal subtotal = 120.62m;
        Decimal shippingCharge = 2.50m;
        Decimal serviceCharge = 5.00m;

        foreach (var hasServiceCharge in hasServiceCharges) {
            Decimal total = subtotal + shippingCharge +
                           (hasServiceCharge ? serviceCharge : 0);
            Console.WriteLine("hasServiceCharge = {1}: The total is {0:C2}.",
                             total, hasServiceCharge);
        }
    }
}

// The example displays output like the following:
```

```
//      hasServiceCharge = True: The total is $128.12.  
//      hasServiceCharge = False: The total is $123.12.
```

## Booleans and interop

While marshaling base data types to COM is generally straightforward, the `Boolean` data type is an exception. You can apply the `MarshalAsAttribute` attribute to marshal the `Boolean` type to any of the following representations:

[Expand table](#)

Enumeration type	Unmanaged format
<code>UnmanagedType.Bool</code>	A 4-byte integer value, where any nonzero value represents <code>true</code> and 0 represents <code>false</code> . This is the default format of a <code>Boolean</code> field in a structure and of a <code>Boolean</code> parameter in platform invoke calls.
<code>UnmanagedType.U1</code>	A 1-byte integer value, where the 1 represents <code>true</code> and 0 represents <code>false</code> .
<code>UnmanagedType.VariantBool</code>	A 2-byte integer value, where -1 represents <code>true</code> and 0 represents <code>false</code> . This is the default format of a <code>Boolean</code> parameter in COM interop calls.

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Byte struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

[Byte](#) is an immutable value type that represents unsigned integers with values that range from 0 (which is represented by the [Byte.MinValue](#) constant) to 255 (which is represented by the [Byte.MaxValue](#) constant). .NET also includes a signed 8-bit integer value type, [SByte](#), which represents values that range from -128 to 127.

## Instantiate a Byte value

You can instantiate a [Byte](#) value in several ways:

- You can declare a [Byte](#) variable and assign it a literal integer value that is within the range of the [Byte](#) data type. The following example declares two [Byte](#) variables and assigns them values in this way.

```
C#
```

```
byte value1 = 64;
byte value2 = 255;
```

- You can assign a non-byte numeric value to a byte. This is a narrowing conversion, so it requires a cast operator in C# and F#, or a conversion method in Visual Basic if `Option Strict` is on. If the non-byte value is a [Single](#), [Double](#), or [Decimal](#) value that includes a fractional component, the handling of its fractional part depends on the compiler performing the conversion. The following example assigns several numeric values to [Byte](#) variables.

```
C#
```

```
int int1 = 128;
try
{
    byte value1 = (byte)int1;
    Console.WriteLine(value1);
}
catch (OverflowException)
{
    Console.WriteLine("{0} is out of range of a byte.", int1);
}
```

```

double dbl2 = 3.997;
try
{
    byte value2 = (byte)dbl2;
    Console.WriteLine(value2);
}
catch (OverflowException)
{
    Console.WriteLine("{0} is out of range of a byte.", dbl2);
}
// The example displays the following output:
//      128
//      3

```

- You can call a method of the [Convert](#) class to convert any supported type to a [Byte](#) value. This is possible because [Byte](#) supports the [IConvertible](#) interface. The following example illustrates the conversion of an array of [Int32](#) values to [Byte](#) values.

C#

```

int[] numbers = { Int32.MinValue, -1, 0, 121, 340, Int32.MaxValue };
byte result;
foreach (int number in numbers)
{
    try
    {
        result = Convert.ToByte(number);
        Console.WriteLine("Converted the {0} value {1} to the {2} value
{3}.",
                           number.GetType().Name, number,
                           result.GetType().Name, result);
    }
    catch (OverflowException)
    {
        Console.WriteLine("The {0} value {1} is outside the range of
the Byte type.",
                           number.GetType().Name, number);
    }
}
// The example displays the following output:
//      The Int32 value -2147483648 is outside the range of the Byte
//      type.
//      The Int32 value -1 is outside the range of the Byte type.
//      Converted the Int32 value 0 to the Byte value 0.
//      Converted the Int32 value 121 to the Byte value 121.
//      The Int32 value 340 is outside the range of the Byte type.
//      The Int32 value 2147483647 is outside the range of the Byte
//      type.

```

- You can call the [Parse](#) or [TryParse](#) method to convert the string representation of a [Byte](#) value to a [Byte](#). The string can contain either decimal or hexadecimal digits. The following example illustrates the parse operation by using both a decimal and a hexadecimal string.

C#

```
string string1 = "244";
try
{
    byte byte1 = Byte.Parse(string1);
    Console.WriteLine(byte1);
}
catch (OverflowException)
{
    Console.WriteLine("'{0}' is out of range of a byte.", string1);
}
catch (FormatException)
{
    Console.WriteLine("'{0}' is out of range of a byte.", string1);
}

string string2 = "F9";
try
{
    byte byte2 = Byte.Parse(string2,
        System.Globalization.NumberStyles.HexNumber);
    Console.WriteLine(byte2);
}
catch (OverflowException)
{
    Console.WriteLine("'{0}' is out of range of a byte.", string2);
}
catch (FormatException)
{
    Console.WriteLine("'{0}' is out of range of a byte.", string2);
}
// The example displays the following output:
//      244
//      249
```

## Perform operations on Byte values

The [Byte](#) type supports standard mathematical operations such as addition, subtraction, division, multiplication, subtraction, negation, and unary negation. Like the other integral types, the [Byte](#) type also supports the bitwise [AND](#), [OR](#), [XOR](#), left shift, and right shift operators.

You can use the standard numeric operators to compare two [Byte](#) values, or you can call the [CompareTo](#) or [Equals](#) method.

You can also call the members of the [Math](#) class to perform a wide range of numeric operations, including getting the absolute value of a number, calculating the quotient and remainder from integral division, determining the maximum or minimum value of two integers, getting the sign of a number, and rounding a number.

## Represent a Byte as a String

The [Byte](#) type provides full support for standard and custom numeric format strings. (For more information, see [Formatting Types](#), [Standard Numeric Format Strings](#), and [Custom Numeric Format Strings](#).) However, most commonly, byte values are represented as one-digit to three-digit values without any additional formatting, or as two-digit hexadecimal values.

To format a [Byte](#) value as an integral string with no leading zeros, you can call the parameterless [ToString\(\)](#) method. By using the "D" format specifier, you can also include a specified number of leading zeros in the string representation. By using the "X" format specifier, you can represent a [Byte](#) value as a hexadecimal string. The following example formats the elements in an array of [Byte](#) values in these three ways.

C#

```
byte[] numbers = { 0, 16, 104, 213 };
foreach (byte number in numbers)
{
    // Display value using default formatting.
    Console.Write("{0,-3} --> ", number.ToString());
    // Display value with 3 digits and leading zeros.
    Console.Write(number.ToString("D3") + " ");
    // Display value with hexadecimal.
    Console.Write(number.ToString("X2") + " ");
    // Display value with four hexadecimal digits.
    Console.WriteLine(number.ToString("X4"));
}
// The example displays the following output:
//      0    -->  000  00  0000
//     16    -->  016  10  0010
//    104    -->  104  68  0068
//   213    -->  213  D5  00D5
```

You can also format a [Byte](#) value as a binary, octal, decimal, or hexadecimal string by calling the [ToString\(Byte, Int32\)](#) method and supplying the base as the method's second

parameter. The following example calls this method to display the binary, octal, and hexadecimal representations of an array of byte values.

C#

```
byte[] numbers = { 0, 16, 104, 213 };
Console.WriteLine("{0} {1,8} {2,5} {3,5}",
                  "Value", "Binary", "Octal", "Hex");
foreach (byte number in numbers)
{
    Console.WriteLine("{0,5} {1,8} {2,5} {3,5}",
                      number, Convert.ToString(number, 2),
                      Convert.ToString(number, 8),
                      Convert.ToString(number, 16));
}
// The example displays the following output:
//      Value      Binary      Octal      Hex
//      0          0          0          0
//      16         10000      20         10
//      104        1101000    150        68
//      213        11010101   325        d5
```

## Work with non-decimal Byte values

In addition to working with individual bytes as decimal values, you may want to perform bitwise operations with byte values, or work with byte arrays or with the binary or hexadecimal representations of byte values. For example, overloads of the [BitConverter.GetBytes](#) method can convert each of the primitive data types to a byte array, and the [BigInteger.ToByteArray](#) method converts a [BigInteger](#) value to a byte array.

[Byte](#) values are represented in 8 bits by their magnitude only, without a sign bit. This is important to keep in mind when you perform bitwise operations on [Byte](#) values or when you work with individual bits. To perform a numeric, Boolean, or comparison operation on any two non-decimal values, both values must use the same representation.

When an operation is performed on two [Byte](#) values, the values share the same representation, so the result is accurate. This is illustrated in the following example, which masks the lowest-order bit of a [Byte](#) value to ensure that it is even.

C#

```
using System;
using System.Globalization;

public class Example
{
```

```

public static void Main()
{
    string[] values = { Convert.ToString(12, 16),
                        Convert.ToString(123, 16),
                        Convert.ToString(245, 16) };

    byte mask = 0xFE;
    foreach (string value in values) {
        Byte byteValue = Byte.Parse(value, NumberStyles.AllowHexSpecifier);
        Console.WriteLine("{0} And {1} = {2}", byteValue, mask,
                          byteValue & mask);
    }
}
// The example displays the following output:
//      12 And 254 = 12
//      123 And 254 = 122
//      245 And 254 = 244

```

On the other hand, when you work with both unsigned and signed bits, bitwise operations are complicated by the fact that the `SByte` values use sign-and-magnitude representation for positive values, and two's complement representation for negative values. In order to perform a meaningful bitwise operation, the values must be converted to two equivalent representations, and information about the sign bit must be preserved. The following example does this to mask out bits 2 and 4 of an array of 8-bit signed and unsigned values.

C#

```

using System;
using System.Collections.Generic;
using System.Globalization;

public struct ByteString
{
    public string Value;
    public int Sign;
}

public class Example1
{
    public static void Main()
    {
        ByteString[] values = CreateArray(-15, 123, 245);

        byte mask = 0x14;           // Mask all bits but 2 and 4.

        foreach (ByteString strValue in values)
        {
            byte byteValue = Byte.Parse(strValue.Value,
NumberStyles.AllowHexSpecifier);

```

```

        Console.WriteLine("{0} ({1}) And {2} ({3}) = {4} ({5})",
            strValue.Sign * byteValue,
            Convert.ToString(byteValue, 2),
            mask, Convert.ToString(mask, 2),
            (strValue.Sign & Math.Sign(mask)) * (byteValue
& mask),
            Convert.ToString(byteValue & mask, 2));
    }
}

private static ByteString[] CreateArray(params int[] values)
{
    List<ByteString> byteStrings = new List<ByteString>();

    foreach (object value in values)
    {
        ByteString temp = new ByteString();
        int sign = Math.Sign((int)value);
        temp.Sign = sign;

        // Change two's complement to magnitude-only representation.
        temp.Value = Convert.ToString(((int)value) * sign, 16);

        byteStrings.Add(temp);
    }
    return byteStrings.ToArray();
}
// The example displays the following output:
//      -15 (1111) And 20 (10100) = 4 (100)
//      123 (1111011) And 20 (10100) = 16 (10000)
//      245 (11110101) And 20 (10100) = 20 (10100)

```

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Decimal struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Decimal](#) value type represents decimal numbers ranging from positive 79,228,162,514,264,337,593,543,950,335 to negative 79,228,162,514,264,337,593,543,950,335. The default value of a [Decimal](#) is 0. The [Decimal](#) value type is appropriate for financial calculations that require large numbers of significant integral and fractional digits and no round-off errors. The [Decimal](#) type does not eliminate the need for rounding. Rather, it minimizes errors due to rounding. For example, the following code produces a result of 0.9999999999999999999999999999 instead of 1.

C#

```
decimal dividend = Decimal.One;
decimal divisor = 3;
// The following displays 0.999999999999999999999999999 to the console
Console.WriteLine(dividend/divisor * divisor);
```

When the result of the division and multiplication is passed to the [Round](#) method, the result suffers no loss of precision, as the following code shows.

C#

```
decimal dividend = Decimal.One;
decimal divisor = 3;
// The following displays 1.00 to the console
Console.WriteLine(Math.Round(dividend/divisor * divisor, 2));
```

A decimal number is a floating-point value that consists of a sign, a numeric value where each digit in the value ranges from 0 to 9, and a scaling factor that indicates the position of a floating decimal point that separates the integral and fractional parts of the numeric value.

The binary representation of a [Decimal](#) value is 128-bits consisting of a 96-bit integer number, and a 32-bit set of flags representing things such as the sign and scaling factor used to specify what portion of it is a decimal fraction. Therefore, the binary representation of a [Decimal](#) value the form,  $((-2^{96} \text{ to } 2^{96}) / 10^{(0 \text{ to } 28)})$ , where  $-(2^{96}-1)$  is equal to [MinValue](#), and  $2^{96}-1$  is equal to [MaxValue](#). For more information about the

binary representation of [Decimal](#) values and an example, see the [Decimal\(Int32\[\]\)](#) constructor and the [GetBits](#) method.

The scaling factor also preserves any trailing zeros in a [Decimal](#) number. Trailing zeros do not affect the value of a [Decimal](#) number in arithmetic or comparison operations. However, trailing zeros might be revealed by the [ToString](#) method if an appropriate format string is applied.

## Conversion considerations

This type provides methods that convert [Decimal](#) values to and from [SByte](#), [Int16](#), [Int32](#), [Int64](#), [Byte](#), [UInt16](#), [UInt32](#), and [UInt64](#) values. Conversions from these integral types to [Decimal](#) are widening conversions that never lose information or throw exceptions.

Conversions from [Decimal](#) to any of the integral types are narrowing conversions that round the [Decimal](#) value to the nearest integer value toward zero. Some languages, such as C#, also support the conversion of [Decimal](#) values to [Char](#) values. If the result of these conversions cannot be represented in the destination type, an [OverflowException](#) exception is thrown.

The [Decimal](#) type also provides methods that convert [Decimal](#) values to and from [Single](#) and [Double](#) values. Conversions from [Decimal](#) to [Single](#) or [Double](#) are narrowing conversions that might lose precision but not information about the magnitude of the converted value. The conversion does not throw an exception.

Conversions from [Single](#) or [Double](#) to [Decimal](#) throw an [OverflowException](#) exception if the result of the conversion cannot be represented as a [Decimal](#).

## Perform operations on [Decimal](#) values

The [Decimal](#) type supports standard mathematical operations such as addition, subtraction, division, multiplication, and unary negation. You can also work directly with the binary representation of a [Decimal](#) value by calling the [GetBits](#) method.

To compare two [Decimal](#) values, you can use the standard numeric comparison operators, or you can call the [CompareTo](#) or [Equals](#) method.

You can also call the members of the [Math](#) class to perform a wide range of numeric operations, including getting the absolute value of a number, determining the maximum or minimum value of two [Decimal](#) values, getting the sign of a number, and rounding a number.

# Examples

The following code example demonstrates the use of [Decimal](#).

C#

```
/// <summary>
/// Keeping my fortune in Decimals to avoid the round-off errors.
/// </summary>
class PiggyBank {
    protected decimal MyFortune;

    public void AddPenny() {
        MyFortune = Decimal.Add(MyFortune, .01m);
    }

    public decimal Capacity {
        get {
            return Decimal.MaxValue;
        }
    }

    public decimal Dollars {
        get {
            return Decimal.Floor(MyFortune);
        }
    }

    public decimal Cents {
        get {
            return Decimal.Subtract(MyFortune, Decimal.Floor(MyFortune));
        }
    }

    public override string ToString() {
        return MyFortune.ToString("C")+" in piggy bank";
    }
}
```

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

 .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

# System.Double struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Double](#) value type represents a double-precision 64-bit number with values ranging from negative 1.79769313486232e308 to positive 1.79769313486232e308, as well as positive or negative zero, [PositiveInfinity](#), [NegativeInfinity](#), and not a number ([NaN](#)). It is intended to represent values that are extremely large (such as distances between planets or galaxies) or extremely small (such as the molecular mass of a substance in kilograms) and that often are imprecise (such as the distance from earth to another solar system). The [Double](#) type complies with the IEC 60559:1989 (IEEE 754) standard for binary floating-point arithmetic.

## Floating-point representation and precision

The [Double](#) data type stores double-precision floating-point values in a 64-bit binary format, as shown in the following table:

Expand table

Part	Bits
Significand or mantissa	0-51
Exponent	52-62
Sign (0 = Positive, 1 = Negative)	63

Just as decimal fractions are unable to precisely represent some fractional values (such as 1/3 or [Math.PI](#)), binary fractions are unable to represent some fractional values. For example, 1/10, which is represented precisely by .1 as a decimal fraction, is represented by .001100110011 as a binary fraction, with the pattern "0011" repeating to infinity. In this case, the floating-point value provides an imprecise representation of the number that it represents. Performing additional mathematical operations on the original floating-point value often tends to increase its lack of precision. For example, if we compare the result of multiplying .1 by 10 and adding .1 to .1 nine times, we see that addition, because it has involved eight more operations, has produced the less precise result. Note that this disparity is apparent only if we display the two [Double](#) values by

using the "R" [standard numeric format string](#), which if necessary displays all 17 digits of precision supported by the `Double` type.

C#

```
using System;

public class Example13
{
    public static void Main()
    {
        Double value = .1;
        Double result1 = value * 10;
        Double result2 = 0;
        for (int ctr = 1; ctr <= 10; ctr++)
            result2 += value;

        Console.WriteLine(".1 * 10: {0:R}", result1);
        Console.WriteLine(".1 Added 10 times: {0:R}", result2);
    }
}

// The example displays the following output:
//      .1 * 10:      1
//      .1 Added 10 times: 0.9999999999999998
```

Because some numbers cannot be represented exactly as fractional binary values, floating-point numbers can only approximate real numbers.

All floating-point numbers also have a limited number of significant digits, which also determines how accurately a floating-point value approximates a real number. A `Double` value has up to 15 decimal digits of precision, although a maximum of 17 digits is maintained internally. This means that some floating-point operations may lack the precision to change a floating point value. The following example provides an illustration. It defines a very large floating-point value, and then adds the product of `Double.Epsilon` and one quadrillion to it. The product, however, is too small to modify the original floating-point value. Its least significant digit is thousandths, whereas the most significant digit in the product is  $10^{-309}$ .

C#

```
using System;

public class Example14
{
    public static void Main()
    {
        Double value = 123456789012.34567;
        Double additional = Double.Epsilon * 1e15;
        Console.WriteLine("{0} + {1} = {2}", value, additional,
```

```

        value + additional);
    }
}

// The example displays the following output:
//      123456789012.346 + 4.94065645841247E-309 = 123456789012.346

```

The limited precision of a floating-point number has several consequences:

- Two floating-point numbers that appear equal for a particular precision might not compare equal because their least significant digits are different. In the following example, a series of numbers are added together, and their total is compared with their expected total. Although the two values appear to be the same, a call to the `Equals` method indicates that they are not.

```

C#

using System;

public class Example10
{
    public static void Main()
    {
        Double[] values = { 10.0, 2.88, 2.88, 2.88, 9.0 };
        Double result = 27.64;
        Double total = 0;
        foreach (var value in values)
            total += value;

        if (total.Equals(result))
            Console.WriteLine("The sum of the values equals the
total.");
        else
            Console.WriteLine("The sum of the values ({0}) does not
equal the total ({1}).",
                            total, result);
    }
}

// The example displays the following output:
//      The sum of the values (36.64) does not equal the total (36.64).
//
// If the index items in the Console.WriteLine statement are changed to
// {0:R},
// the example displays the following output:
//      The sum of the values (27.63999999999997) does not equal the
total (27.64).

```

If you change the format items in the `Console.WriteLine(String, Object, Object)` statement from `{0}` and `{1}` to `{0:R}` and `{1:R}` to display all significant digits of the two `Double` values, it is clear that the two values are unequal because of a loss

of precision during the addition operations. In this case, the issue can be resolved by calling the `Math.Round(Double, Int32)` method to round the `Double` values to the desired precision before performing the comparison.

- A mathematical or comparison operation that uses a floating-point number might not yield the same result if a decimal number is used, because the binary floating-point number might not equal the decimal number. A previous example illustrated this by displaying the result of multiplying .1 by 10 and adding .1 times.

When accuracy in numeric operations with fractional values is important, you can use the `Decimal` rather than the `Double` type. When accuracy in numeric operations with integral values beyond the range of the `Int64` or `UInt64` types is important, use the `BigInteger` type.

- A value might not round-trip if a floating-point number is involved. A value is said to round-trip if an operation converts an original floating-point number to another form, an inverse operation transforms the converted form back to a floating-point number, and the final floating-point number is not equal to the original floating-point number. The round trip might fail because one or more least significant digits are lost or changed in a conversion. In the following example, three `Double` values are converted to strings and saved in a file. As the output shows, however, even though the values appear to be identical, the restored values are not equal to the original values.

```
C#  
  
using System;  
using System.IO;  
  
public class Example11  
{  
    public static void Main()  
    {  
        StreamWriter sw = new StreamWriter(@".\Doubles.dat");  
        Double[] values = { 2.2 / 1.01, 1.0 / 3, Math.PI };  
        for (int ctr = 0; ctr < values.Length; ctr++)  
        {  
            sw.Write(values[ctr].ToString());  
            if (ctr != values.Length - 1)  
                sw.Write("|");  
        }  
        sw.Close();  
  
        Double[] restoredValues = new Double[values.Length];  
        StreamReader sr = new StreamReader(@".\Doubles.dat");  
        string temp = sr.ReadToEnd();  
        string[] tempStrings = temp.Split('|');  
        for (int ctr = 0; ctr < tempStrings.Length; ctr++)
```

```

        restoredValues[ctr] = Double.Parse(tempStrings[ctr]);

        for (int ctr = 0; ctr < values.Length; ctr++)
            Console.WriteLine("{0} {2} {1}", values[ctr],
                               restoredValues[ctr],
                               values[ctr].Equals(restoredValues[ctr]) ?
                               "=" : "<>");
    }
}

// The example displays the following output:
//      2.17821782178218 <> 2.17821782178218
//      0.3333333333333333 <> 0.3333333333333333
//      3.14159265358979 <> 3.14159265358979

```

In this case, the values can be successfully round-tripped by using the "G17" [standard numeric format string](#) to preserve the full precision of `Double` values, as the following example shows.

C#

```

using System;
using System.IO;

public class Example12
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\Doubles.dat");
        Double[] values = { 2.2 / 1.01, 1.0 / 3, Math.PI };
        for (int ctr = 0; ctr < values.Length; ctr++)
            sw.Write("{0:G17}{1}", values[ctr], ctr < values.Length - 1
? "|" : "");

        sw.Close();

        Double[] restoredValues = new Double[values.Length];
        StreamReader sr = new StreamReader(@".\Doubles.dat");
        string temp = sr.ReadToEnd();
        string[] tempStrings = temp.Split('|');
        for (int ctr = 0; ctr < tempStrings.Length; ctr++)
            restoredValues[ctr] = Double.Parse(tempStrings[ctr]);

        for (int ctr = 0; ctr < values.Length; ctr++)
            Console.WriteLine("{0} {2} {1}", values[ctr],
                               restoredValues[ctr],
                               values[ctr].Equals(restoredValues[ctr]) ?
                               "=" : "<>");
    }
}

// The example displays the following output:
//      2.17821782178218 = 2.17821782178218

```

```
//      0.3333333333333333 = 0.3333333333333333
//      3.14159265358979 = 3.14159265358979
```

### ⓘ Important

When used with a **Double** value, the "R" format specifier in some cases fails to successfully round-trip the original value. To ensure that **Double** values successfully round-trip, use the "G17" format specifier.

- **Single** values have less precision than **Double** values. A **Single** value that is converted to a seemingly equivalent **Double** often does not equal the **Double** value because of differences in precision. In the following example, the result of identical division operations is assigned to a **Double** and a **Single** value. After the **Single** value is cast to a **Double**, a comparison of the two values shows that they are unequal.

C#

```
using System;

public class Example9
{
    public static void Main()
    {
        Double value1 = 1 / 3.0;
        Single sValue2 = 1 / 3.0f;
        Double value2 = (Double)sValue2;
        Console.WriteLine("{0:R} = {1:R}: {2}", value1, value2,
                           value1.Equals(value2));
    }
}
// The example displays the following output:
//      0.3333333333333331 = 0.333333432674408: False
```

To avoid this problem, use either the **Double** in place of the **Single** data type, or use the **Round** method so that both values have the same precision.

In addition, the result of arithmetic and assignment operations with **Double** values may differ slightly by platform because of the loss of precision of the **Double** type. For example, the result of assigning a literal **Double** value may differ in the 32-bit and 64-bit versions of .NET. The following example illustrates this difference when the literal value -4.42330604244772E-305 and a variable whose value is -4.42330604244772E-305 are assigned to a **Double** variable. Note that the result of the **Parse(String)** method in this case does not suffer from a loss of precision.

C#

```
double value = -4.42330604244772E-305;

double fromLiteral = -4.42330604244772E-305;
double fromVariable = value;
double fromParse = Double.Parse("-4.42330604244772E-305");

Console.WriteLine("Double value from literal: {0,29:R}", fromLiteral);
Console.WriteLine("Double value from variable: {0,28:R}", fromVariable);
Console.WriteLine("Double value from Parse method: {0,24:R}", fromParse);
// On 32-bit versions of the .NET Framework, the output is:
//    Double value from literal:      -4.42330604244772E-305
//    Double value from variable:     -4.42330604244772E-305
//    Double value from Parse method: -4.42330604244772E-305
//
// On other versions of the .NET Framework, the output is:
//    Double value from literal:      -4.4233060424477198E-305
//    Double value from variable:     -4.4233060424477198E-305
//    Double value from Parse method: -4.42330604244772E-305
```

## Test for equality

To be considered equal, two [Double](#) values must represent identical values. However, because of differences in precision between values, or because of a loss of precision by one or both values, floating-point values that are expected to be identical often turn out to be unequal because of differences in their least significant digits. As a result, calls to the [Equals](#) method to determine whether two values are equal, or calls to the [CompareTo](#) method to determine the relationship between two [Double](#) values, often yield unexpected results. This is evident in the following example, where two apparently equal [Double](#) values turn out to be unequal because the first has 15 digits of precision, while the second has 17.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        double value1 = .3333333333333333;
        double value2 = 1.0/3;
        Console.WriteLine("{0:R} = {1:R}: {2}", value1, value2,
value1.Equals(value2));
    }
}
```

```
// The example displays the following output:  
//      0.3333333333333333 = 0.3333333333333333: False
```

Calculated values that follow different code paths and that are manipulated in different ways often prove to be unequal. In the following example, one `Double` value is squared, and then the square root is calculated to restore the original value. A second `Double` is multiplied by 3.51 and squared before the square root of the result is divided by 3.51 to restore the original value. Although the two values appear to be identical, a call to the `Equals(Double)` method indicates that they are not equal. Using the "R" standard format string to return a result string that displays all the significant digits of each `Double` value shows that the second value is .000000000001 less than the first.

C#

```
using System;  
  
public class Example1  
{  
    public static void Main()  
    {  
        double value1 = 100.10142;  
        value1 = Math.Sqrt(Math.Pow(value1, 2));  
        double value2 = Math.Pow(value1 * 3.51, 2);  
        value2 = Math.Sqrt(value2) / 3.51;  
        Console.WriteLine("{0} = {1}: {2}\n",  
                           value1, value2, value1.Equals(value2));  
        Console.WriteLine("{0:R} = {1:R}", value1, value2);  
    }  
}  
// The example displays the following output:  
//      100.10142 = 100.10142: False  
//  
//      100.10142 = 100.10141999999999
```

In cases where a loss of precision is likely to affect the result of a comparison, you can adopt any of the following alternatives to calling the `Equals` or `CompareTo` method:

- Call the `Math.Round` method to ensure that both values have the same precision. The following example modifies a previous example to use this approach so that two fractional values are equivalent.

C#

```
using System;  
  
public class Example2  
{  
    public static void Main()
```

```

    {
        double value1 = .333333333333333;
        double value2 = 1.0 / 3;
        int precision = 7;
        value1 = Math.Round(value1, precision);
        value2 = Math.Round(value2, precision);
        Console.WriteLine("{0:R} = {1:R}: {2}", value1, value2,
value1.Equals(value2));
    }
}

// The example displays the following output:
//      0.3333333 = 0.3333333: True

```

The problem of precision still applies to rounding of midpoint values. For more information, see the [Math.Round\(Double, Int32, MidpointRounding\)](#) method.

- Test for approximate equality rather than equality. This requires that you define either an absolute amount by which the two values can differ but still be equal, or that you define a relative amount by which the smaller value can diverge from the larger value.

### ⚠ Warning

**Double.Epsilon** is sometimes used as an absolute measure of the distance between two **Double** values when testing for equality. However, **Double.Epsilon** measures the smallest possible value that can be added to, or subtracted from, a **Double** whose value is zero. For most positive and negative **Double** values, the value of **Double.Epsilon** is too small to be detected. Therefore, except for values that are zero, we do not recommend its use in tests for equality.

The following example uses the latter approach to define an `IsApproximatelyEqual` method that tests the relative difference between two values. It also contrasts the result of calls to the `IsApproximatelyEqual` method and the `Equals(Double)` method.

C#

```

using System;

public class Example3
{
    public static void Main()
    {
        double one1 = .1 * 10;
        double one2 = 0;

```

```

        for (int ctr = 1; ctr <= 10; ctr++)
            one2 += .1;

            Console.WriteLine("{0:R} = {1:R}: {2}", one1, one2,
one1.Equals(one2));
            Console.WriteLine("{0:R} is approximately equal to {1:R}: {2}",
one1, one2,
IsApproximatelyEqual(one1, one2,
.00000001));
    }

    static bool IsApproximatelyEqual(double value1, double value2,
double epsilon)
    {
        // If they are equal anyway, just return True.
        if (value1.Equals(value2))
            return true;

        // Handle NaN, Infinity.
        if (Double.IsInfinity(value1) | Double.IsNaN(value1))
            return value1.Equals(value2);
        else if (Double.IsInfinity(value2) | Double.IsNaN(value2))
            return value1.Equals(value2);

        // Handle zero to avoid division by zero
        double divisor = Math.Max(value1, value2);
        if (divisor.Equals(0))
            divisor = Math.Min(value1, value2);

        return Math.Abs((value1 - value2) / divisor) <= epsilon;
    }
}

// The example displays the following output:
//      1 = 0.9999999999999989: False
//      1 is approximately equal to 0.9999999999999989: True

```

## Floating-point values and exceptions

Unlike operations with integral types, which throw exceptions in cases of overflow or illegal operations such as division by zero, operations with floating-point values do not throw exceptions. Instead, in exceptional situations, the result of a floating-point operation is zero, positive infinity, negative infinity, or not a number (NaN):

- If the result of a floating-point operation is too small for the destination format, the result is zero. This can occur when two very small numbers are multiplied, as the following example shows.

```

using System;

public class Example6
{
    public static void Main()
    {
        Double value1 = 1.1632875981534209e-225;
        Double value2 = 9.1642346778e-175;
        Double result = value1 * value2;
        Console.WriteLine("{0} * {1} = {2}", value1, value2, result);
        Console.WriteLine("{0} = 0: {1}", result, result.Equals(0.0));
    }
}
// The example displays the following output:
//      1.16328759815342E-225 * 9.1642346778E-175 = 0
//      0 = 0: True

```

- If the magnitude of the result of a floating-point operation exceeds the range of the destination format, the result of the operation is [PositiveInfinity](#) or [NegativeInfinity](#), as appropriate for the sign of the result. The result of an operation that overflows [Double.MaxValue](#) is [PositiveInfinity](#), and the result of an operation that overflows [Double.MinValue](#) is [NegativeInfinity](#), as the following example shows.

C#

```

using System;

public class Example7
{
    public static void Main()
    {
        Double value1 = 4.565e153;
        Double value2 = 6.9375e172;
        Double result = value1 * value2;
        Console.WriteLine("PositiveInfinity: {0}",
                          Double.IsPositiveInfinity(result));
        Console.WriteLine("NegativeInfinity: {0}\n",
                          Double.IsNegativeInfinity(result));

        value1 = -value1;
        result = value1 * value2;
        Console.WriteLine("PositiveInfinity: {0}",
                          Double.IsPositiveInfinity(result));
        Console.WriteLine("NegativeInfinity: {0}",
                          Double.IsNegativeInfinity(result));
    }
}

// The example displays the following output:
//      PositiveInfinity: True

```

```
//      NegativeInfinity: False
//      PositiveInfinity: False
//      NegativeInfinity: True
```

`PositiveInfinity` also results from a division by zero with a positive dividend, and `NegativeInfinity` results from a division by zero with a negative dividend.

- If a floating-point operation is invalid, the result of the operation is **NaN**. For example, **NaN** results from the following operations:
    - Division by zero with a dividend of zero. Note that other cases of division by zero result in either **PositiveInfinity** or **NegativeInfinity**.
    - Any floating-point operation with an invalid input. For example, calling the **Math.Sqrt** method with a negative value returns **NaN**, as does calling the **Math.Acos** method with a value that is greater than one or less than negative one.
    - Any operation with an argument whose value is **Double.NaN**.

# Type conversions

The [Double](#) structure does not define any explicit or implicit conversion operators; instead, conversions are implemented by the compiler.

The conversion of the value of any primitive numeric type to a [Double](#) is a widening conversion and therefore does not require an explicit cast operator or call to a conversion method unless a compiler explicitly requires it. For example, the C# compiler requires a casting operator for conversions from [Decimal](#) to [Double](#), while the Visual Basic compiler does not. The following example converts the minimum or maximum value of other primitive numeric types to a [Double](#).

C#

```

UInt16.MinValue,
                    UInt16.MaxValue, UInt32.MinValue,
UInt32.MaxValue,
                    UInt64.MinValue, UInt64.MaxValue };

        double dblValue;
        foreach (var value in values)
        {
            if (value.GetType() == typeof(Decimal))
                dblValue = (Double)value;
            else
                dblValue = value;
            Console.WriteLine("{0} ({1}) --> {2:R} ({3})",
                value, value.GetType().Name,
                dblValue, dblValue.GetType().Name);
        }
    }
}

// The example displays the following output:
//    0 (Byte) --> 0 (Double)
//    255 (Byte) --> 255 (Double)
//    -79228162514264337593543950335 (Decimal) --> -7.9228162514264338E+28
// (Double)
//    79228162514264337593543950335 (Decimal) --> 7.9228162514264338E+28
// (Double)
//    -32768 (Int16) --> -32768 (Double)
//    32767 (Int16) --> 32767 (Double)
//    -2147483648 (Int32) --> -2147483648 (Double)
//    2147483647 (Int32) --> 2147483647 (Double)
//    -9223372036854775808 (Int64) --> -9.2233720368547758E+18 (Double)
//    9223372036854775807 (Int64) --> 9.2233720368547758E+18 (Double)
//    -128 (SByte) --> -128 (Double)
//    127 (SByte) --> 127 (Double)
//    -3.402823E+38 (Single) --> -3.4028234663852886E+38 (Double)
//    3.402823E+38 (Single) --> 3.4028234663852886E+38 (Double)
//    0 (UInt16) --> 0 (Double)
//    65535 (UInt16) --> 65535 (Double)
//    0 (UInt32) --> 0 (Double)
//    4294967295 (UInt32) --> 4294967295 (Double)
//    0 (UInt64) --> 0 (Double)
//    18446744073709551615 (UInt64) --> 1.8446744073709552E+19 (Double)

```

In addition, the [Single](#) values [Single.NaN](#), [Single.PositiveInfinity](#), and [Single.NegativeInfinity](#) convert to [Double.NaN](#), [Double.PositiveInfinity](#), and [Double.NegativeInfinity](#), respectively.

Note that the conversion of the value of some numeric types to a [Double](#) value can involve a loss of precision. As the example illustrates, a loss of precision is possible when converting [Decimal](#), [Int64](#), and [UInt64](#) values to [Double](#) values.

The conversion of a [Double](#) value to a value of any other primitive numeric data type is a narrowing conversion and requires a cast operator (in C#), a conversion method (in

Visual Basic), or a call to a [Convert](#) method. Values that are outside the range of the target data type, which are defined by the target type's `MinValue` and `MaxValue` properties, behave as shown in the following table.

[\[\] Expand table](#)

Target type	Result
Any integral type	An <a href="#">OverflowException</a> exception if the conversion occurs in a checked context. If the conversion occurs in an unchecked context (the default in C#), the conversion operation succeeds but the value overflows.
Decimal	An <a href="#">OverflowException</a> exception.
Single	<a href="#">Single.NegativeInfinity</a> for negative values. <a href="#">Single.PositiveInfinity</a> for positive values.

In addition, [Double.NaN](#), [Double.PositiveInfinity](#), and [Double.NegativeInfinity](#) throw an [OverflowException](#) for conversions to integers in a checked context, but these values overflow when converted to integers in an unchecked context. For conversions to [Decimal](#), they always throw an [OverflowException](#). For conversions to [Single](#), they convert to [Single.NaN](#), [Single.PositiveInfinity](#), and [Single.NegativeInfinity](#), respectively.

A loss of precision may result from converting a [Double](#) value to another numeric type. In the case of converting to any of the integral types, as the output from the example shows, the fractional component is lost when the [Double](#) value is either rounded (as in Visual Basic) or truncated (as in C#). For conversions to [Decimal](#) and [Single](#) values, the [Double](#) value may not have a precise representation in the target data type.

The following example converts a number of [Double](#) values to several other numeric types. The conversions occur in a checked context in Visual Basic (the default), in C# (because of the `checked` keyword), and in F# (because of the `Checked` module). The output from the example shows the result for conversions in both a checked and unchecked context. You can perform conversions in an unchecked context in Visual Basic by compiling with the `/removeintchecks+` compiler switch, in C# by commenting out the `checked` statement, and in F# by commenting out the `open Checked` statement.

C#

```
using System;

public class Example5
{
    public static void Main()
```

```
{  
    Double[] values = { Double.MinValue, -67890.1234, -12345.6789,  
                        12345.6789, 67890.1234, Double.MaxValue,  
                        Double.NaN, Double.PositiveInfinity,  
                        Double.NegativeInfinity };  
    checked  
    {  
        foreach (var value in values)  
        {  
            try  
            {  
                Int64 lValue = (long)value;  
                Console.WriteLine("{0} ({1}) --> {2} (0x{2:X16}) ({3})",
value, value.GetType().Name,  
                           lValue, lValue.GetType().Name);  
            }  
            catch (OverflowException)  
            {  
                Console.WriteLine("Unable to convert {0} to Int64.",
value);  
            }  
            try  
            {  
                UInt64 ulValue = (ulong)value;  
                Console.WriteLine("{0} ({1}) --> {2} (0x{2:X16}) ({3})",
value, value.GetType().Name,  
                           ulValue, ulValue.GetType().Name);  
            }  
            catch (OverflowException)  
            {  
                Console.WriteLine("Unable to convert {0} to UInt64.",
value);  
            }  
            try  
            {  
                Decimal dValue = (decimal)value;  
                Console.WriteLine("{0} ({1}) --> {2} ({3})",
value, value.GetType().Name,  
                           dValue, dValue.GetType().Name);  
            }  
            catch (OverflowException)  
            {  
                Console.WriteLine("Unable to convert {0} to Decimal.",
value);  
            }  
            try  
            {  
                Single sValue = (float)value;  
                Console.WriteLine("{0} ({1}) --> {2} ({3})",
value, value.GetType().Name,  
                           sValue, sValue.GetType().Name);  
            }  
            catch (OverflowException)  
            {  
                Console.WriteLine("Unable to convert {0} to Single.",
value);  
            }  
        }  
    }  
}
```

```
value);
        }
        Console.WriteLine();
    }
}
// The example displays the following output for conversions performed
// in a checked context:
//    Unable to convert -1.79769313486232E+308 to Int64.
//    Unable to convert -1.79769313486232E+308 to UInt64.
//    Unable to convert -1.79769313486232E+308 to Decimal.
//    -1.79769313486232E+308 (Double) --> -Infinity (Single)
//
//    -67890.1234 (Double) --> -67890 (0xFFFFFFFFFFFFFEF6CE) (Int64)
//    Unable to convert -67890.1234 to UInt64.
//    -67890.1234 (Double) --> -67890.1234 (Decimal)
//    -67890.1234 (Double) --> -67890.13 (Single)
//
//    -12345.6789 (Double) --> -12345 (0xFFFFFFFFFFFFFCFC7) (Int64)
//    Unable to convert -12345.6789 to UInt64.
//    -12345.6789 (Double) --> -12345.6789 (Decimal)
//    -12345.6789 (Double) --> -12345.68 (Single)
//
//    12345.6789 (Double) --> 12345 (0x0000000000003039) (Int64)
//    12345.6789 (Double) --> 12345 (0x0000000000003039) (UInt64)
//    12345.6789 (Double) --> 12345.6789 (Decimal)
//    12345.6789 (Double) --> 12345.68 (Single)
//
//    67890.1234 (Double) --> 67890 (0x00000000000010932) (Int64)
//    67890.1234 (Double) --> 67890 (0x00000000000010932) (UInt64)
//    67890.1234 (Double) --> 67890.1234 (Decimal)
//    67890.1234 (Double) --> 67890.13 (Single)
//
//    Unable to convert 1.79769313486232E+308 to Int64.
//    Unable to convert 1.79769313486232E+308 to UInt64.
//    Unable to convert 1.79769313486232E+308 to Decimal.
//    1.79769313486232E+308 (Double) --> Infinity (Single)
//
//    Unable to convert NaN to Int64.
//    Unable to convert NaN to UInt64.
//    Unable to convert NaN to Decimal.
//    NaN (Double) --> NaN (Single)
//
//    Unable to convert Infinity to Int64.
//    Unable to convert Infinity to UInt64.
//    Unable to convert Infinity to Decimal.
//    Infinity (Double) --> Infinity (Single)
//
//    Unable to convert -Infinity to Int64.
//    Unable to convert -Infinity to UInt64.
//    Unable to convert -Infinity to Decimal.
//    -Infinity (Double) --> -Infinity (Single)
// The example displays the following output for conversions performed
// in an unchecked context:
```

```
//      -1.79769313486232E+308 (Double) --> -9223372036854775808
(0x8000000000000000) (Int64)
//      -1.79769313486232E+308 (Double) --> 9223372036854775808
(0x8000000000000000) (UInt64)
//      Unable to convert -1.79769313486232E+308 to Decimal.
//      -1.79769313486232E+308 (Double) --> -Infinity (Single)
//
//      -67890.1234 (Double) --> -67890 (0xFFFFFFFFFFFFEF6CE) (Int64)
//      -67890.1234 (Double) --> 18446744073709483726 (0xFFFFFFFFFFFFEF6CE)
(UInt64)
//      -67890.1234 (Double) --> -67890.1234 (Decimal)
//      -67890.1234 (Double) --> -67890.13 (Single)
//
//      -12345.6789 (Double) --> -12345 (0xFFFFFFFFFFFFCFC7) (Int64)
//      -12345.6789 (Double) --> 18446744073709539271 (0xFFFFFFFFFFFFCFC7)
(UInt64)
//      -12345.6789 (Double) --> -12345.6789 (Decimal)
//      -12345.6789 (Double) --> -12345.68 (Single)
//
//      12345.6789 (Double) --> 12345 (0x000000000003039) (Int64)
//      12345.6789 (Double) --> 12345 (0x000000000003039) (UInt64)
//      12345.6789 (Double) --> 12345.6789 (Decimal)
//      12345.6789 (Double) --> 12345.68 (Single)
//
//      67890.1234 (Double) --> 67890 (0x0000000000010932) (Int64)
//      67890.1234 (Double) --> 67890 (0x0000000000010932) (UInt64)
//      67890.1234 (Double) --> 67890.1234 (Decimal)
//      67890.1234 (Double) --> 67890.13 (Single)
//
//      1.79769313486232E+308 (Double) --> -9223372036854775808
(0x8000000000000000) (Int64)
//      1.79769313486232E+308 (Double) --> 0 (0x0000000000000000) (UInt64)
//      Unable to convert 1.79769313486232E+308 to Decimal.
//      1.79769313486232E+308 (Double) --> Infinity (Single)
//
//      NaN (Double) --> -9223372036854775808 (0x8000000000000000) (Int64)
//      NaN (Double) --> 0 (0x0000000000000000) (UInt64)
//      Unable to convert NaN to Decimal.
//      NaN (Double) --> NaN (Single)
//
//      Infinity (Double) --> -9223372036854775808 (0x8000000000000000)
(Int64)
//      Infinity (Double) --> 0 (0x0000000000000000) (UInt64)
//      Unable to convert Infinity to Decimal.
//      Infinity (Double) --> Infinity (Single)
//
//      -Infinity (Double) --> -9223372036854775808 (0x8000000000000000)
(Int64)
//      -Infinity (Double) --> 9223372036854775808 (0x8000000000000000)
(UInt64)
//      Unable to convert -Infinity to Decimal.
//      -Infinity (Double) --> -Infinity (Single)
```

For more information on the conversion of numeric types, see [Type Conversion in .NET](#) and [Type Conversion Tables](#).

## Floating-point functionality

The [Double](#) structure and related types provide methods to perform operations in the following areas:

- **Comparison of values.** You can call the [Equals](#) method to determine whether two [Double](#) values are equal, or the [CompareTo](#) method to determine the relationship between two values.

The [Double](#) structure also supports a complete set of comparison operators. For example, you can test for equality or inequality, or determine whether one value is greater than or equal to another. If one of the operands is a numeric type other than a [Double](#), it is converted to a [Double](#) before performing the comparison.

### ⚠ Warning

Because of differences in precision, two [Double](#) values that you expect to be equal may turn out to be unequal, which affects the result of the comparison. See the [Test for equality](#) section for more information about comparing two [Double](#) values.

You can also call the [IsNaN](#), [IsInfinity](#), [IsPositiveInfinity](#), and [IsNegativeInfinity](#) methods to test for these special values.

- **Mathematical operations.** Common arithmetic operations, such as addition, subtraction, multiplication, and division, are implemented by language compilers and Common Intermediate Language (CIL) instructions, rather than by [Double](#) methods. If one of the operands in a mathematical operation is a numeric type other than a [Double](#), it is converted to a [Double](#) before performing the operation. The result of the operation is also a [Double](#) value.

Other mathematical operations can be performed by calling `static` (`Shared` in Visual Basic) methods in the [System.Math](#) class. It includes additional methods commonly used for arithmetic (such as [Math.Abs](#), [Math.Sign](#), and [Math.Sqrt](#)), geometry (such as [Math.Cos](#) and [Math.Sin](#)), and calculus (such as [Math.Log](#)).

You can also manipulate the individual bits in a [Double](#) value. The [BitConverter.DoubleToInt64Bits](#) method preserves a [Double](#) value's bit pattern in a

64-bit integer. The [BitConverter.GetBytes\(Double\)](#) method returns its bit pattern in a byte array.

- **Rounding.** Rounding is often used as a technique for reducing the impact of differences between values caused by problems of floating-point representation and precision. You can round a [Double](#) value by calling the [Math.Round](#) method.
- **Formatting.** You can convert a [Double](#) value to its string representation by calling the [ToString](#) method or by using the composite formatting feature. For information about how format strings control the string representation of floating-point values, see the [Standard Numeric Format Strings](#) and [Custom Numeric Format Strings](#) topics.
- **Parsing strings.** You can convert the string representation of a floating-point value to a [Double](#) value by calling either the [Parse](#) or [TryParse](#) method. If the parse operation fails, the [Parse](#) method throws an exception, whereas the [TryParse](#) method returns `false`.
- **Type conversion.** The [Double](#) structure provides an explicit interface implementation for the [IConvertible](#) interface, which supports conversion between any two standard .NET data types. Language compilers also support the implicit conversion of values of all other standard numeric types to [Double](#) values. Conversion of a value of any standard numeric type to a [Double](#) is a widening conversion and does not require the user of a casting operator or conversion method,

However, conversion of [Int64](#) and [Single](#) values can involve a loss of precision. The following table lists the differences in precision for each of these types:

[ ] [Expand table](#)

Type	Maximum precision	Internal precision
<a href="#">Double</a>	15	17
<a href="#">Int64</a>	19 decimal digits	19 decimal digits
<a href="#">Single</a>	7 decimal digits	9 decimal digits

The problem of precision most frequently affects [Single](#) values that are converted to [Double](#) values. In the following example, two values produced by identical division operations are unequal because one of the values is a single-precision floating point value converted to a [Double](#).

C#

```
using System;

public class Example13
{
    public static void Main()
    {
        Double value = .1;
        Double result1 = value * 10;
        Double result2 = 0;
        for (int ctr = 1; ctr <= 10; ctr++)
            result2 += value;

        Console.WriteLine(".1 * 10: {0:R}", result1);
        Console.WriteLine(".1 Added 10 times: {0:R}", result2);
    }
}

// The example displays the following output:
//      .1 * 10: 1
//      .1 Added 10 times: 0.999999999999989
```

## Examples

The following code example illustrates the use of [Double](#):

C#

```
// The Temperature class stores the temperature as a Double
// and delegates most of the functionality to the Double
// implementation.
public class Temperature : IComparable, IFormattable
{
    // IComparable.CompareTo implementation.
    public int CompareTo(object obj) {
        if (obj == null) return 1;

        Temperature temp = obj as Temperature;
        if (obj != null)
            return m_value.CompareTo(temp.m_value);
        else
            throw new ArgumentException("object is not a Temperature");
    }

    // IFormattable.ToString implementation.
    public string ToString(string format, IFormatProvider provider) {
        if( format != null ) {
            if( format.Equals("F") ) {
                return String.Format("{0}'F", this.Value.ToString());
            }
            if( format.Equals("C") ) {
```

```
        return String.Format("{0}'C", this.Celsius.ToString()));
    }

    return m_value.ToString(format, provider);
}

// Parses the temperature from a string in the form
// [ws][sign]digits['F'|'C'][ws]
public static Temperature Parse(string s, NumberStyles styles,
IFormatProvider provider) {
    Temperature temp = new Temperature();

    if( s.TrimEnd(null).EndsWith("'F") ) {
        temp.Value = Double.Parse( s.Remove(s.LastIndexOf('\''), 2),
styles, provider);
    }
    else if( s.TrimEnd(null).EndsWith("'C") ) {
        temp.Celsius = Double.Parse( s.Remove(s.LastIndexOf('\''), 2),
styles, provider);
    }
    else {
        temp.Value = Double.Parse(s, styles, provider);
    }

    return temp;
}

// The value holder
protected double m_value;

public double Value {
    get {
        return m_value;
    }
    set {
        m_value = value;
    }
}

public double Celsius {
    get {
        return (m_value-32.0)/1.8;
    }
    set {
        m_value = 1.8*value+32.0;
    }
}
}
```

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Double.CompareTo methods

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

## CompareTo(Double) method

Values must be identical to be considered equal. Particularly when floating-point values depend on multiple mathematical operations, it is common for them to lose precision and for their values to be nearly identical except for their least significant digits. Because of this, the return value of the [CompareTo](#) method at times may seem surprising. For example, multiplication by a particular value followed by division by the same value should produce the original value. In the following example, however, the computed value turns out to be greater than the original value. Showing all significant digits of the two values by using the "R" [standard numeric format string](#) indicates that the computed value differs from the original value in its least significant digits. For information on handling such comparisons, see the Remarks section of the [Equals\(Double\)](#) method.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        double value1 = 6.185;
        double value2 = value1 * .1 / .1;
        Console.WriteLine("Comparing {0} and {1}: {2}\n",
                           value1, value2, value1.CompareTo(value2));
        Console.WriteLine("Comparing {0:R} and {1:R}: {2}",
                           value1, value2, value1.CompareTo(value2));
    }
}
// The example displays the following output:
//      Comparing 6.185 and 6.185: -1
//
//      Comparing 6.185 and 6.185000000000005: -1
```

This method implements the [System.IComparable<T>](#) interface and performs slightly better than the [Double.CompareTo](#) method because it does not have to convert the `value` parameter to an object.

Note that, although an object whose value is `NaN` is not considered equal to another object whose value is `NaN` (even itself), the `IComparable<T>` interface requires that `A.CompareTo(A)` return zero.

## CompareTo(Object) method

The `value` parameter must be `null` or an instance of `Double`; otherwise, an exception is thrown. Any instance of `Double`, regardless of its value, is considered greater than `null`.

Values must be identical to be considered equal. Particularly when floating-point values depend on multiple mathematical operations, it is common for them to lose precision and for their values to be nearly identical except for their least significant digits. Because of this, the return value of the `CompareTo` method at times may seem surprising. For example, multiplication by a particular value followed by division by the same value should produce the original value. In the following example, however, the computed value turns out to be greater than the original value. Showing all significant digits of the two values by using the "R" [standard numeric format string](#) indicates that the computed value differs from the original value in its least significant digits. For information on handling such comparisons, see the Remarks section of the [Equals\(Double\)](#) method.

C#

```
using System;

public class Example3
{
    public static void Main()
    {
        double value1 = 6.185;
        object value2 = value1 * .1 / .1;
        Console.WriteLine("Comparing {0} and {1}: {2}\n",
                           value1, value2, value1.CompareTo(value2));
        Console.WriteLine("Comparing {0:R} and {1:R}: {2}",
                           value1, value2, value1.CompareTo(value2));
    }
}
// The example displays the following output:
//      Comparing 6.185 and 6.185: -1
//
//      Comparing 6.185 and 6.185000000000005: -1
```

This method is implemented to support the `IComparable` interface. Note that, although a `NaN` is not considered to be equal to another `NaN` (even itself), the `IComparable` interface requires that `A.CompareTo(A)` return zero.

# Widening conversions

Depending on your programming language, it might be possible to code a `CompareTo` method where the parameter type has fewer bits (is narrower) than the instance type. This is possible because some programming languages perform an implicit widening conversion that represents the parameter as a type with as many bits as the instance.

For example, suppose the instance type is `Double` and the parameter type is `Int32`. The Microsoft C# compiler generates instructions to represent the value of the parameter as a `Double` object, then generates a `Double.CompareTo(Double)` method that compares the values of the instance and the widened representation of the parameter.

Consult your programming language's documentation to determine if its compiler performs implicit widening conversions of numeric types. For more information, see the [Type Conversion Tables](#) topic.

## Precision in comparisons

The precision of floating-point numbers beyond the documented precision is specific to the implementation and version of .NET. Consequently, a comparison of two particular numbers might change between versions of .NET because the precision of the numbers' internal representation might change.

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Double.Equals method

Article • 01/30/2024

The [Double.Equals\(Double\)](#) method implements the [System.IEquatable<T>](#) interface, and performs slightly better than [Double.Equals\(Object\)](#) because it doesn't have to convert the `obj` parameter to an object.

## Widening conversions

Depending on your programming language, it might be possible to code a [Equals](#) method where the parameter type has fewer bits (is narrower) than the instance type. This is possible because some programming languages perform an implicit widening conversion that represents the parameter as a type with as many bits as the instance.

For example, suppose the instance type is [Double](#) and the parameter type is [Int32](#). The Microsoft C# compiler generates instructions to represent the value of the parameter as a [Double](#) object, then generates a [Double.Equals\(Double\)](#) method that compares the values of the instance and the widened representation of the parameter.

Consult your programming language's documentation to determine if its compiler performs implicit widening conversions of numeric types. For more information, see the [Type Conversion Tables](#) topic.

## Precision in comparisons

The [Equals](#) method should be used with caution, because two apparently equivalent values can be unequal due to the differing precision of the two values. The following example reports that the [Double](#) value `.333333` and the [Double](#) value returned by dividing 1 by 3 are unequal.

C#

```
// Initialize two doubles with apparently identical values
double double1 = .33333;
double double2 = (double) 1/3;
// Compare them for equality
Console.WriteLine(double1.Equals(double2));    // displays false
```

Rather than comparing for equality, one technique involves defining an acceptable relative margin of difference between two values (such as `.001%` of one of the values). If the absolute value of the difference between the two values is less than or equal to that

margin, the difference is likely to be due to differences in precision and, therefore, the values are likely to be equal. The following example uses this technique to compare .33333 and 1/3, the two **Double** values that the previous code example found to be unequal. In this case, the values are equal.

C#

```
// Initialize two doubles with apparently identical values
double double1 = .333333;
double double2 = (double) 1/3;
// Define the tolerance for variation in their values
double difference = Math.Abs(double1 * .00001);

// Compare the values
// The output to the console indicates that the two values are equal
if (Math.Abs(double1 - double2) <= difference)
    Console.WriteLine("double1 and double2 are equal.");
else
    Console.WriteLine("double1 and double2 are unequal.");
```

### ⓘ Note

Because **Epsilon** defines the minimum expression of a positive value whose range is near zero, the margin of difference between two similar values must be greater than **Epsilon**. Typically, it is many times greater than **Epsilon**. Because of this, we recommend that you do not use **Epsilon** when comparing **Double** values for equality.

A second technique involves comparing the difference between two floating-point numbers with some absolute value. If the difference is less than or equal to that absolute value, the numbers are equal. If it is greater, the numbers are not equal. One alternative is to arbitrarily select an absolute value. This is problematic, however, because an acceptable margin of difference depends on the magnitude of the **Double** values. A second alternative takes advantage of a design feature of the floating-point format: The difference between the integer representation of two floating-point values indicates the number of possible floating-point values that separates them. For example, the difference between 0.0 and **Epsilon** is 1, because **Epsilon** is the smallest representable value when working with a **Double** whose value is zero. The following example uses this technique to compare .33333 and 1/3, which are the two **Double** values that the previous code example with the **Equals(Double)** method found to be unequal. Note that the example uses the **BitConverter.DoubleToInt64Bits** method to convert a double-precision floating-point value to its integer representation.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        double value1 = .1 * 10;
        double value2 = 0;
        for (int ctr = 0; ctr < 10; ctr++)
            value2 += .1;

        Console.WriteLine("{0:R} = {1:R}: {2}",
                          value1, value2,
                          HasMinimalDifference(value1, value2, 1));
    }

    public static bool HasMinimalDifference(double value1, double value2, int
units)
    {
        long lValue1 = BitConverter.DoubleToInt64Bits(value1);
        long lValue2 = BitConverter.DoubleToInt64Bits(value2);

        // If the signs are different, return false except for +0 and -0.
        if ((lValue1 >> 63) != (lValue2 >> 63))
        {
            if (value1 == value2)
                return true;

            return false;
        }

        long diff = Math.Abs(lValue1 - lValue2);

        if (diff <= (long) units)
            return true;

        return false;
    }
}

// The example displays the following output:
//      1 = 0.9999999999999989: True
```

The precision of floating-point numbers beyond the documented precision is specific to the implementation and version of the .NET Framework. Consequently, a comparison of two particular numbers might change between versions of the .NET Framework because the precision of the numbers' internal representation might change.

If two `Double.NaN` values are tested for equality by calling the `Equals` method, the method returns `true`. However, if two `NaN` values are tested for equality by using the

equality operator, the operator returns `false`. When you want to determine whether the value of a `Double` is not a number (NaN), an alternative is to call the `IsNaN` method.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

## .NET feedback

.NET is an open source project.  
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Double.Epsilon property

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

The value of the [Epsilon](#) property reflects the smallest positive [Double](#) value that is significant in numeric operations or comparisons when the value of the [Double](#) instance is zero. For example, the following code shows that zero and [Epsilon](#) are considered to be unequal values, whereas zero and half the value of [Epsilon](#) are considered to be equal.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        double[] values = { 0, Double.Epsilon, Double.Epsilon * .5 };

        for (int ctr = 0; ctr <= values.Length - 2; ctr++)
        {
            for (int ctr2 = ctr + 1; ctr2 <= values.Length - 1; ctr2++)
            {
                Console.WriteLine("{0:r} = {1:r}: {2}",
                    values[ctr], values[ctr2],
                    values[ctr].Equals(values[ctr2]));
            }
            Console.WriteLine();
        }
    }
}

// The example displays the following output:
//      0 = 4.94065645841247E-324: False
//      0 = 0: True
//
//      4.94065645841247E-324 = 0: False
```

More precisely, the floating point format consists of a sign, a 52-bit mantissa or significand, and an 11-bit exponent. As the following example shows, zero has an exponent of -1022 and a mantissa of 0. [Epsilon](#) has an exponent of -1022 and a mantissa of 1. This means that [Epsilon](#) is the smallest positive [Double](#) value greater than zero and represents the smallest possible value and the smallest possible increment for a [Double](#) whose exponent is -1022.

C#

```
using System;

public class Example1
{
    public static void Main()
    {
        double[] values = { 0.0, Double.Epsilon };
        foreach (var value in values)
        {
            Console.WriteLine(GetComponentParts(value));
            Console.WriteLine();
        }
    }

    private static string GetComponentParts(double value)
    {
        string result = String.Format("{0:R}: ", value);
        int indent = result.Length;

        // Convert the double to an 8-byte array.
        byte[] bytes = BitConverter.GetBytes(value);
        // Get the sign bit (byte 7, bit 7).
        result += String.Format("Sign: {0}\n",
                               (bytes[7] & 0x80) == 0x80 ? "1 (-)" : "0
(+)");

        // Get the exponent (byte 6 bits 4-7 to byte 7, bits 0-6)
        int exponent = (bytes[7] & 0x07F) << 4;
        exponent = exponent | ((bytes[6] & 0xF0) >> 4);
        int adjustment = exponent != 0 ? 1023 : 1022;
        result += String.Format("{0}Exponent: 0x{1:X4} ({2})\n", new
String(' ', indent), exponent - adjustment);

        // Get the significand (bits 0-51)
        long significand = ((bytes[6] & 0x0F) << 48);
        significand = significand | ((long)bytes[5] << 40);
        significand = significand | ((long)bytes[4] << 32);
        significand = significand | ((long)bytes[3] << 24);
        significand = significand | ((long)bytes[2] << 16);
        significand = significand | ((long)bytes[1] << 8);
        significand = significand | bytes[0];
        result += String.Format("{0}Mantissa: 0x{1:X13}\n", new String(' ', indent), significand);

        return result;
    }
}

//      // The example displays the following output:
//      //      Sign: 0 (+)
//      //      Exponent: 0xFFFFFC02 (-1022)
//      //      Mantissa: 0x0000000000000000
```

```
//          4.94065645841247E-324: Sign: 0 (+)
//                                         Exponent: 0xFFFFFC02 (-1022)
//                                         Mantissa: 0x00000000000001
```

However, the [Epsilon](#) property is not a general measure of precision of the [Double](#) type; it applies only to [Double](#) instances that have a value of zero or an exponent of -1022.

### Note

The value of the [Epsilon](#) property is not equivalent to machine epsilon, which represents the upper bound of the relative error due to rounding in floating-point arithmetic.

The value of this constant is 4.94065645841247e-324.

Two apparently equivalent floating-point numbers might not compare equal because of differences in their least significant digits. For example, the C# expression, `(double)1/3 == (double)0.33333`, does not compare equal because the division operation on the left side has maximum precision while the constant on the right side is precise only to the specified digits. If you create a custom algorithm that determines whether two floating-point numbers can be considered equal, we do not recommend that you base your algorithm on the value of the [Epsilon](#) constant to establish the acceptable absolute margin of difference for the two values to be considered equal. (Typically, that margin of difference is many times greater than [Epsilon](#).) For information about comparing two double-precision floating-point values, see [Double](#) and [Equals\(Double\)](#).

## Platform notes

On ARM systems, the value of the [Epsilon](#) constant is too small to be detected, so it equates to zero. You can define an alternative epsilon value that equals 2.2250738585072014E-308 instead.

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

# System.Int32 struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

[Int32](#) is an immutable value type that represents signed integers with values that range from negative 2,147,483,648 (which is represented by the [Int32.MinValue](#) constant) through positive 2,147,483,647 (which is represented by the [Int32.MaxValue](#) constant). .NET also includes an unsigned 32-bit integer value type, [UInt32](#), which represents values that range from 0 to 4,294,967,295.

## Instantiate an Int32 value

You can instantiate an [Int32](#) value in several ways:

- You can declare an [Int32](#) variable and assign it a literal integer value that is within the range of the [Int32](#) data type. The following example declares two [Int32](#) variables and assigns them values in this way.

C#

```
int number1 = 64301;
int number2 = 25548612;
```

- You can assign the value of an integer type whose range is a subset of the [Int32](#) type. This is a widening conversion that does not require a cast operator in C# or a conversion method in Visual Basic but does require one in F#.

C#

```
sbyte value1 = 124;
short value2 = 1618;

int number1 = value1;
int number2 = value2;
```

- You can assign the value of a numeric type whose range exceeds that of the [Int32](#) type. This is a narrowing conversion, so it requires a cast operator in C# or F#, and a conversion method in Visual Basic if [Option Strict](#) is on. If the numeric value is a [Single](#), [Double](#), or [Decimal](#) value that includes a fractional component, the handling of its fractional part depends on the compiler performing the conversion.

The following example performs narrowing conversions to assign several numeric values to [Int32](#) variables.

```
C#  
  
long lNumber = 163245617;  
try {  
    int number1 = (int) lNumber;  
    Console.WriteLine(number1);  
}  
catch (OverflowException) {  
    Console.WriteLine("{0} is out of range of an Int32.", lNumber);  
}  
  
double dbl2 = 35901.997;  
try {  
    int number2 = (int) dbl2;  
    Console.WriteLine(number2);  
}  
catch (OverflowException) {  
    Console.WriteLine("{0} is out of range of an Int32.", dbl2);  
}  
  
BigInteger bigNumber = 132451;  
try {  
    int number3 = (int) bigNumber;  
    Console.WriteLine(number3);  
}  
catch (OverflowException) {  
    Console.WriteLine("{0} is out of range of an Int32.", bigNumber);  
}  
// The example displays the following output:  
//      163245617  
//      35902  
//      132451
```

- You can call a method of the [Convert](#) class to convert any supported type to an [Int32](#) value. This is possible because [Int32](#) supports the [IConvertible](#) interface. The following example illustrates the conversion of an array of [Decimal](#) values to [Int32](#) values.

```
C#  
  
decimal[] values= { Decimal.MinValue, -1034.23m, -12m, 0m, 147m,  
                   199.55m, 9214.16m, Decimal.MaxValue };  
int result;  
  
foreach (decimal value in values)  
{  
    try {  
        result = Convert.ToInt32(value);  
    }
```

```

        Console.WriteLine("Converted the {0} value '{1}' to the {2} value
{3}.",
                           value.GetType().Name, value,
                           result.GetType().Name, result);
    }
    catch (OverflowException) {
        Console.WriteLine("{0} is outside the range of the Int32 type.",
                           value);
    }
}
// The example displays the following output:
//      -79228162514264337593543950335 is outside the range of the Int32
type.
//      Converted the Decimal value '-1034.23' to the Int32 value -1034.
//      Converted the Decimal value '-12' to the Int32 value -12.
//      Converted the Decimal value '0' to the Int32 value 0.
//      Converted the Decimal value '147' to the Int32 value 147.
//      Converted the Decimal value '199.55' to the Int32 value 200.
//      Converted the Decimal value '9214.16' to the Int32 value 9214.
//      79228162514264337593543950335 is outside the range of the Int32
type.

```

- You can call the [Parse](#) or [TryParse](#) method to convert the string representation of an [Int32](#) value to an [Int32](#). The string can contain either decimal or hexadecimal digits. The following example illustrates the parse operation by using both a decimal and a hexadecimal string.

C#

```

string string1 = "244681";
try {
    int number1 = Int32.Parse(string1);
    Console.WriteLine(number1);
}
catch (OverflowException) {
    Console.WriteLine("{0} is out of range of a 32-bit integer.", string1);
}
catch (FormatException) {
    Console.WriteLine("The format of '{0}' is invalid.", string1);
}

string string2 = "F9A3C";
try {
    int number2 = Int32.Parse(string2,
                               System.Globalization.NumberStyles.HexNumber);
    Console.WriteLine(number2);
}
catch (OverflowException) {
    Console.WriteLine("{0} is out of range of a 32-bit integer.", string2);
}

```

```
        }
        catch (FormatException) {
            Console.WriteLine("The format of '{0}' is invalid.", string2);
        }
        // The example displays the following output:
        //      244681
        //      1022524
```

## Perform operations on Int32 values

The [Int32](#) type supports standard mathematical operations such as addition, subtraction, division, multiplication, negation, and unary negation. Like the other integral types, the [Int32](#) type also supports the bitwise `AND`, `OR`, `XOR`, left shift, and right shift operators.

You can use the standard numeric operators to compare two [Int32](#) values, or you can call the [CompareTo](#) or [Equals](#) method.

You can also call the members of the [Math](#) class to perform a wide range of numeric operations, including getting the absolute value of a number, calculating the quotient and remainder from integral division, determining the maximum or minimum value of two integers, getting the sign of a number, and rounding a number.

## Represent an Int32 as a string

The [Int32](#) type provides full support for standard and custom numeric format strings. (For more information, see [Formatting Types](#), [Standard Numeric Format Strings](#), and [Custom Numeric Format Strings](#).)

To format an [Int32](#) value as an integral string with no leading zeros, you can call the parameterless [ToString\(\)](#) method. By using the "D" format specifier, you can also include a specified number of leading zeros in the string representation. By using the "N" format specifier, you can include group separators and specify the number of decimal digits to appear in the string representation of the number. By using the "X" format specifier, you can represent an [Int32](#) value as a hexadecimal string. The following example formats the elements in an array of [Int32](#) values in these four ways.

C#

```
int[] numbers = { -1403, 0, 169, 1483104 };
foreach (int number in numbers)
{
    // Display value using default formatting.
    Console.WriteLine("{0,-8} --> ", number.ToString());
    // Display value with 3 digits and leading zeros.
```

```

        Console.WriteLine("{0,11:D3}", number);
        // Display value with 1 decimal digit.
        Console.WriteLine("{0,13:N1}", number);
        // Display value as hexadecimal.
        Console.WriteLine("{0,12:X2}", number);
        // Display value with eight hexadecimal digits.
        Console.WriteLine("{0,14:X8}", number);
    }
    // The example displays the following output:
    //    -1403    -->      -1403      -1,403.0      FFFFFA85      FFFFFA85
    //    0        -->       000        0.0          00        00000000
    //    169      -->       169        169.0        A9        000000A9
    //    1483104  -->    1483104    1,483,104.0    16A160    0016A160

```

You can also format an `Int32` value as a binary, octal, decimal, or hexadecimal string by calling the `ToString(Int32, Int32)` method and supplying the base as the method's second parameter. The following example calls this method to display the binary, octal, and hexadecimal representations of an array of integer values.

C#

```

int[] numbers = { -146, 11043, 2781913 };
Console.WriteLine("{0,8}  {1,32}  {2,11}  {3,10}",
                  "Value", "Binary", "Octal", "Hex");
foreach (int number in numbers)
{
    Console.WriteLine("{0,8}  {1,32}  {2,11}  {3,10}",
                      number, Convert.ToString(number, 2),
                      Convert.ToString(number, 8),
                      Convert.ToString(number, 16));
}
// The example displays the following output:
//    Value          Binary          Octal          Hex
//    -146  1111111111111111111111111101101110  37777777556  ffffff6e
//    11043           10101100100011                  25443        2b23
//    2781913         1010100111001011011001      12471331      2a72d9

```

## Work with non-decimal 32-bit integer values

In addition to working with individual integers as decimal values, you may want to perform bitwise operations with integer values, or work with the binary or hexadecimal representations of integer values. `Int32` values are represented in 31 bits, with the thirty-second bit used as a sign bit. Positive values are represented by using sign-and-magnitude representation. Negative values are in two's complement representation. This is important to keep in mind when you perform bitwise operations on `Int32` values or when you work with individual bits. In order to perform a numeric, Boolean, or

comparison operation on any two non-decimal values, both values must use the same representation.

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Int64 struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

[Int64](#) is an immutable value type that represents signed integers with values that range from negative 9,223,372,036,854,775,808 (which is represented by the [Int64.MinValue](#) constant) through positive 9,223,372,036,854,775,807 (which is represented by the [Int64.MaxValue](#) constant). .NET also includes an unsigned 64-bit integer value type, [UInt64](#), which represents values that range from 0 to 18,446,744,073,709,551,615.

## Instantiate an Int64 value

You can instantiate an [Int64](#) value in several ways:

- You can declare an [Int64](#) variable and assign it a literal integer value that is within the range of the [Int64](#) data type. The following example declares two [Int64](#) variables and assigns them values in this way.

C#

```
long number1 = -64301728;
long number2 = 255486129307;
```

- You can assign the value of an integral type whose range is a subset of the [Int64](#) type. This is a widening conversion that does not require a cast operator in C# or a conversion method in Visual Basic. In F#, only the [Int32](#) type can be widened automatically.

C#

```
sbyte value1 = 124;
short value2 = 1618;
int value3 = Int32.MaxValue;

long number1 = value1;
long number2 = value2;
long number3 = value3;
```

- You can assign the value of a numeric type whose range exceeds that of the [Int64](#) type. This is a narrowing conversion, so it requires a cast operator in C# or F# and

a conversion method in Visual Basic if `Option Strict` is on. If the numeric value is a `Single`, `Double`, or `Decimal` value that includes a fractional component, the handling of its fractional part depends on the compiler performing the conversion. The following example performs narrowing conversions to assign several numeric values to `Int64` variables.

C#

```
ulong ulNumber = 163245617943825;
try {
    long number1 = (long) ulNumber;
    Console.WriteLine(number1);
}
catch (OverflowException) {
    Console.WriteLine("{0} is out of range of an Int64.", ulNumber);
}

double dbl2 = 35901.997;
try {
    long number2 = (long) dbl2;
    Console.WriteLine(number2);
}
catch (OverflowException) {
    Console.WriteLine("{0} is out of range of an Int64.", dbl2);
}

BigInteger bigNumber = (BigInteger) 1.63201978555e30;
try {
    long number3 = (long) bigNumber;
    Console.WriteLine(number3);
}
catch (OverflowException) {
    Console.WriteLine("{0} is out of range of an Int64.", bigNumber);
}
// The example displays the following output:
//    163245617943825
//    35902
//    1,632,019,785,549,999,969,612,091,883,520 is out of range of an
Int64.
```

- You can call a method of the `Convert` class to convert any supported type to an `Int64` value. This is possible because `Int64` supports the `IConvertible` interface. The following example illustrates the conversion of an array of `Decimal` values to `Int64` values.

C#

```
decimal[] values= { Decimal.MinValue, -1034.23m, -12m, 0m, 147m,
                    199.55m, 9214.16m, Decimal.MaxValue };
long result;
```

```

foreach (decimal value in values)
{
    try {
        result = Convert.ToInt64(value);
        Console.WriteLine("Converted the {0} value '{1}' to the {2} value
{3}.",
                           value.GetType().Name, value,
                           result.GetType().Name, result);
    }
    catch (OverflowException) {
        Console.WriteLine("{0} is outside the range of the Int64 type.",
                           value);
    }
}
// The example displays the following output:
//      -79228162514264337593543950335 is outside the range of the Int64
type.
//      Converted the Decimal value '-1034.23' to the Int64 value -1034.
//      Converted the Decimal value '-12' to the Int64 value -12.
//      Converted the Decimal value '0' to the Int64 value 0.
//      Converted the Decimal value '147' to the Int64 value 147.
//      Converted the Decimal value '199.55' to the Int64 value 200.
//      Converted the Decimal value '9214.16' to the Int64 value 9214.
//      79228162514264337593543950335 is outside the range of the Int64
type.

```

- You can call the [Parse](#) or [TryParse](#) method to convert the string representation of an [Int64](#) value to an [Int64](#). The string can contain either decimal or hexadecimal digits. The following example illustrates the parse operation by using both a decimal and a hexadecimal string.

C#

```

string string1 = "244681903147";
try {
    long number1 = Int64.Parse(string1);
    Console.WriteLine(number1);
}
catch (OverflowException) {
    Console.WriteLine("{0} is out of range of a 64-bit integer.", string1);
}
catch (FormatException) {
    Console.WriteLine("The format of '{0}' is invalid.", string1);
}

string string2 = "F9A3CFF0A";
try {
    long number2 = Int64.Parse(string2,
                               System.Globalization.NumberStyles.HexNumber);
}

```

```
        Console.WriteLine(number2);
    }
    catch (OverflowException) {
        Console.WriteLine("{0}' is out of range of a 64-bit integer.", string2);
    }
    catch (FormatException) {
        Console.WriteLine("The format of '{0}' is invalid.", string2);
    }
    // The example displays the following output:
    //    244681903147
    //    67012198154
```

## Perform operations on Int64 values

The [Int64](#) type supports standard mathematical operations such as addition, subtraction, division, multiplication, negation, and unary negation. Like the other integral types, the [Int64](#) type also supports the bitwise `AND`, `OR`, `XOR`, left shift, and right shift operators.

You can use the standard numeric operators to compare two [Int64](#) values, or you can call the [CompareTo](#) or [Equals](#) method.

You can also call the members of the [Math](#) class to perform a wide range of numeric operations, including getting the absolute value of a number, calculating the quotient and remainder from integral division, determining the maximum or minimum value of two long integers, getting the sign of a number, and rounding a number.

## Represent an Int64 as a string

The [Int64](#) type provides full support for standard and custom numeric format strings. (For more information, see [Formatting Types](#), [Standard Numeric Format Strings](#), and [Custom Numeric Format Strings](#).)

To format an [Int64](#) value as an integral string with no leading zeros, you can call the parameterless [ToString\(\)](#) method. By using the "D" format specifier, you can also include a specified number of leading zeros in the string representation. By using the "N" format specifier, you can include group separators and specify the number of decimal digits to appear in the string representation of the number. By using the "X" format specifier, you can represent an [Int64](#) value as a hexadecimal string. The following example formats the elements in an array of [Int64](#) values in these four ways.

C#

```
long[] numbers = { -1403, 0, 169, 1483104 };
foreach (var number in numbers)
{
    // Display value using default formatting.
    Console.Write("{0,-8} --> ", number.ToString());
    // Display value with 3 digits and leading zeros.
    Console.Write("{0,8:D3}", number);
    // Display value with 1 decimal digit.
    Console.Write("{0,13:N1}", number);
    // Display value as hexadecimal.
    Console.Write("{0,18:X2}", number);
    // Display value with eight hexadecimal digits.
    Console.WriteLine("{0,18:X8}", number);
}
// The example displays the following output:
//      -1403      -->      -1403      -1,403.0  FFFFFFFFFFFFFFA85
FFFFFFFFFFFFFA85
//      0          -->      000          0.0          00
00000000
//      169        -->      169          169.0        A9
000000A9
//      1483104    -->    1483104  1,483,104.0    16A160
0016A160
```

You can also format an `Int64` value as a binary, octal, decimal, or hexadecimal string by calling the `ToString(Int64, Int32)` method and supplying the base as the method's second parameter. The following example calls this method to display the binary, octal, and hexadecimal representations of an array of integer values.

C#

```
// 2781913 (Base 10):  
//     Binary: 1010100111001011011001  
//     Octal: 12471331  
//     Hex: 2a72d9
```

## Work with non-decimal 32-bit integer values

In addition to working with individual long integers as decimal values, you may want to perform bitwise operations with long integer values, or work with the binary or hexadecimal representations of long integer values. [Int64](#) values are represented in 63 bits, with the sixty-fourth bit used as a sign bit. Positive values are represented by using sign-and-magnitude representation. Negative values are in two's complement representation. This is important to keep in mind when you perform bitwise operations on [Int64](#) values or when you work with individual bits. In order to perform a numeric, Boolean, or comparison operation on any two non-decimal values, both values must use the same representation.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### [.NET feedback](#)

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Numerics.BigInteger struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [BigInteger](#) type is an immutable type that represents an arbitrarily large integer whose value in theory has no upper or lower bounds. The members of the [BigInteger](#) type closely parallel those of other integral types (the [Byte](#), [Int16](#), [Int32](#), [Int64](#), [SByte](#), [UInt16](#), [UInt32](#), and [UInt64](#) types). This type differs from the other integral types in .NET, which have a range indicated by their [MinValue](#) and [MaxValue](#) properties.

## ⓘ Note

Because the [BigInteger](#) type is immutable (see [Mutability](#)) and because it has no upper or lower bounds, an [OutOfMemoryException](#) can be thrown for any operation that causes a [BigInteger](#) value to grow too large.

## Instantiate a BigInteger object

You can instantiate a [BigInteger](#) object in several ways:

- You can use the `new` keyword and provide any integral or floating-point value as a parameter to the [BigInteger](#) constructor. (Floating-point values are truncated before they are assigned to the [BigInteger](#).) The following example illustrates how to use the `new` keyword to instantiate [BigInteger](#) values.

C#

```
BigInteger bigIntFromDouble = new BigInteger(179032.6541);
Console.WriteLine(bigIntFromDouble);
BigInteger bigIntFromInt64 = new BigInteger(934157136952);
Console.WriteLine(bigIntFromInt64);
// The example displays the following output:
// 179032
// 934157136952
```

- You can declare a [BigInteger](#) variable and assign it a value just as you would any numeric type, as long as that value is an integral type. The following example uses assignment to create a [BigInteger](#) value from an [Int64](#).

C#

```
long longValue = 6315489358112;
BigInteger assignedFromLong = longValue;
Console.WriteLine(assignedFromLong);
// The example displays the following output:
// 6315489358112
```

- You can assign a decimal or floating-point value to a [BigInteger](#) object if you cast the value or convert it first. The following example explicitly casts (in C#) or converts (in Visual Basic) a [Double](#) and a [Decimal](#) value to a [BigInteger](#).

C#

```
BigInteger assignedFromDouble = (BigInteger) 179032.6541;
Console.WriteLine(assignedFromDouble);
BigInteger assignedFromDecimal = (BigInteger) 64312.65m;
Console.WriteLine(assignedFromDecimal);
// The example displays the following output:
// 179032
// 64312
```

These methods enable you to instantiate a [BigInteger](#) object whose value is in the range of one of the existing numeric types only. You can instantiate a [BigInteger](#) object whose value can exceed the range of the existing numeric types in one of three ways:

- You can use the `new` keyword and provide a byte array of any size to the [BigInteger\(BigInteger\)](#) constructor. For example:

C#

```
byte[] byteArray = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
BigInteger newBigInt = new BigInteger(byteArray);
Console.WriteLine("The value of newBigInt is {0} (or 0x{0:x}).",
newBigInt);
// The example displays the following output:
// The value of newBigInt is 4759477275222530853130 (or
0x102030405060708090a).
```

- You can call the [Parse](#) or [TryParse](#) methods to convert the string representation of a number to a [BigInteger](#). For example:

C#

```
string positiveString = "91389681247993671255432112000000";
string negativeString = "-90315837410896312071002088037140000";
BigInteger posBigInt = 0;
```

```

BigInteger negBigInt = 0;

try {
    posBigInt = BigInteger.Parse(positiveString);
    Console.WriteLine(posBigInt);
}
catch (FormatException)
{
    Console.WriteLine("Unable to convert the string '{0}' to a
BigInteger value.",
                      positiveString);
}

if (BigInteger.TryParse(negativeString, out negBigInt))
    Console.WriteLine(negBigInt);
else
    Console.WriteLine("Unable to convert the string '{0}' to a
BigInteger value.",
                      negativeString);

// The example displays the following output:
// 9.1389681247993671255432112E+31
// -9.0315837410896312071002088037E+34

```

- You can call a `static` (`Shared` in Visual Basic) `BigInteger` method that performs some operation on a numeric expression and returns a calculated `BigInteger` result. The following example does this by cubing `UInt64.MaxValue` and assigning the result to a `BigInteger`.

C#

```

BigInteger number = BigInteger.Pow(UInt64.MaxValue, 3);
Console.WriteLine(number);
// The example displays the following output:
// 627710173538668076281494232244851025767571854389858533375

```

The uninitialized value of a `BigInteger` is `Zero`.

## Perform operations on `BigInteger` values

You can use a `BigInteger` instance as you would use any other integral type. `BigInteger` overloads the standard numeric operators to enable you to perform basic mathematical operations such as addition, subtraction, division, multiplication, and unary negation. You can also use the standard numeric operators to compare two `BigInteger` values with each other. Like the other integral types, `BigInteger` also supports the bitwise `And`, `Or`, `Xor`, left shift, and right shift operators. For languages that do not support custom operators, the `BigInteger` structure also provides equivalent methods for performing

mathematical operations. These include [Add](#), [Divide](#), [Multiply](#), [Negate](#), [Subtract](#), and several others.

Many members of the [BigInteger](#) structure correspond directly to members of the other integral types. In addition, [BigInteger](#) adds members such as the following:

- [Sign](#), which returns a value that indicates the sign of a [BigInteger](#) value.
- [Abs](#), which returns the absolute value of a [BigInteger](#) value.
- [DivRem](#), which returns both the quotient and remainder of a division operation.
- [GreatestCommonDivisor](#), which returns the greatest common divisor of two [BigInteger](#) values.

Many of these additional members correspond to the members of the [Math](#) class, which provides the functionality to work with the primitive numeric types.

## Mutability

The following example instantiates a [BigInteger](#) object and then increments its value by one.

```
C#
```

```
BigInteger number = BigInteger.Multiply(Int64.MaxValue, 3);
number++;
Console.WriteLine(number);
```

Although this example appears to modify the value of the existing object, this is not the case. [BigInteger](#) objects are immutable, which means that internally, the common language runtime actually creates a new [BigInteger](#) object and assigns it a value one greater than its previous value. This new object is then returned to the caller.

### ⓘ Note

The other numeric types in .NET are also immutable. However, because the [BigInteger](#) type has no upper or lower bounds, its values can grow extremely large and have a measurable impact on performance.

Although this process is transparent to the caller, it does incur a performance penalty. In some cases, especially when repeated operations are performed in a loop on very large [BigInteger](#) values, that performance penalty can be significant. For example, in the

following example, an operation is performed repetitively up to a million times, and a [BigInteger](#) value is incremented by one every time the operation succeeds.

C#

```
BigInteger number = Int64.MaxValue ^ 5;
int repetitions = 1000000;
// Perform some repetitive operation 1 million times.
for (int ctr = 0; ctr <= repetitions; ctr++)
{
    // Perform some operation. If it fails, exit the loop.
    if (!SomeOperationSucceeds()) break;
    // The following code executes if the operation succeeds.
    number++;
}
```

In such a case, you can improve performance by performing all intermediate assignments to an [Int32](#) variable. The final value of the variable can then be assigned to the [BigInteger](#) object when the loop exits. The following example provides an illustration.

C#

```
BigInteger number = Int64.MaxValue ^ 5;
int repetitions = 1000000;
int actualRepetitions = 0;
// Perform some repetitive operation 1 million times.
for (int ctr = 0; ctr <= repetitions; ctr++)
{
    // Perform some operation. If it fails, exit the loop.
    if (!SomeOperationSucceeds()) break;
    // The following code executes if the operation succeeds.
    actualRepetitions++;
}
number += actualRepetitions;
```

## Byte arrays and hexadecimal strings

If you convert [BigInteger](#) values to byte arrays, or if you convert byte arrays to [BigInteger](#) values, you must consider the order of bytes. The [BigInteger](#) structure expects the individual bytes in a byte array to appear in little-endian order (that is, the lower-order bytes of the value precede the higher-order bytes). You can round-trip a [BigInteger](#) value by calling the [ToByteArray](#) method and then passing the resulting byte array to the [BigInteger\(Byte\[\]\)](#) constructor, as the following example shows.

C#

```

BigInteger number = BigInteger.Pow(Int64.MaxValue, 2);
Console.WriteLine(number);

// Write the BigInteger value to a byte array.
byte[] bytes = number.ToByteArray();

// Display the byte array.
foreach (byte byteValue in bytes)
    Console.Write("0x{0:X2} ", byteValue);
Console.WriteLine();

// Restore the BigInteger value from a Byte array.
BigInteger newNumber = new BigInteger(bytes);
Console.WriteLine(newNumber);
// The example displays the following output:
//     8.5070591730234615847396907784E+37
//     0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
0xFF 0x3F
//
//     8.5070591730234615847396907784E+37

```

To instantiate a [BigInteger](#) value from a byte array that represents a value of some other integral type, you can pass the integral value to the [BitConverter.GetBytes](#) method, and then pass the resulting byte array to the [BigInteger\(Byte\[\]\)](#) constructor. The following example instantiates a [BigInteger](#) value from a byte array that represents an [Int16](#) value.

C#

```

short originalValue = 30000;
Console.WriteLine(originalValue);

// Convert the Int16 value to a byte array.
byte[] bytes = BitConverter.GetBytes(originalValue);

// Display the byte array.
foreach (byte byteValue in bytes)
    Console.Write("0x{0} ", byteValue.ToString("X2"));
Console.WriteLine();

// Pass byte array to the BigInteger constructor.
BigInteger number = new BigInteger(bytes);
Console.WriteLine(number);
// The example displays the following output:
//     30000
//     0x30 0x75
//     30000

```

The [BigInteger](#) structure assumes that negative values are stored by using two's complement representation. Because the [BigInteger](#) structure represents a numeric

value with no fixed length, the [BigInteger\(Byte\[\]\)](#) constructor always interprets the most significant bit of the last byte in the array as a sign bit. To prevent the [BigInteger\(Byte\[\]\)](#) constructor from confusing the two's complement representation of a negative value with the sign and magnitude representation of a positive value, positive values in which the most significant bit of the last byte in the byte array would ordinarily be set should include an additional byte whose value is 0. For example, 0xC0 0xBD 0xF0 0xFF is the little-endian hexadecimal representation of either -1,000,000 or 4,293,967,296. Because the most significant bit of the last byte in this array is on, the value of the byte array would be interpreted by the [BigInteger\(Byte\[\]\)](#) constructor as -1,000,000. To instantiate a [BigInteger](#) whose value is positive, a byte array whose elements are 0xC0 0xBD 0xF0 0xFF 0x00 must be passed to the constructor. The following example illustrates this.

C#

```
int negativeNumber = -1000000;
uint positiveNumber = 4293967296;

byte[] negativeBytes = BitConverter.GetBytes(negativeNumber);
BigInteger negativeBigInt = new BigInteger(negativeBytes);
Console.WriteLine(negativeBigInt.ToString("N0"));

byte[] tempPosBytes = BitConverter.GetBytes(positiveNumber);
byte[] positiveBytes = new byte[tempPosBytes.Length + 1];
Array.Copy(tempPosBytes, positiveBytes, tempPosBytes.Length);
BigInteger positiveBigInt = new BigInteger(positiveBytes);
Console.WriteLine(positiveBigInt.ToString("N0"));
// The example displays the following output:
//      -1,000,000
//      4,293,967,296
```

Byte arrays created by the [ToByteArray](#) method from positive values include this extra zero-value byte. Therefore, the [BigInteger](#) structure can successfully round-trip values by assigning them to, and then restoring them from, byte arrays, as the following example shows.

C#

```
BigInteger positiveValue = 15777216;
BigInteger negativeValue = -1000000;

Console.WriteLine("Positive value: " + positiveValue.ToString("N0"));
byte[] bytes = positiveValue.ToByteArray();

foreach (byte byteValue in bytes)
    Console.Write("{0:X2} ", byteValue);
Console.WriteLine();
positiveValue = new BigInteger(bytes);
Console.WriteLine("Restored positive value: " +
```

```

positiveValue.ToString("N0"));

Console.WriteLine();

Console.WriteLine("Negative value: " + negativeValue.ToString("N0"));
bytes = negativeValue.ToByteArray();
foreach (byte byteValue in bytes)
    Console.Write("{0:X2} ", byteValue);
Console.WriteLine();
negativeValue = new BigInteger(bytes);
Console.WriteLine("Restored negative value: " +
negativeValue.ToString("N0"));
// The example displays the following output:
//      Positive value: 15,777,216
//      C0 BD F0 00
//      Restored positive value: 15,777,216
//
//      Negative value: -1,000,000
//      C0 BD F0
//      Restored negative value: -1,000,000

```

However, you may need to add this additional zero-value byte to byte arrays that are created dynamically by the developer or that are returned by methods that convert unsigned integers to byte arrays (such as [BitConverter.GetBytes\(UInt16\)](#), [BitConverter.GetBytes\(UInt32\)](#), and [BitConverter.GetBytes\(UInt64\)](#)).

When parsing a hexadecimal string, the [BigInteger.Parse\(String, NumberStyles\)](#) and [BigInteger.Parse\(String, NumberStyles, IFormatProvider\)](#) methods assume that if the most significant bit of the first byte in the string is set, or if the first hexadecimal digit of the string represents the lower four bits of a byte value, the value is represented by using two's complement representation. For example, both "FF01" and "F01" represent the decimal value -255. To differentiate positive from negative values, positive values should include a leading zero. The relevant overloads of the [ToString](#) method, when they are passed the "X" format string, add a leading zero to the returned hexadecimal string for positive values. This makes it possible to round-trip [BigInteger](#) values by using the [ToString](#) and [Parse](#) methods, as the following example shows.

C#

```

BigInteger negativeNumber = -1000000;
BigInteger positiveNumber = 15777216;

string negativeHex = negativeNumber.ToString("X");
string positiveHex = positiveNumber.ToString("X");

BigInteger negativeNumber2, positiveNumber2;
negativeNumber2 = BigInteger.Parse(negativeHex,
                                   NumberStyles.HexNumber);
positiveNumber2 = BigInteger.Parse(positiveHex,

```

```

        NumberStyles.HexNumber);

Console.WriteLine("Converted {0:N0} to {1} back to {2:N0}.",
                  negativeNumber, negativeHex, negativeNumber2);
Console.WriteLine("Converted {0:N0} to {1} back to {2:N0}.",
                  positiveNumber, positiveHex, positiveNumber2);
// The example displays the following output:
//      Converted -1,000,000 to F0BDC0 back to -1,000,000.
//      Converted 15,777,216 to 0F0BDC0 back to 15,777,216.

```

However, the hexadecimal strings created by calling the `ToString` methods of the other integral types or the overloads of the `ToString` method that include a `toBase` parameter do not indicate the sign of the value or the source data type from which the hexadecimal string was derived. Successfully instantiating a `BigInteger` value from such a string requires some additional logic. The following example provides one possible implementation.

C#

```

using System;
using System.Globalization;
using System.Numerics;

public struct HexValue
{
    public int Sign;
    public string Value;
}

public class ByteHexExample2
{
    public static void Main()
    {
        uint positiveNumber = 4039543321;
        int negativeNumber = -255423975;

        // Convert the numbers to hex strings.
        HexValue hexValue1, hexValue2;
        hexValue1.Value = positiveNumber.ToString("X");
        hexValue1.Sign = Math.Sign(positiveNumber);

        hexValue2.Value = Convert.ToString(negativeNumber, 16);
        hexValue2.Sign = Math.Sign(negativeNumber);

        // Round-trip the hexadecimal values to BigInteger values.
        string hexString;
        BigInteger positiveBigInt, negativeBigInt;

        hexString = (hexValue1.Sign == 1 ? "0" : "") + hexValue1.Value;
        positiveBigInt = BigInteger.Parse(hexString,
NumberStyles.HexNumber);

```

```
        Console.WriteLine("Converted {0} to {1} and back to {2}.",
                           positiveNumber, hexValue1.Value, positiveBigInt);

        hexString = (hexValue2.Sign == 1 ? "0" : "") + hexValue2.Value;
        negativeBigInt = BigInteger.Parse(hexString,
NumberStyles.HexNumber);
        Console.WriteLine("Converted {0} to {1} and back to {2}.",
                           negativeNumber, hexValue2.Value, negativeBigInt);
    }
}

// The example displays the following output:
//      Converted 4039543321 to F0C68A19 and back to 4039543321.
//      Converted -255423975 to f0c68a19 and back to -255423975.
```

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Numerics.Complex struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

A complex number is a number that comprises a real number part and an imaginary number part. A complex number  $z$  is usually written in the form  $z = x + yi$ , where  $x$  and  $y$  are real numbers, and  $i$  is the imaginary unit that has the property  $i^2 = -1$ . The real part of the complex number is represented by  $x$ , and the imaginary part of the complex number is represented by  $y$ .

The [Complex](#) type uses the Cartesian coordinate system (real, imaginary) when instantiating and manipulating complex numbers. A complex number can be represented as a point in a two-dimensional coordinate system, which is known as the complex plane. The real part of the complex number is positioned on the x-axis (the horizontal axis), and the imaginary part is positioned on the y-axis (the vertical axis).

Any point in the complex plane can also be expressed based on its absolute value, by using the polar coordinate system. In polar coordinates, a point is characterized by two numbers:

- Its magnitude, which is the distance of the point from the origin (that is, 0,0, or the point at which the x-axis and the y-axis intersect).
- Its phase, which is the angle between the real axis and the line drawn from the origin to the point.

## Instantiate a complex number

You can assign a value to a complex number in one of the following ways:

- By passing two [Double](#) values to its constructor. The first value represents the real part of the complex number, and the second value represents its imaginary part. These values represent the position of the complex number in the two-dimensional Cartesian coordinate system.
- By calling the static (`Shared` in Visual Basic) [Complex.FromPolarCoordinates](#) method to create a complex number from its polar coordinates.
- By assigning a [Byte](#), [SByte](#), [Int16](#), [UInt16](#), [Int32](#), [UInt32](#), [Int64](#), [UInt64](#), [Single](#), or [Double](#) value to a [Complex](#) object. The value becomes the real part of the complex number, and its imaginary part equals 0.

- By casting (in C#) or converting (in Visual Basic) a [Decimal](#) or [BigInteger](#) value to a [Complex](#) object. The value becomes the real part of the complex number, and its imaginary part equals 0.
- By assigning the complex number that is returned by a method or operator to a [Complex](#) object. For example, [Complex.Add](#) is a static method that returns a complex number that is the sum of two complex numbers, and the [Complex.Addition](#) operator adds two complex numbers and returns the result.

The following example demonstrates each of these five ways of assigning a value to a complex number.

C#

```
using System;
using System.Numerics;

public class CreateEx
{
    public static void Main()
    {
        // Create a complex number by calling its class constructor.
        Complex c1 = new Complex(12, 6);
        Console.WriteLine(c1);

        // Assign a Double to a complex number.
        Complex c2 = 3.14;
        Console.WriteLine(c2);

        // Cast a Decimal to a complex number.
        Complex c3 = (Complex)12.3m;
        Console.WriteLine(c3);

        // Assign the return value of a method to a Complex variable.
        Complex c4 = Complex.Pow(Complex.One, -1);
        Console.WriteLine(c4);

        // Assign the value returned by an operator to a Complex variable.
        Complex c5 = Complex.One + Complex.One;
        Console.WriteLine(c5);

        // Instantiate a complex number from its polar coordinates.
        Complex c6 = Complex.FromPolarCoordinates(10, .524);
        Console.WriteLine(c6);
    }
}

// The example displays the following output:
//      (12, 6)
//      (3.14, 0)
//      (12.3, 0)
//      (1, 0)
```

```
//      (2, 0)
//      (8.65824721882145, 5.00347430269914)
```

## Operations with complex numbers

The [Complex](#) structure in .NET includes members that provide the following functionality:

- Methods to compare two complex numbers to determine whether they are equal.
- Operators to perform arithmetic operations on complex numbers. [Complex](#) operators enable you to perform addition, subtraction, multiplication, division, and unary negation with complex numbers.
- Methods to perform other numerical operations on complex numbers. In addition to the four basic arithmetic operations, you can raise a complex number to a specified power, find the square root of a complex number, and get the absolute value of a complex number.
- Methods to perform trigonometric operations on complex numbers. For example, you can calculate the tangent of an angle represented by a complex number.

Note that, because the [Real](#) and [Imaginary](#) properties are read-only, you cannot modify the value of an existing [Complex](#) object. All methods that perform an operation on a [Complex](#) number, if their return value is of type [Complex](#), return a new [Complex](#) number.

## Precision and complex numbers

The real and imaginary parts of a complex number are represented by two double-precision floating-point values. This means that [Complex](#) values, like double-precision floating-point values, can lose precision as a result of numerical operations. This means that strict comparisons for equality of two [Complex](#) values may fail, even if the difference between the two values is due to a loss of precision. For more information, see [Double](#).

For example, performing exponentiation on the logarithm of a number should return the original number. However, in some cases, the loss of precision of floating-point values can cause slight differences between the two values, as the following example illustrates.

C#

```
Complex value = new Complex(Double.MinValue / 2, Double.MinValue / 2);
Complex value2 = Complex.Exp(Complex.Log(value));
```

```

Console.WriteLine("{0} \n{1} \nEqual: {2}", value, value2,
                  value == value2);
// The example displays the following output:
//      (-8.98846567431158E+307, -8.98846567431158E+307)
//      (-8.98846567431161E+307, -8.98846567431161E+307)
//      Equal: False

```

Similarly, the following example, which calculates the square root of a [Complex](#) number, produces slightly different results on the 32-bit and IA64 versions of .NET.

C#

```

Complex minusOne = new Complex(-1, 0);
Console.WriteLine(Complex.Sqrt(minusOne));
// The example displays the following output:
//      (6.12303176911189E-17, 1) on 32-bit systems.
//      (6.12323399573677E-17,1) on IA64 systems.

```

## Infinity and NaN

The real and imaginary parts of a complex number are represented by [Double](#) values. In addition to ranging from [Double.MinValue](#) to [Double.MaxValue](#), the real or imaginary part of a complex number can have a value of [Double.PositiveInfinity](#), [Double.NegativeInfinity](#), or [Double.NaN](#). [Double.PositiveInfinity](#), [Double.NegativeInfinity](#), and [Double.NaN](#) all propagate in any arithmetic or trigonometric operation.

In the following example, division by [Zero](#) produces a complex number whose real and imaginary parts are both [Double.NaN](#). As a result, performing multiplication with this value also produces a complex number whose real and imaginary parts are [Double.NaN](#). Similarly, performing a multiplication that overflows the range of the [Double](#) type produces a complex number whose real part is [Double.NaN](#) and whose imaginary part is [Double.PositiveInfinity](#). Subsequently performing division with this complex number returns a complex number whose real part is [Double.NaN](#) and whose imaginary part is [Double.PositiveInfinity](#).

C#

```

using System;
using System.Numerics;

public class NaNEx
{
    public static void Main()
    {
        Complex c1 = new Complex(Double.MaxValue / 2, Double.MaxValue / 2);

```

```

        Complex c2 = c1 / Complex.Zero;
        Console.WriteLine(c2.ToString());
        c2 = c2 * new Complex(1.5, 1.5);
        Console.WriteLine(c2.ToString());
        Console.WriteLine();

        Complex c3 = c1 * new Complex(2.5, 3.5);
        Console.WriteLine(c3.ToString());
        c3 = c3 + new Complex(Double.MinValue / 2, Double.MaxValue / 2);
        Console.WriteLine(c3);
    }
}

// The example displays the following output:
//      (NaN, NaN)
//      (NaN, NaN)
//      (NaN, Infinity)
//      (NaN, Infinity)

```

Mathematical operations with complex numbers that are invalid or that overflow the range of the [Double](#) data type do not throw an exception. Instead, they return a [Double.PositiveInfinity](#), [Double.NegativeInfinity](#), or [Double.NaN](#) under the following conditions:

- The division of a positive number by zero returns [Double.PositiveInfinity](#).
- Any operation that overflows the upper bound of the [Double](#) data type returns [Double.PositiveInfinity](#).
- The division of a negative number by zero returns [Double.NegativeInfinity](#).
- Any operation that overflows the lower bound of the [Double](#) data type returns [Double.NegativeInfinity](#).
- The division of a zero by zero returns [Double.NaN](#).
- Any operation that is performed on operands whose values are [Double.PositiveInfinity](#), [Double.NegativeInfinity](#), or [Double.NaN](#) returns [Double.PositiveInfinity](#), [Double.NegativeInfinity](#), or [Double.NaN](#), depending on the specific operation.

Note that this applies to any intermediate calculations performed by a method. For example, the multiplication of `new Complex(9e308, 9e308)` and `new Complex(2.5, 3.5)` uses the formula  $(ac - bd) + (ad + bc)i$ . The calculation of the real component that results from the multiplication evaluates the expression  $9e308 \cdot 2.5 - 9e308 \cdot 3.5$ . Each intermediate multiplication in this expression returns [Double.PositiveInfinity](#), and the attempt to subtract [Double.PositiveInfinity](#) from [Double.PositiveInfinity](#) returns [Double.NaN](#).

## Format a complex number

By default, the string representation of a complex number takes the form `(real, imaginary)`, where *real* and *imaginary* are the string representations of the `Double` values that form the complex number's real and imaginary components. Some overloads of the `ToString` method allow customization of the string representations of these `Double` values to reflect the formatting conventions of a particular culture or to appear in a particular format defined by a standard or custom numeric format string. (For more information, see [Standard Numeric Format Strings](#) and [Custom Numeric Format Strings](#).)

One of the more common ways of expressing the string representation of a complex number takes the form  $a + bi$ , where *a* is the complex number's real component, and *b* is the complex number's imaginary component. In electrical engineering, a complex number is most commonly expressed as  $a + bj$ . You can return the string representation of a complex number in either of these two forms. To do this, define a custom format provider by implementing the `ICustomFormatter` and `IFormatProvider` interfaces, and then call the `String.Format(IFormatProvider, String, Object[])` method.

The following example defines a `ComplexFormatter` class that represents a complex number as a string in the form of either  $a + bi$  or  $a + bj$ .

C#

```
using System;
using System.Numerics;

public class ComplexFormatter : IFormatProvider, ICustomFormatter
{
    public object GetFormat(Type formatType)
    {
        if (formatType == typeof(ICustomFormatter))
            return this;
        else
            return null;
    }

    public string Format(string format, object arg,
                         IFormatProvider provider)
    {
        if (arg is Complex)
        {
            Complex c1 = (Complex)arg;
            // Check if the format string has a precision specifier.
            int precision;
            string fmtString = String.Empty;
            if (format.Length > 1)
            {
                try
                {

```

```
        precision = Int32.Parse(format.Substring(1));
    }
    catch (FormatException)
    {
        precision = 0;
    }
    fmtString = "N" + precision.ToString();
}
if (format.Substring(0, 1).Equals("I",
 StringComparison.OrdinalIgnoreCase))
    return c1.Real.ToString(fmtString) + " + " +
c1.Imaginary.ToString(fmtString) + "i";
else if (format.Substring(0, 1).Equals("J",
 StringComparison.OrdinalIgnoreCase))
    return c1.Real.ToString(fmtString) + " + " +
c1.Imaginary.ToString(fmtString) + "j";
else
    return c1.ToString(format, provider);
}
else
{
    if (arg is IFormattable)
        return ((IFormattable)arg).ToString(format, provider);
    else if (arg != null)
        return arg.ToString();
    else
        return String.Empty;
}
}
```

The following example then uses this custom formatter to display the string representation of a complex number.

C#

```
public class CustomFormatEx
{
    public static void Main()
    {
        Complex c1 = new Complex(12.1, 15.4);
        Console.WriteLine("Formatting with ToString(): " +
                          c1.ToString());
        Console.WriteLine("Formatting with ToString(format): " +
                          c1.ToString("N2"));
        Console.WriteLine("Custom formatting with I0: " +
                          String.Format(new ComplexFormatter(), "{0:I0}",
c1));
        Console.WriteLine("Custom formatting with J3: " +
                          String.Format(new ComplexFormatter(), "{0:J3}",
c1));
    }
}
```

```
// The example displays the following output:  
//   Formatting with ToString(): (12.1, 15.4)  
//   Formatting with ToString(format): (12.10, 15.40)  
//   Custom formatting with I0: 12 + 15i  
//   Custom formatting with J3: 12.100 + 15.400j
```

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Single struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Single](#) value type represents a single-precision 32-bit number with values ranging from negative 3.402823e38 to positive 3.402823e38, as well as positive or negative zero, [PositiveInfinity](#), [NegativeInfinity](#), and not a number ([NaN](#)). It is intended to represent values that are extremely large (such as distances between planets or galaxies) or extremely small (such as the molecular mass of a substance in kilograms) and that often are imprecise (such as the distance from earth to another solar system). The [Single](#) type complies with the IEC 60559:1989 (IEEE 754) standard for binary floating-point arithmetic.

[System.Single](#) provides methods to compare instances of this type, to convert the value of an instance to its string representation, and to convert the string representation of a number to an instance of this type. For information about how format specification codes control the string representation of value types, see [Formatting Types](#), [Standard Numeric Format Strings](#), and [Custom Numeric Format Strings](#).

## Floating-point representation and precision

The [Single](#) data type stores single-precision floating-point values in a 32-bit binary format, as shown in the following table:

[Expand table](#)

Part	Bits
Significand or mantissa	0-22
Exponent	23-30
Sign (0 = positive, 1 = negative)	31

Just as decimal fractions are unable to precisely represent some fractional values (such as 1/3 or [Math.PI](#)), binary fractions are unable to represent some fractional values. For example, 2/10, which is represented precisely by .2 as a decimal fraction, is represented by .0011111001001100 as a binary fraction, with the pattern "1100" repeating to infinity. In this case, the floating-point value provides an imprecise representation of the number that it represents. Performing additional mathematical operations on the original

floating-point value often increases its lack of precision. For example, if you compare the results of multiplying .3 by 10 and adding .3 to .3 nine times, you will see that addition produces the less precise result, because it involves eight more operations than multiplication. Note that this disparity is apparent only if you display the two [Single](#) values by using the "R" [standard numeric format string](#), which, if necessary, displays all 9 digits of precision supported by the [Single](#) type.

C#

```
using System;

public class Example12
{
    public static void Main()
    {
        Single value = .2f;
        Single result1 = value * 10f;
        Single result2 = 0f;
        for (int ctr = 1; ctr <= 10; ctr++)
            result2 += value;

        Console.WriteLine(".2 * 10: {0:R}", result1);
        Console.WriteLine(".2 Added 10 times: {0:R}", result2);
    }
}

// The example displays the following output:
//      .2 * 10: 2
//      .2 Added 10 times: 2.00000024
```

Because some numbers cannot be represented exactly as fractional binary values, floating-point numbers can only approximate real numbers.

All floating-point numbers have a limited number of significant digits, which also determines how accurately a floating-point value approximates a real number. A [Single](#) value has up to 7 decimal digits of precision, although a maximum of 9 digits is maintained internally. This means that some floating-point operations may lack the precision to change a floating-point value. The following example defines a large single-precision floating-point value, and then adds the product of [Single.Epsilon](#) and one quadrillion to it. However, the product is too small to modify the original floating-point value. Its least significant digit is thousandths, whereas the most significant digit in the product is  $10^{-30}$ .

C#

```
using System;

public class Example13
```

```

{
    public static void Main()
    {
        Single value = 123.456f;
        Single additional = Single.Epsilon * 1e15f;
        Console.WriteLine($"{value} + {additional} = {value + additional}");
    }
}
// The example displays the following output:
//    123.456 + 1.401298E-30 = 123.456

```

The limited precision of a floating-point number has several consequences:

- Two floating-point numbers that appear equal for a particular precision might not compare equal because their least significant digits are different. In the following example, a series of numbers are added together, and their total is compared with their expected total. Although the two values appear to be the same, a call to the `Equals` method indicates that they are not.

```

C#
using System;

public class Example9
{
    public static void Main()
    {
        Single[] values = { 10.01f, 2.88f, 2.88f, 2.88f, 9.0f };
        Single result = 27.65f;
        Single total = 0f;
        foreach (var value in values)
            total += value;

        if (total.Equals(result))
            Console.WriteLine("The sum of the values equals the
total.");
        else
            Console.WriteLine("The sum of the values ({0}) does not
equal the total ({1}).",
                            total, result);
    }
}
// The example displays the following output:
//    The sum of the values (27.65) does not equal the total (27.65).
//
// If the index items in the Console.WriteLine statement are changed to
// {0:R},
// the example displays the following output:
//    The sum of the values (27.6500015) does not equal the total
//(27.65).

```

If you change the format items in the `Console.WriteLine(String, Object, Object)` statement from `{0}` and `{1}` to `{0:R}` and `{1:R}` to display all significant digits of the two `Single` values, it is clear that the two values are unequal because of a loss of precision during the addition operations. In this case, the issue can be resolved by calling the `Math.Round(Double, Int32)` method to round the `Single` values to the desired precision before performing the comparison.

- A mathematical or comparison operation that uses a floating-point number might not yield the same result if a decimal number is used, because the binary floating-point number might not equal the decimal number. A previous example illustrated this by displaying the result of multiplying `.3` by `10` and adding `.3` to `.3` nine times.

When accuracy in numeric operations with fractional values is important, use the `Decimal` type instead of the `Single` type. When accuracy in numeric operations with integral values beyond the range of the `Int64` or `UInt64` types is important, use the `BigInteger` type.

- A value might not round-trip if a floating-point number is involved. A value is said to round-trip if an operation converts an original floating-point number to another form, an inverse operation transforms the converted form back to a floating-point number, and the final floating-point number is equal to the original floating-point number. The round trip might fail because one or more least significant digits are lost or changed in a conversion. In the following example, three `Single` values are converted to strings and saved in a file. As the output shows, although the values appear to be identical, the restored values are not equal to the original values.

C#

```
using System;
using System.IO;

public class Example10
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\Singles.dat");
        Single[] values = { 3.2f / 1.11f, 1.0f / 3f, (float)Math.PI };
        for (int ctr = 0; ctr < values.Length; ctr++)
        {
            sw.Write(values[ctr].ToString());
            if (ctr != values.Length - 1)
                sw.Write("|");
        }
        sw.Close();

        Single[] restoredValues = new Single[values.Length];
        StreamReader sr = new StreamReader(@".\Singles.dat");
```

```

        string temp = sr.ReadToEnd();
        string[] tempStrings = temp.Split(' ');
        for (int ctr = 0; ctr < tempStrings.Length; ctr++)
            restoredValues[ctr] = Single.Parse(tempStrings[ctr]);

        for (int ctr = 0; ctr < values.Length; ctr++)
            Console.WriteLine("{0} {2} {1}", values[ctr],
                restoredValues[ctr],
                values[ctr].Equals(restoredValues[ctr]) ?
                    "=" : "<>");
    }
}

// The example displays the following output:
//      2.882883 <> 2.882883
//      0.3333333 <> 0.3333333
//      3.141593 <> 3.141593

```

In this case, the values can be successfully round-tripped by using the "G9" [standard numeric format string](#) to preserve the full precision of `Single` values, as the following example shows.

C#

```

using System;
using System.IO;

public class Example11
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\Singles.dat");
        Single[] values = { 3.2f / 1.11f, 1.0f / 3f, (float)Math.PI };
        for (int ctr = 0; ctr < values.Length; ctr++)
            sw.Write("{0:G9}{1}", values[ctr], ctr < values.Length - 1
                ? " | " : "");

        sw.Close();

        Single[] restoredValues = new Single[values.Length];
        StreamReader sr = new StreamReader(@".\Singles.dat");
        string temp = sr.ReadToEnd();
        string[] tempStrings = temp.Split(' ');
        for (int ctr = 0; ctr < tempStrings.Length; ctr++)
            restoredValues[ctr] = Single.Parse(tempStrings[ctr]);

        for (int ctr = 0; ctr < values.Length; ctr++)
            Console.WriteLine("{0} {2} {1}", values[ctr],
                restoredValues[ctr],
                values[ctr].Equals(restoredValues[ctr]) ?
                    "=" : "<>");
    }
}

```

```
// The example displays the following output:  
//      2.882883 = 2.882883  
//      0.3333333 = 0.3333333  
//      3.141593 = 3.141593
```

- Single values have less precision than Double values. A Single value that is converted to a seemingly equivalent Double often does not equal the Double value because of differences in precision. In the following example, the result of identical division operations is assigned to a Double value and a Single value. After the Single value is cast to a Double, a comparison of the two values shows that they are unequal.

C#

```
using System;  
  
public class Example9  
{  
    public static void Main()  
    {  
        Double value1 = 1 / 3.0;  
        Single sValue2 = 1 / 3.0f;  
        Double value2 = (Double)sValue2;  
        Console.WriteLine("{0:R} = {1:R}: {2}", value1, value2,  
                           value1.Equals(value2));  
    }  
}  
// The example displays the following output:  
//      0.3333333333333331 = 0.333333432674408: False
```

To avoid this problem, either use the Double data type in place of the Single data type, or use the Round method so that both values have the same precision.

## Test for equality

To be considered equal, two Single values must represent identical values. However, because of differences in precision between values, or because of a loss of precision by one or both values, floating-point values that are expected to be identical often turn out to be unequal due to differences in their least significant digits. As a result, calls to the Equals method to determine whether two values are equal, or calls to the CompareTo method to determine the relationship between two Single values, often yield unexpected results. This is evident in the following example, where two apparently equal Single values turn out to be unequal, because the first value has 7 digits of precision, whereas the second value has 9.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        float value1 = .3333333f;
        float value2 = 1.0f/3;
        Console.WriteLine("{0:R} = {1:R}: {2}", value1, value2,
value1.Equals(value2));
    }
}
// The example displays the following output:
//      0.3333333 = 0.333333343: False
```

Calculated values that follow different code paths and that are manipulated in different ways often prove to be unequal. In the following example, one [Single](#) value is squared, and then the square root is calculated to restore the original value. A second [Single](#) is multiplied by 3.51 and squared before the square root of the result is divided by 3.51 to restore the original value. Although the two values appear to be identical, a call to the [Equals\(Single\)](#) method indicates that they are not equal. Using the "G9" standard format string to return a result string that displays all the significant digits of each [Single](#) value shows that the second value is .000000000001 less than the first.

C#

```
using System;

public class Example1
{
    public static void Main()
    {
        float value1 = 10.201438f;
        value1 = (float)Math.Sqrt((float)Math.Pow(value1, 2));
        float value2 = (float)Math.Pow((float)value1 * 3.51f, 2);
        value2 = ((float)Math.Sqrt(value2)) / 3.51f;
        Console.WriteLine("{0} = {1}: {2}\n",
                           value1, value2, value1.Equals(value2));
        Console.WriteLine("{0:G9} = {1:G9}", value1, value2);
    }
}
// The example displays the following output:
//      10.20144 = 10.20144: False
//      10.201438 = 10.2014389
```

In cases where a loss of precision is likely to affect the result of a comparison, you can use the following techniques instead of calling the [Equals](#) or [CompareTo](#) method:

- Call the [Math.Round](#) method to ensure that both values have the same precision. The following example modifies a previous example to use this approach so that two fractional values are equivalent.

C#

```
using System;

public class Example2
{
    public static void Main()
    {
        float value1 = .3333333f;
        float value2 = 1.0f / 3;
        int precision = 7;
        value1 = (float)Math.Round(value1, precision);
        value2 = (float)Math.Round(value2, precision);
        Console.WriteLine("{0:R} = {1:R}: {2}", value1, value2,
value1.Equals(value2));
    }
}
// The example displays the following output:
//      0.3333333 = 0.3333333: True
```

The problem of precision still applies to rounding of midpoint values. For more information, see the [Math.Round\(Double, Int32, MidpointRounding\)](#) method.

- Test for approximate equality instead of equality. This technique requires that you define either an absolute amount by which the two values can differ but still be equal, or that you define a relative amount by which the smaller value can diverge from the larger value.

### ⚠ Warning

**Single.Epsilon** is sometimes used as an absolute measure of the distance between two **Single** values when testing for equality. However, **Single.Epsilon** measures the smallest possible value that can be added to, or subtracted from, a **Single** whose value is zero. For most positive and negative **Single** values, the value of **Single.Epsilon** is too small to be detected. Therefore, except for values that are zero, we do not recommend its use in tests for equality.

The following example uses the latter approach to define an `IsApproximatelyEqual` method that tests the relative difference between two values. It also contrasts the result of calls to the `IsApproximatelyEqual` method and the `Equals(Single)` method.

C#

```
using System;

public class Example3
{
    public static void Main()
    {
        float one1 = .1f * 10;
        float one2 = 0f;
        for (int ctr = 1; ctr <= 10; ctr++)
            one2 += .1f;

        Console.WriteLine("{0:R} = {1:R}: {2}", one1, one2,
one1.Equals(one2));
        Console.WriteLine("{0:R} is approximately equal to {1:R}: {2}",
one1, one2,
IsApproximatelyEqual(one1, one2, .000001f));
    }

    static bool IsApproximatelyEqual(float value1, float value2, float
epsilon)
    {
        // If they are equal anyway, just return True.
        if (value1.Equals(value2))
            return true;

        // Handle NaN, Infinity.
        if (Double.IsInfinity(value1) | Double.IsNaN(value1))
            return value1.Equals(value2);
        else if (Double.IsInfinity(value2) | Double.IsNaN(value2))
            return value1.Equals(value2);

        // Handle zero to avoid division by zero
        double divisor = Math.Max(value1, value2);
        if (divisor.Equals(0))
            divisor = Math.Min(value1, value2);

        return Math.Abs(value1 - value2) / divisor <= epsilon;
    }
}

// The example displays the following output:
//      1 = 1.0000012: False
//      1 is approximately equal to 1.0000012: True
```

## Floating-point values and exceptions

Operations with floating-point values do not throw exceptions, unlike operations with integral types, which throw exceptions in cases of illegal operations such as division by zero or overflow. Instead, in these situations, the result of a floating-point operation is zero, positive infinity, negative infinity, or not a number (NaN):

- If the result of a floating-point operation is too small for the destination format, the result is zero. This can occur when two very small floating-point numbers are multiplied, as the following example shows.

C#

```
using System;

public class Example6
{
    public static void Main()
    {
        float value1 = 1.163287e-36f;
        float value2 = 9.164234e-25f;
        float result = value1 * value2;
        Console.WriteLine("{0} * {1} = {2}", value1, value2, result);
        Console.WriteLine("{0} = 0: {1}", result, result.Equals(0.0f));
    }
}
// The example displays the following output:
//      1.163287E-36 * 9.164234E-25 = 0
//      0 = 0: True
```

- If the magnitude of the result of a floating-point operation exceeds the range of the destination format, the result of the operation is [PositiveInfinity](#) or [NegativeInfinity](#), as appropriate for the sign of the result. The result of an operation that overflows [Single.MaxValue](#) is [PositiveInfinity](#), and the result of an operation that overflows [Single.MinValue](#) is [NegativeInfinity](#), as the following example shows.

C#

```
using System;

public class Example7
{
    public static void Main()
    {
        float value1 = 3.065e35f;
        float value2 = 6.9375e32f;
        float result = value1 * value2;
        Console.WriteLine("PositiveInfinity: {0}",
                          Single.IsPositiveInfinity(result));
        Console.WriteLine("NegativeInfinity: {0}\n",
                          Single.IsNegativeInfinity(result));
    }
}
```

```

        Single.IsNegativeInfinity(result));

        value1 = -value1;
        result = value1 * value2;
        Console.WriteLine("PositiveInfinity: {0}",
                           Single.IsPositiveInfinity(result));
        Console.WriteLine("NegativeInfinity: {0}",
                           Single.IsNegativeInfinity(result));
    }
}

// The example displays the following output:
//      PositiveInfinity: True
//      NegativeInfinity: False
//
//      PositiveInfinity: False
//      NegativeInfinity: True

```

`PositiveInfinity` also results from a division by zero with a positive dividend, and `NegativeInfinity` results from a division by zero with a negative dividend.

- If a floating-point operation is invalid, the result of the operation is `NaN`. For example, `NaN` results from the following operations:
  - Division by zero with a dividend of zero. Note that other cases of division by zero result in either `PositiveInfinity` or `NegativeInfinity`.
  - Any floating-point operation with invalid input. For example, attempting to find the square root of a negative value returns `NaN`.
  - Any operation with an argument whose value is `Single.NaN`.

## Type conversions

The `Single` structure does not define any explicit or implicit conversion operators; instead, conversions are implemented by the compiler.

The following table lists the possible conversions of a value of the other primitive numeric types to a `Single` value. It also indicates whether the conversion is widening or narrowing and whether the resulting `Single` may have less precision than the original value.

[ ] Expand table

Conversion from	Widening/narrowing	Possible loss of precision
<code>Byte</code>	Widening	No

Conversion from	Widening/narrowing	Possible loss of precision
Decimal	Widening  Note that C# requires a cast operator.	Yes. <a href="#">Decimal</a> supports 29 decimal digits of precision; <a href="#">Single</a> supports 9.
Double	Narrowing; out-of-range values are converted to <a href="#">Double.NegativeInfinity</a> or <a href="#">Double.PositiveInfinity</a> .	Yes. <a href="#">Double</a> supports 17 decimal digits of precision; <a href="#">Single</a> supports 9.
Int16	Widening	No
Int32	Widening	Yes. <a href="#">Int32</a> supports 10 decimal digits of precision; <a href="#">Single</a> supports 9.
Int64	Widening	Yes. <a href="#">Int64</a> supports 19 decimal digits of precision; <a href="#">Single</a> supports 9.
SByte	Widening	No
UInt16	Widening	No
UInt32	Widening	Yes. <a href="#">UInt32</a> supports 10 decimal digits of precision; <a href="#">Single</a> supports 9.
UInt64	Widening	Yes. <a href="#">Int64</a> supports 20 decimal digits of precision; <a href="#">Single</a> supports 9.

The following example converts the minimum or maximum value of other primitive numeric types to a [Single](#) value.

```

UInt32.MaxValue,
                UInt64.MinValue, UInt64.MaxValue };

float sngValue;
foreach (var value in values)
{
    if (value.GetType() == typeof(Decimal) ||
        value.GetType() == typeof(Double))
        sngValue = (float)value;
    else
        sngValue = value;
    Console.WriteLine("{0} ({1}) --> {2:R} ({3})",
                      value, value.GetType().Name,
                      sngValue, sngValue.GetType().Name);
}
}

// The example displays the following output:
//      0 (Byte) --> 0 (Single)
//      255 (Byte) --> 255 (Single)
//      -79228162514264337593543950335 (Decimal) --> -7.92281625E+28
(Single)
//      79228162514264337593543950335 (Decimal) --> 7.92281625E+28 (Single)
//      -1.79769313486232E+308 (Double) --> -Infinity (Single)
//      1.79769313486232E+308 (Double) --> Infinity (Single)
//      -32768 (Int16) --> -32768 (Single)
//      32767 (Int16) --> 32767 (Single)
//      -2147483648 (Int32) --> -2.14748365E+09 (Single)
//      2147483647 (Int32) --> 2.14748365E+09 (Single)
//      -9223372036854775808 (Int64) --> -9.223372E+18 (Single)
//      9223372036854775807 (Int64) --> 9.223372E+18 (Single)
//      -128 (SByte) --> -128 (Single)
//      127 (SByte) --> 127 (Single)
//      0 (UInt16) --> 0 (Single)
//      65535 (UInt16) --> 65535 (Single)
//      0 (UInt32) --> 0 (Single)
//      4294967295 (UInt32) --> 4.2949673E+09 (Single)
//      0 (UInt64) --> 0 (Single)
//      18446744073709551615 (UInt64) --> 1.84467441E+19 (Single)

```

In addition, the [Double](#) values [Double.NaN](#), [Double.PositiveInfinity](#), and [Double.NegativeInfinity](#) convert to [Single.NaN](#), [Single.PositiveInfinity](#), and [Single.NegativeInfinity](#), respectively.

Note that the conversion of the value of some numeric types to a [Single](#) value can involve a loss of precision. As the example illustrates, a loss of precision is possible when converting [Decimal](#), [Double](#), [Int32](#), [Int64](#), [UInt32](#), and [UInt64](#) values to [Single](#) values.

The conversion of a [Single](#) value to a [Double](#) is a widening conversion. The conversion may result in a loss of precision if the [Double](#) type does not have a precise representation for the [Single](#) value.

The conversion of a [Single](#) value to a value of any primitive numeric data type other than a [Double](#) is a narrowing conversion and requires a cast operator (in C#) or a conversion method (in Visual Basic). Values that are outside the range of the target data type, which are defined by the target type's `MinValue` and `MaxValue` properties, behave as shown in the following table.

  [Expand table](#)

Target type	Result
Any integral type	An <a href="#">OverflowException</a> exception if the conversion occurs in a checked context. If the conversion occurs in an unchecked context (the default in C#), the conversion operation succeeds but the value overflows.
<a href="#">Decimal</a>	An <a href="#">OverflowException</a> exception,

In addition, [Single.NaN](#), [Single.PositiveInfinity](#), and [Single.NegativeInfinity](#) throw an [OverflowException](#) for conversions to integers in a checked context, but these values overflow when converted to integers in an unchecked context. For conversions to [Decimal](#), they always throw an [OverflowException](#). For conversions to [Double](#), they convert to [Double.NaN](#), [Double.PositiveInfinity](#), and [Double.NegativeInfinity](#), respectively.

Note that a loss of precision may result from converting a [Single](#) value to another numeric type. In the case of converting non-integral [Single](#) values, as the output from the example shows, the fractional component is lost when the [Single](#) value is either rounded (as in Visual Basic) or truncated (as in C# and F#). For conversions to [Decimal](#) values, the [Single](#) value may not have a precise representation in the target data type.

The following example converts a number of [Single](#) values to several other numeric types. The conversions occur in a checked context in Visual Basic (the default), in C# (because of the `checked` keyword), and in F# (because of the `open Checked` statement). The output from the example shows the result for conversions in both a checked and unchecked context. You can perform conversions in an unchecked context in Visual Basic by compiling with the `/removeintchecks+` compiler switch, in C# by commenting out the `checked` statement, and in F# by commenting out the `open Checked` statement.

C#

```
using System;

public class Example5
{
    public static void Main()
```

```

    {
        float[] values = { Single.MinValue, -67890.1234f, -12345.6789f,
                           12345.6789f, 67890.1234f, Single.MaxValue,
                           Single.NaN, Single.PositiveInfinity,
                           Single.NegativeInfinity };
    checked
    {
        foreach (var value in values)
        {
            try
            {
                Int64 lValue = (long)value;
                Console.WriteLine("{0} ({1}) --> {2} (0x{2:X16}) ({3})",
                                  value, value.GetType().Name,
                                  lValue, lValue.GetType().Name);
            }
            catch (OverflowException)
            {
                Console.WriteLine("Unable to convert {0} to Int64.",
value);
            }
            try
            {
                UInt64 ulValue = (ulong)value;
                Console.WriteLine("{0} ({1}) --> {2} (0x{2:X16}) ({3})",
                                  value, value.GetType().Name,
                                  ulValue, ulValue.GetType().Name);
            }
            catch (OverflowException)
            {
                Console.WriteLine("Unable to convert {0} to UInt64.",
value);
            }
            try
            {
                Decimal dValue = (decimal)value;
                Console.WriteLine("{0} ({1}) --> {2} ({3})",
                                  value, value.GetType().Name,
                                  dValue, dValue.GetType().Name);
            }
            catch (OverflowException)
            {
                Console.WriteLine("Unable to convert {0} to Decimal.",
value);
            }
        }

        Double dblValue = value;
        Console.WriteLine("{0} ({1}) --> {2} ({3})",
                          value, value.GetType().Name,
                          dblValue, dblValue.GetType().Name);
        Console.WriteLine();
    }
}
}

```

```
// The example displays the following output for conversions performed
// in a checked context:
//      Unable to convert -3.402823E+38 to Int64.
//      Unable to convert -3.402823E+38 to UInt64.
//      Unable to convert -3.402823E+38 to Decimal.
//      -3.402823E+38 (Single) --> -3.40282346638529E+38 (Double)
//
//      -67890.13 (Single) --> -67890 (0xFFFFFFFFFFFFF6CE) (Int64)
//      Unable to convert -67890.13 to UInt64.
//      -67890.13 (Single) --> -67890.12 (Decimal)
//      -67890.13 (Single) --> -67890.125 (Double)
//
//      -12345.68 (Single) --> -12345 (0xFFFFFFFFFFFFFCFC7) (Int64)
//      Unable to convert -12345.68 to UInt64.
//      -12345.68 (Single) --> -12345.68 (Decimal)
//      -12345.68 (Single) --> -12345.6787109375 (Double)
//
//      12345.68 (Single) --> 12345 (0x00000000000003039) (Int64)
//      12345.68 (Single) --> 12345 (0x00000000000003039) (UInt64)
//      12345.68 (Single) --> 12345.68 (Decimal)
//      12345.68 (Single) --> 12345.6787109375 (Double)
//
//      67890.13 (Single) --> 67890 (0x00000000000010932) (Int64)
//      67890.13 (Single) --> 67890 (0x00000000000010932) (UInt64)
//      67890.13 (Single) --> 67890.12 (Decimal)
//      67890.13 (Single) --> 67890.125 (Double)
//
//      Unable to convert 3.402823E+38 to Int64.
//      Unable to convert 3.402823E+38 to UInt64.
//      Unable to convert 3.402823E+38 to Decimal.
//      3.402823E+38 (Single) --> 3.40282346638529E+38 (Double)
//
//      Unable to convert NaN to Int64.
//      Unable to convert NaN to UInt64.
//      Unable to convert NaN to Decimal.
//      NaN (Single) --> NaN (Double)
//
//      Unable to convert Infinity to Int64.
//      Unable to convert Infinity to UInt64.
//      Unable to convert Infinity to Decimal.
//      Infinity (Single) --> Infinity (Double)
//
//      Unable to convert -Infinity to Int64.
//      Unable to convert -Infinity to UInt64.
//      Unable to convert -Infinity to Decimal.
//      -Infinity (Single) --> -Infinity (Double)
// The example displays the following output for conversions performed
// in an unchecked context:
//      -3.402823E+38 (Single) --> -9223372036854775808
//(0x8000000000000000) (Int64)
//      -3.402823E+38 (Single) --> 9223372036854775808 (0x8000000000000000)
(UInt64)
//      Unable to convert -3.402823E+38 to Decimal.
//      -3.402823E+38 (Single) --> -3.40282346638529E+38 (Double)
//
```

```

//      -67890.13 (Single) --> -67890 (0xFFFFFFFFFFFFFEF6CE) (Int64)
//      -67890.13 (Single) --> 18446744073709483726 (0xFFFFFFFFFFFFFEF6CE)
(UInt64)
//      -67890.13 (Single) --> -67890.12 (Decimal)
//      -67890.13 (Single) --> -67890.125 (Double)
//
//      -12345.68 (Single) --> -12345 (0xFFFFFFFFFFFFCFC7) (Int64)
//      -12345.68 (Single) --> 18446744073709539271 (0xFFFFFFFFFFFFCFC7)
(UInt64)
//      -12345.68 (Single) --> -12345.68 (Decimal)
//      -12345.68 (Single) --> -12345.6787109375 (Double)
//
//      12345.68 (Single) --> 12345 (0x0000000000003039) (Int64)
//      12345.68 (Single) --> 12345 (0x0000000000003039) (UInt64)
//      12345.68 (Single) --> 12345.68 (Decimal)
//      12345.68 (Single) --> 12345.6787109375 (Double)
//
//      67890.13 (Single) --> 67890 (0x0000000000010932) (Int64)
//      67890.13 (Single) --> 67890 (0x0000000000010932) (UInt64)
//      67890.13 (Single) --> 67890.12 (Decimal)
//      67890.13 (Single) --> 67890.125 (Double)
//
//      3.402823E+38 (Single) --> -9223372036854775808 (0x8000000000000000)
(Int64)
//      3.402823E+38 (Single) --> 0 (0x0000000000000000) (UInt64)
//      Unable to convert 3.402823E+38 to Decimal.
//      3.402823E+38 (Single) --> 3.40282346638529E+38 (Double)
//
//      NaN (Single) --> -9223372036854775808 (0x8000000000000000) (Int64)
//      NaN (Single) --> 0 (0x0000000000000000) (UInt64)
//      Unable to convert NaN to Decimal.
//      NaN (Single) --> NaN (Double)
//
//      Infinity (Single) --> -9223372036854775808 (0x8000000000000000)
(Int64)
//      Infinity (Single) --> 0 (0x0000000000000000) (UInt64)
//      Unable to convert Infinity to Decimal.
//      Infinity (Single) --> Infinity (Double)
//
//      -Infinity (Single) --> -9223372036854775808 (0x8000000000000000)
(Int64)
//      -Infinity (Single) --> 9223372036854775808 (0x8000000000000000)
(UInt64)
//      Unable to convert -Infinity to Decimal.
//      -Infinity (Single) --> -Infinity (Double)

```

For more information on the conversion of numeric types, see [Type Conversion in .NET](#) and [Type Conversion Tables](#).

## Floating-point functionality

The [Single](#) structure and related types provide methods to perform the following categories of operations:

- **Comparison of values.** You can call the [Equals](#) method to determine whether two [Single](#) values are equal, or the [CompareTo](#) method to determine the relationship between two values.

The [Single](#) structure also supports a complete set of comparison operators. For example, you can test for equality or inequality, or determine whether one value is greater than or equal to another value. If one of the operands is a [Double](#), the [Single](#) value is converted to a [Double](#) before performing the comparison. If one of the operands is an integral type, it is converted to a [Single](#) before performing the comparison. Although these are widening conversions, they may involve a loss of precision.

#### **Warning**

Because of differences in precision, two [Single](#) values that you expect to be equal may turn out to be unequal, which affects the result of the comparison. See the [Test for equality](#) section for more information about comparing two [Single](#) values.

You can also call the [IsNaN](#), [IsInfinity](#), [IsPositiveInfinity](#), and [IsNegativeInfinity](#) methods to test for these special values.

- **Mathematical operations.** Common arithmetic operations such as addition, subtraction, multiplication, and division are implemented by language compilers and Common Intermediate Language (CIL) instructions rather than by [Single](#) methods. If the other operand in a mathematical operation is a [Double](#), the [Single](#) is converted to a [Double](#) before performing the operation, and the result of the operation is also a [Double](#) value. If the other operand is an integral type, it is converted to a [Single](#) before performing the operation, and the result of the operation is also a [Single](#) value.

You can perform other mathematical operations by calling `static` (Shared in Visual Basic) methods in the [System.Math](#) class. These include additional methods commonly used for arithmetic (such as [Math.Abs](#), [Math.Sign](#), and [Math.Sqrt](#)), geometry (such as [Math.Cos](#) and [Math.Sin](#)), and calculus (such as [Math.Log](#)). In all cases, the [Single](#) value is converted to a [Double](#).

You can also manipulate the individual bits in a [Single](#) value. The [BitConverter.GetBytes\(Single\)](#) method returns its bit pattern in a byte array. By

passing that byte array to the [BitConverter.ToInt32](#) method, you can also preserve the [Single](#) value's bit pattern in a 32-bit integer.

- **Rounding.** Rounding is often used as a technique for reducing the impact of differences between values caused by problems of floating-point representation and precision. You can round a [Single](#) value by calling the [Math.Round](#) method. However, note that the [Single](#) value is converted to a [Double](#) before the method is called, and the conversion can involve a loss of precision.
- **Formatting.** You can convert a [Single](#) value to its string representation by calling the [ToString](#) method or by using the [composite formatting](#) feature. For information about how format strings control the string representation of floating-point values, see the [Standard Numeric Format Strings](#) and [Custom Numeric Format Strings](#) topics.
- **Parsing strings.** You can convert the string representation of a floating-point value to a [Single](#) value by calling the [Parse](#) or [TryParse](#) method. If the parse operation fails, the [Parse](#) method throws an exception, whereas the [TryParse](#) method returns `false`.
- **Type conversion.** The [Single](#) structure provides an explicit interface implementation for the [IConvertible](#) interface, which supports conversion between any two standard .NET data types. Language compilers also support the implicit conversion of values for all other standard numeric types except for the conversion of [Double](#) to [Single](#) values. Conversion of a value of any standard numeric type other than a [Double](#) to a [Single](#) is a widening conversion and does not require the use of a casting operator or conversion method.

However, conversion of 32-bit and 64-bit integer values can involve a loss of precision. The following table lists the differences in precision for 32-bit, 64-bit, and [Double](#) types:

[ ] [Expand table](#)

Type	Maximum precision (in decimal digits)	Internal precision (in decimal digits)
<a href="#">Double</a>	15	17
<a href="#">Int32</a> and <a href="#">UInt32</a>	10	10
<a href="#">Int64</a> and <a href="#">UInt64</a>	19	19

Type	Maximum precision (in decimal digits)	Internal precision (in decimal digits)
Single	7	9

The problem of precision most frequently affects [Single](#) values that are converted to [Double](#) values. In the following example, two values produced by identical division operations are unequal, because one of the values is a single-precision floating point value that is converted to a [Double](#).

C#

```
using System;

public class Example8
{
    public static void Main()
    {
        Double value1 = 1 / 3.0;
        Single sValue2 = 1 / 3.0f;
        Double value2 = (Double)sValue2;
        Console.WriteLine("{0:R} = {1:R}: {2}",
                           value1, value2,
                           value1.Equals(value2));
    }
}
// The example displays the following output:
//      0.3333333333333331 = 0.3333333432674408: False
```

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

# System.Single.CompareTo methods

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

Values must be identical to be considered equal. Particularly when floating-point values depend on multiple mathematical operations, it's common for them to lose precision and for their values to be nearly identical except for their least significant digits. Because of this, the return value of the [CompareTo](#) method may seem surprising at times. For example, multiplication by a particular value followed by division by the same value should produce the original value, but in the following example, the computed value turns out to be greater than the original value. Showing all significant digits of the two values by using the "R" [standard numeric format string](#) indicates that the computed value differs from the original value in its least significant digits. For information about handling such comparisons, see the Remarks section of the [Equals\(Single\)](#) method.

Although an object whose value is [NaN](#) is not considered equal to another object whose value is [NaN](#) (even itself), the [IComparable<T>](#) interface requires that `A.CompareTo(A)` return zero.

## CompareTo(System.Object)

The `value` parameter must be `null` or an instance of [Single](#); otherwise, an exception is thrown. Any instance of [Single](#), regardless of its value, is considered greater than `null`.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        float value1 = 16.5457f;
        float operand = 3.8899982f;
        object value2 = value1 * operand / operand;
        Console.WriteLine("Comparing {0} and {1}: {2}\n",
                           value1, value2, value1.CompareTo(value2));
        Console.WriteLine("Comparing {0:R} and {1:R}: {2}",
                           value1, value2, value1.CompareTo(value2));
    }
}

// The example displays the following output:
//      Comparing 16.5457 and 16.5457: -1
```

```
//  
//      Comparing 16.5457 and 16.545702: -1
```

This method is implemented to support the [IComparable](#) interface.

## CompareTo(System.Single)

This method implements the [System.IComparable<T>](#) interface and performs slightly better than the [Single.CompareTo\(Object\)](#) overload because it doesn't have to convert the `value` parameter to an object.

C#

```
using System;  
  
public class Example2  
{  
    public static void Main()  
    {  
        float value1 = 16.5457f;  
        float operand = 3.8899982f;  
        float value2 = value1 * operand / operand;  
        Console.WriteLine("Comparing {0} and {1}: {2}\n",
                           value1, value2, value1.CompareTo(value2));  
        Console.WriteLine("Comparing {0:R} and {1:R}: {2}",
                           value1, value2, value1.CompareTo(value2));  
    }  
}  
// The example displays the following output:  
//      Comparing 16.5457 and 16.5457: -1  
//  
//      Comparing 16.5457 and 16.545702: -1
```

## Widening conversions

Depending on your programming language, it might be possible to code a [CompareTo](#) method where the parameter type has fewer bits (is narrower) than the instance type. This is possible because some programming languages perform an implicit widening conversion that represents the parameter as a type with as many bits as the instance.

For example, suppose the instance type is [Single](#) and the parameter type is [Int32](#). The Microsoft C# compiler generates instructions to represent the value of the parameter as a [Single](#) object, then generates a [Single.CompareTo\(Single\)](#) method that compares the values of the instance and the widened representation of the parameter.

Consult your programming language's documentation to determine if its compiler performs implicit widening conversions of numeric types. For more information, see the [Type Conversion Tables](#) topic.

## Precision in comparisons

The precision of floating-point numbers beyond the documented precision is specific to the implementation and version of .NET. Consequently, a comparison of two particular numbers might change between versions of .NET because the precision of the numbers' internal representation might change.

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Single.Epsilon property

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

The value of the [Epsilon](#) property reflects the smallest positive [Single](#) value that is significant in numeric operations or comparisons when the value of the [Single](#) instance is zero. For example, the following code shows that zero and [Epsilon](#) are considered to be unequal values, whereas zero and half the value of [Epsilon](#) are considered to be equal.

C#

```
using System;

public class Example1
{
    public static void Main()
    {
        float[] values = { 0f, Single.Epsilon, Single.Epsilon * .5f };

        for (int ctr = 0; ctr <= values.Length - 2; ctr++)
        {
            for (int ctr2 = ctr + 1; ctr2 <= values.Length - 1; ctr2++)
            {
                Console.WriteLine("{0:r} = {1:r}: {2}",
                    values[ctr], values[ctr2],
                    values[ctr].Equals(values[ctr2]));
            }
            Console.WriteLine();
        }
    }
}

// The example displays the following output:
//      0 = 1.401298E-45: False
//      0 = 0: True
//
//      1.401298E-45 = 0: False
```

More precisely, the single-precision floating-point format consists of a sign, a 23-bit mantissa or significand, and an 8-bit exponent. As the following example shows, zero has an exponent of -126 and a mantissa of 0. [Epsilon](#) has an exponent of -126 and a mantissa of 1. This means that [Single.Epsilon](#) is the smallest positive [Single](#) value that is greater than zero and represents the smallest possible value and the smallest possible increment for a [Single](#) whose exponent is -126.

C#

```
using System;

public class Example2
{
    public static void Main()
    {
        float[] values = { 0.0f, Single.Epsilon };
        foreach (var value in values) {
            Console.WriteLine(GetComponentParts(value));
            Console.WriteLine();
        }
    }

    private static string GetComponentParts(float value)
    {
        string result = String.Format("{0:R}: ", value);
        int indent = result.Length;

        // Convert the single to a 4-byte array.
        byte[] bytes = BitConverter.GetBytes(value);
        int formattedSingle = BitConverter.ToInt32(bytes, 0);

        // Get the sign bit (byte 3, bit 7).
        result += String.Format("Sign: {0}\n",
                               (formattedSingle >> 31) != 0 ? "1 (-)" : "0
(+)");

        // Get the exponent (byte 2 bit 7 to byte 3, bits 6)
        int exponent = (formattedSingle >> 23) & 0x000000FF;
        int adjustment = (exponent != 0) ? 127 : 126;
        result += String.Format("{0}Exponent: 0x{1:X4} ({1})\n", new String('
', indent), exponent - adjustment);

        // Get the significand (bits 0-22)
        long significand = exponent != 0 ?
                           ((formattedSingle & 0x007FFFFFF) | 0x800000) :
                           (formattedSingle & 0x007FFFFFF);
        result += String.Format("{0}Mantissa: 0x{1:X13}\n", new String('
', indent), significand);
    }
}

//      // The example displays the following output:
//      0: Sign: 0 (+)
//          Exponent: 0xFFFFF82 (-126)
//          Mantissa: 0x00000000000000
//
//      1.401298E-45: Sign: 0 (+)
//          Exponent: 0xFFFFF82 (-126)
//          Mantissa: 0x00000000000001
```

However, the [Epsilon](#) property is not a general measure of precision of the [Single](#) type; it applies only to [Single](#) instances that have a value of zero.

### Note

The value of the [Epsilon](#) property is not equivalent to machine epsilon, which represents the upper bound of the relative error due to rounding in floating-point arithmetic.

The value of this constant is 1.4e-45.

Two apparently equivalent floating-point numbers might not compare equal because of differences in their least significant digits. For example, the C# expression, `(float)1/3 == (float)0.33333`, does not compare equal because the division operation on the left side has maximum precision while the constant on the right side is precise only to the specified digits. If you create a custom algorithm that determines whether two floating-point numbers can be considered equal, you must use a value that is greater than the [Epsilon](#) constant to establish the acceptable absolute margin of difference for the two values to be considered equal. (Typically, that margin of difference is many times greater than [Epsilon](#).)

## Platform notes

On ARM systems, the value of the [Epsilon](#) constant is too small to be detected, so it equates to zero. You can define an alternative epsilon value that equals 1.175494351E-38 instead.

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.Single.Equals method

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Single.Equals\(Single\)](#) method implements the [System.IEquatable<T>](#) interface, and performs slightly better than [Single.Equals\(Object\)](#) because it does not have to convert the `obj` parameter to an object.

## Widening conversions

Depending on your programming language, it might be possible to code an [Equals](#) method where the parameter type has fewer bits (is narrower) than the instance type. This is possible because some programming languages perform an implicit widening conversion that represents the parameter as a type with as many bits as the instance.

For example, suppose the instance type is [Single](#) and the parameter type is [Int32](#). The Microsoft C# compiler generates instructions to represent the value of the parameter as a [Single](#) object, and then generates a [Single.Equals\(Single\)](#) method that compares the values of the instance and the widened representation of the parameter.

Consult your programming language's documentation to determine if its compiler performs implicit widening conversions of numeric types. For more information, see [Type Conversion Tables](#).

## Precision in comparisons

The [Equals](#) method should be used with caution, because two apparently equivalent values can be unequal because of the differing precision of the two values. The following example reports that the [Single](#) value `.3333` and the [Single](#) returned by dividing 1 by 3 are unequal.

C#

```
// Initialize two floats with apparently identical values
float float1 = .33333f;
float float2 = 1/3;
// Compare them for equality
Console.WriteLine(float1.Equals(float2));    // displays false
```

One comparison technique that avoids the problems associated with comparing for equality involves defining an acceptable margin of difference between two values (such as .01% of one of the values). If the absolute value of the difference between the two values is less than or equal to that margin, the difference is likely to be an outcome of differences in precision and, therefore, the values are likely to be equal. The following example uses this technique to compare .33333 and 1/3, which are the two [Single](#) values that the previous code example found to be unequal.

C#

```
// Initialize two floats with apparently identical values
float float1 = .33333f;
float float2 = (float) 1/3;
// Define the tolerance for variation in their values
float difference = Math.Abs(float1 * .0001f);

// Compare the values
// The output to the console indicates that the two values are equal
if (Math.Abs(float1 - float2) <= difference)
    Console.WriteLine("float1 and float2 are equal.");
else
    Console.WriteLine("float1 and float2 are unequal.");
```

In this case, the values are equal.

### ① Note

Because [Epsilon](#) defines the minimum expression of a positive value whose range is near zero, the margin of difference must be greater than [Epsilon](#). Typically, it is many times greater than [Epsilon](#). Because of this, we recommend that you do not use [Epsilon](#) when comparing [Double](#) values for equality.

A second technique that avoids the problems associated with comparing for equality involves comparing the difference between two floating-point numbers with some absolute value. If the difference is less than or equal to that absolute value, the numbers are equal. If it is greater, the numbers are not equal. One way to do this is to arbitrarily select an absolute value. However, this is problematic, because an acceptable margin of difference depends on the magnitude of the [Single](#) values. A second way takes advantage of a design feature of the floating-point format: The difference between the mantissa components in the integer representations of two floating-point values indicates the number of possible floating-point values that separates the two values. For example, the difference between 0.0 and [Epsilon](#) is 1, because [Epsilon](#) is the smallest representable value when working with a [Single](#) whose value is zero. The following example uses this technique to compare .33333 and 1/3, which are the two [Double](#)

values that the previous code example with the `Equals(Single)` method found to be unequal. Note that the example uses the `BitConverter.GetBytes` and `BitConverter.ToInt32` methods to convert a single-precision floating-point value to its integer representation.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        float value1 = .1f * 10f;
        float value2 = 0f;
        for (int ctr = 0; ctr < 10; ctr++)
            value2 += .1f;

        Console.WriteLine("{0:R} = {1:R}: {2}",
            value1, value2,
            HasMinimalDifference(value1, value2, 1));
    }

    public static bool HasMinimalDifference(float value1, float value2, int
units)
    {
        byte[] bytes = BitConverter.GetBytes(value1);
        int iValue1 = BitConverter.ToInt32(bytes, 0);

        bytes = BitConverter.GetBytes(value2);
        int iValue2 = BitConverter.ToInt32(bytes, 0);

        // If the signs are different, return false except for +0 and -0.
        if ((iValue1 >> 31) != (iValue2 >> 31))
        {
            if (value1 == value2)
                return true;

            return false;
        }

        int diff = Math.Abs(iValue1 - iValue2);

        if (diff <= units)
            return true;

        return false;
    }
}

// The example displays the following output:
//      1 = 1.00000012: True
```

The precision of floating-point numbers beyond the documented precision is specific to the implementation and version of .NET. Consequently, a comparison of two numbers

might produce different results depending on the version of .NET, because the precision of the numbers' internal representation might change.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

## **.NET feedback**

.NET is an open source project.  
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Dates, times, and time zones

Article • 01/03/2023

In addition to the basic [DateTime](#) structure, .NET provides the following classes that support working with time zones:

- [TimeZone](#)

Use this class to work with the system's local time zone and the Coordinated Universal Time (UTC) zone. The functionality of the [TimeZone](#) class is largely superseded by the [TimeZoneInfo](#) class.

- [TimeZoneInfo](#)

Use this class to work with any time zone that is predefined on a system, to create new time zones, and to easily convert dates and times from one time zone to another. For new development, use the [TimeZoneInfo](#) class instead of the [TimeZone](#) class.

- [DateTimeOffset](#)

Use this structure to work with dates and times whose offset (or difference) from UTC is known. The [DateTimeOffset](#) structure combines a date and time value with that time's offset from UTC. Because of its relationship to UTC, an individual date and time value unambiguously identifies a single point in time. This makes a [DateTimeOffset](#) value more portable from one computer to another than a [DateTime](#) value.

Starting with .NET 6, the following types are available:

- [DateOnly](#)

Use this structure when working with a value that only represents a date. The date represents the entire day, from the start of the day to the end. [DateOnly](#) has a range of `0001-01-01` through `9999-12-31`. And, this type represents the month, day, and year combination without a specific time. If you previously used a [DateTime](#) type in your code to represent a date that disregarded the time, use this type in its place. For more information, see [How to use the DateOnly and TimeOnly structures](#).

- [TimeOnly](#)

Use this structure to represent a time without a date. The time represents the hours, minutes, and seconds of a non-specific day. `TimeOnly` has a range of `00:00:00.0000000` to `23:59:59.9999999`. This type can be used to replace `DateTime` and `TimeSpan` types in your code when you used those types to represent a time. For more information, see [How to use the DateOnly and TimeOnly structures](#).

The next section provides the information that you need to work with time zones and to create time zone-aware applications that can convert dates and times from one time zone to another.

## In this section

### [Time zone overview](#)

Discusses the terminology, concepts, and issues involved in creating time zone-aware applications.

### [Choosing between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo](#)

Discusses when to use the `DateTime`, `DateTimeOffset`, and `TimeZoneInfo` types when working with date and time data.

### [Finding the time zones defined on a local system](#)

Describes how to enumerate the time zones found on a local system.

### [How to: Enumerate time zones present on a computer](#)

Provides examples that enumerate the time zones defined in a computer's registry and that let users select a predefined time zone from a list.

### [How to: Access the predefined UTC and local time zone objects](#)

Describes how to access Coordinated Universal Time and the local time zone.

### [How to: Instantiate a TimeZoneInfo object](#)

Describes how to instantiate a `TimeZoneInfo` object from the local system registry.

### [Instantiating a DateTimeOffset object](#)

Discusses the ways in which a `DateTimeOffset` object can be instantiated, and the ways in which a `DateTime` value can be converted to a `DateTimeOffset` value.

### [How to: Create time zones without adjustment rules](#)

Describes how to create a custom time zone that does not support the transition to and from daylight saving time.

### [How to: Create time zones with adjustment rules](#)

Describes how to create a custom time zone that supports one or more transitions to

and from daylight saving time.

### [Saving and restoring time zones](#)

Describes [TimeZoneInfo](#) support for serialization and deserialization of time zone data and illustrates some of the scenarios in which these features can be used.

### [How to: Save time zones to an embedded resource](#)

Describes how to create a custom time zone and save its information in a resource file.

### [How to: Restore time zones from an embedded resource](#)

Describes how to instantiate custom time zones that have been saved to an embedded resource file.

### [Performing arithmetic operations with dates and times](#)

Discusses the issues involved in adding, subtracting, and comparing [DateTime](#) and [DateTimeOffset](#) values.

### [How to: Use time zones in date and time arithmetic](#)

Discusses how to perform date and time arithmetic that reflects a time zone's adjustment rules.

### [Converting between DateTime and DateTimeOffset](#)

Describes how to convert between [DateTime](#) and [DateTimeOffset](#) values.

### [Converting times between time zones](#)

Describes how to convert times from one time zone to another.

### [How to: Resolve ambiguous times](#)

Describes how to resolve an ambiguous time by mapping it to the time zone's standard time.

### [How to: Let users resolve ambiguous times](#)

Describes how to let a user determine the mapping between an ambiguous local time and Coordinated Universal Time.

## Reference

### [System.TimeZoneInfo](#)



Collaborate with us on  
GitHub



.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Choose between `DateTime`, `DateOnly`, `DateTimeOffset`, `TimeSpan`, `TimeOnly`, and `TimeZoneInfo`

Article • 05/02/2023

.NET applications can use date and time information in several ways. The more common uses of date and time information include:

- To reflect a date only, so that time information is not important.
- To reflect a time only, so that date information is not important.
- To reflect an abstract date and time that's not tied to a specific time and place (for example, most stores in an international chain open on weekdays at 9:00 A.M.).
- To retrieve date and time information from sources outside of .NET, typically where date and time information is stored in a simple data type.
- To uniquely and unambiguously identify a single point in time. Some applications require that a date and time be unambiguous only on the host system. Other apps require that it be unambiguous across systems (that is, a date serialized on one system can be meaningfully deserialized and used on another system anywhere in the world).
- To preserve multiple related times (such as the requester's local time and the server's time of receipt for a web request).
- To perform date and time arithmetic, possibly with a result that uniquely and unambiguously identifies a single point in time.

.NET includes the [DateTime](#), [DateOnly](#), [DateTimeOffset](#), [TimeSpan](#), [TimeOnly](#), and [TimeZoneInfo](#) types, all of which can be used to build applications that work with dates and times.

## ⓘ Note

This article doesn't discuss `TimeZone` because its functionality is almost entirely incorporated in the `TimeZoneInfo` class. Whenever possible, use the `TimeZoneInfo` class instead of the `TimeZone` class.

## The `DateTimeOffset` structure

The `DateTimeOffset` structure represents a date and time value, together with an offset that indicates how much that value differs from UTC. Thus, the value always

unambiguously identifies a single point in time.

The [DateTimeOffset](#) type includes all of the functionality of the [DateTime](#) type along with time zone awareness. This makes it suitable for applications that:

- Uniquely and unambiguously identify a single point in time. The [DateTimeOffset](#) type can be used to unambiguously define the meaning of "now", to log transaction times, to log the times of system or application events, and to record file creation and modification times.
- Perform general date and time arithmetic.
- Preserve multiple related times, as long as those times are stored as two separate values or as two members of a structure.

### ⓘ Note

These uses for [DateTimeOffset](#) values are much more common than those for [DateTime](#) values. As a result, consider [DateTimeOffset](#) as the default date and time type for application development.

A [DateTimeOffset](#) value isn't tied to a particular time zone, but can originate from a variety of time zones. The following example lists the time zones to which a number of [DateTimeOffset](#) values (including a local Pacific Standard Time) can belong.

C#

```
using System;
using System.Collections.ObjectModel;

public class TimeOffsets
{
    public static void Main()
    {
        DateTime thisDate = new DateTime(2007, 3, 10, 0, 0, 0);
        DateTime dstDate = new DateTime(2007, 6, 10, 0, 0, 0);
        DateTimeOffset thisTime;

        thisTime = new DateTimeOffset(dstDate, new TimeSpan(-7, 0, 0));
        ShowPossibleTimeZones(thisTime);

        thisTime = new DateTimeOffset(thisDate, new TimeSpan(-6, 0, 0));
        ShowPossibleTimeZones(thisTime);

        thisTime = new DateTimeOffset(thisDate, new TimeSpan(+1, 0, 0));
        ShowPossibleTimeZones(thisTime);
    }

    private static void ShowPossibleTimeZones(DateTimeOffset offsetTime)
```

```

{
    TimeSpan offset = offsetTime.Offset;
    ReadOnlyCollection<TimeZoneInfo> timeZones;

    Console.WriteLine("{0} could belong to the following time zones:",
        offsetTime.ToString());
    // Get all time zones defined on local system
    timeZones = TimeZoneInfo.GetSystemTimeZones();
    // Iterate time zones
    foreach (TimeZoneInfo timeZone in timeZones)
    {
        // Compare offset with offset for that date in that time zone
        if (timeZone.GetUtcOffset(offsetTime.DateTime).Equals(offset))
            Console.WriteLine("  {0}", timeZone.DisplayName);
    }
    Console.WriteLine();
}

// This example displays the following output to the console:
//       6/10/2007 12:00:00 AM -07:00 could belong to the following time
// zones:
//           (GMT-07:00) Arizona
//           (GMT-08:00) Pacific Time (US & Canada)
//           (GMT-08:00) Tijuana, Baja California
//
//       3/10/2007 12:00:00 AM -06:00 could belong to the following time
// zones:
//           (GMT-06:00) Central America
//           (GMT-06:00) Central Time (US & Canada)
//           (GMT-06:00) Guadalajara, Mexico City, Monterrey - New
//           (GMT-06:00) Guadalajara, Mexico City, Monterrey - Old
//           (GMT-06:00) Saskatchewan
//
//       3/10/2007 12:00:00 AM +01:00 could belong to the following time
// zones:
//           (GMT+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna
//           (GMT+01:00) Belgrade, Bratislava, Budapest, Ljubljana, Prague
//           (GMT+01:00) Brussels, Copenhagen, Madrid, Paris
//           (GMT+01:00) Sarajevo, Skopje, Warsaw, Zagreb
//           (GMT+01:00) West Central Africa

```

The output shows that each date and time value in this example can belong to at least three different time zones. The [DateTimeOffset](#) value of 6/10/2007 shows that if a date and time value represents a daylight saving time, its offset from UTC doesn't even necessarily correspond to the originating time zone's base UTC offset or to the offset from UTC found in its display name. Because a single [DateTimeOffset](#) value isn't tightly coupled with its time zone, it can't reflect a time zone's transition to and from daylight saving time. This can be problematic when date and time arithmetic is used to manipulate a [DateTimeOffset](#) value. For a discussion of how to perform date and time

arithmetic in a way that takes account of a time zone's adjustment rules, see [Performing arithmetic operations with dates and times](#).

## The DateTime structure

A [DateTime](#) value defines a particular date and time. It includes a [Kind](#) property that provides limited information about the time zone to which that date and time belongs. The [DateTimeKind](#) value returned by the [Kind](#) property indicates whether the [DateTime](#) value represents the local time ([DateTimeKind.Local](#)), Coordinated Universal Time (UTC) ([DateTimeKind.Utc](#)), or an unspecified time ([DateTimeKind.Unspecified](#)).

The [DateTime](#) structure is suitable for applications with one or more of the following characteristics:

- Work with abstract dates and times.
- Work with dates and times for which time zone information is missing.
- Work with UTC dates and times only.
- Perform date and time arithmetic, but are concerned with general results. For example, in an addition operation that adds six months to a particular date and time, it is often not important whether the result is adjusted for daylight saving time.

Unless a particular [DateTime](#) value represents UTC, that date and time value is often ambiguous or limited in its portability. For example, if a [DateTime](#) value represents the local time, it's portable within that local time zone (that is, if the value is serialized on another system in the same time zone, that value still unambiguously identifies a single point in time). Outside the local time zone, that [DateTime](#) value can have multiple interpretations. If the value's [Kind](#) property is [DateTimeKind.Unspecified](#), it's even less portable: it is now ambiguous within the same time zone and possibly even on the same system where it was first serialized. Only if a [DateTime](#) value represents UTC does that value unambiguously identify a single point in time regardless of the system or time zone in which the value is used.

 **Important**

When saving or sharing [DateTime](#) data, use UTC and set the [DateTime](#) value's [Kind](#) property to [DateTimeKind.Utc](#).

## The DateOnly structure

The `DateOnly` structure represents a specific date, without time. Since it has no time component, it represents a date from the start of the day to the end of the day. This structure is ideal for storing specific dates, such as a birth date, an anniversary date, a holiday, or a business-related date.

Although you could use `DateTime` while ignoring the time component, there are a few benefits to using `DateOnly` over `DateTime`:

- The `DateTime` structure may roll into the previous or next day if it's offset by a time zone. `DateOnly` can't be offset by a time zone, and it always represents the date that was set.
- Serializing a `DateTime` structure includes the time component, which may obscure the intent of the data. Also, `DateOnly` serializes less data.
- When code interacts with a database, such as SQL Server, whole dates are generally stored as the `date` data type, which doesn't include a time. `DateOnly` matches the database type better.

For more information about `DateOnly`, see [How to use the DateOnly and TimeOnly structures](#).

### ⓘ Important

`DateOnly` isn't available in .NET Framework.

## The `TimeSpan` structure

The `TimeSpan` structure represents a time interval. Its two typical uses are:

- Reflecting the time interval between two date and time values. For example, subtracting one `DateTime` value from another returns a `TimeSpan` value.
- Measuring elapsed time. For example, the `Stopwatch.Elapsed` property returns a `TimeSpan` value that reflects the time interval that has elapsed since the call to one of the `Stopwatch` methods that begins to measure elapsed time.

A `TimeSpan` value can also be used as a replacement for a `DateTime` value when that value reflects a time without reference to a particular day. This usage is similar to the `DateTime.TimeOfDay` and `DateTimeOffset.TimeOfDay` properties, which return a `TimeSpan` value that represents the time without reference to a date. For example, the `TimeSpan` structure can be used to reflect a store's daily opening or closing time, or it can be used to represent the time at which any regular event occurs.

The following example defines a `StoreInfo` structure that includes `TimeSpan` objects for store opening and closing times, as well as a `TimeZoneInfo` object that represents the store's time zone. The structure also includes two methods, `IsOpenNow` and `IsOpenAt`, that indicates whether the store is open at a time specified by the user, who is assumed to be in the local time zone.

C#

```
using System;

public struct StoreInfo
{
    public String store;
    public TimeZoneInfo tz;
    public TimeSpan open;
    public TimeSpan close;

    public bool IsOpenNow()
    {
        return IsOpenAt(DateTime.Now.TimeOfDay);
    }

    public bool IsOpenAt(TimeSpan time)
    {
        TimeZoneInfo local = TimeZoneInfo.Local;
        TimeSpan offset = TimeZoneInfo.Local.BaseUtcOffset;

        // Is the store in the same time zone?
        if (tz.Equals(local)) {
            return time >= open & time <= close;
        }
        else {
            TimeSpan delta = TimeSpan.Zero;
            TimeSpan storeDelta = TimeSpan.Zero;

            // Is it daylight saving time in either time zone?
            if (local.IsDaylightSavingTime(DateTime.Now.Date + time))
                delta = local.GetAdjustmentRules()
[local.GetAdjustmentRules().Length - 1].DaylightDelta;

            if
(tz.IsDaylightSavingTime(TimeZoneInfo.ConvertTime(DateTime.Now.Date + time,
local, tz)))
                storeDelta = tz.GetAdjustmentRules()
[tz.GetAdjustmentRules().Length - 1].DaylightDelta;

            TimeSpan comparisonTime = time + (offset -
tz.BaseUtcOffset).Negate() + (delta - storeDelta).Negate();
            return comparisonTime >= open & comparisonTime <= close;
        }
    }
}
```

```
    }  
}
```

The `StoreInfo` structure can then be used by client code like the following.

C#

```
public class Example  
{  
    public static void Main()  
    {  
        // Instantiate a StoreInfo object.  
        var store103 = new StoreInfo();  
        store103.store = "Store #103";  
        store103.tz = TimeZoneInfo.FindSystemTimeZoneById("Eastern Standard  
Time");  
        // Store opens at 8:00.  
        store103.open = new TimeSpan(8, 0, 0);  
        // Store closes at 9:30.  
        store103.close = new TimeSpan(21, 30, 0);  
  
        Console.WriteLine("Store is open now at {0}: {1}",  
                          DateTime.Now.TimeOfDay, store103.IsOpenNow());  
        TimeSpan[] times = { new TimeSpan(8, 0, 0), new TimeSpan(21, 0, 0),  
                            new TimeSpan(4, 59, 0), new TimeSpan(18, 31, 0)  
};  
        foreach (var time in times)  
        {  
            Console.WriteLine("Store is open at {0}: {1}",  
                             time, store103.IsOpenAt(time));  
        }  
    }  
    // The example displays the following output:  
    //      Store is open now at 15:29:01.6129911: True  
    //      Store is open at 08:00:00: True  
    //      Store is open at 21:00:00: False  
    //      Store is open at 04:59:00: False  
    //      Store is open at 18:31:00: False
```

## The `TimeOnly` structure

The `TimeOnly` structure represents a time-of-day value, such as a daily alarm clock or what time you eat lunch each day. `TimeOnly` is limited to the range of `00:00:00.0000000` - `23:59:59.9999999`, a specific time of day.

Prior to the `TimeOnly` type being introduced, programmers typically used either the `DateTime` type or the `TimeSpan` type to represent a specific time. However, using these structures to simulate a time without a date may introduce some problems, which `TimeOnly` solves:

- `TimeSpan` represents elapsed time, such as time measured with a stopwatch. The upper range is more than 29,000 years, and its value can be negative to indicate moving backwards in time. A negative `TimeSpan` doesn't indicate a specific time of the day.
- If `TimeSpan` is used as a time of day, there's a risk that it could be manipulated to a value outside of the 24-hour day. `TimeOnly` doesn't have this risk. For example, if an employee's work shift starts at 18:00 and lasts for 8 hours, adding 8 hours to the `TimeOnly` structure rolls over to 2:00.
- Using `DateTime` for a time of day requires that an arbitrary date be associated with the time, and then later disregarded. It's common practice to choose `DateTime.MinValue` (0001-01-01) as the date, however, if hours are subtracted from the `DateTime` value, an `OutOfRange` exception might occur. `TimeOnly` doesn't have this problem as the time rolls forwards and backwards around the 24-hour timeframe.
- Serializing a `DateTime` structure includes the date component, which may obscure the intent of the data. Also, `TimeOnly` serializes less data.

For more information about `TimeOnly`, see [How to use the DateOnly and TimeOnly structures](#).

 **Important**

`TimeOnly` isn't available in .NET Framework.

## The `TimeZoneInfo` class

The `TimeZoneInfo` class represents any of the Earth's time zones, and enables the conversion of any date and time in one time zone to its equivalent in another time zone. The `TimeZoneInfo` class makes it possible to work with dates and times so that any date and time value unambiguously identifies a single point in time. The `TimeZoneInfo` class is also extensible. Although it depends on time zone information provided for Windows systems and defined in the registry, it supports the creation of custom time zones. It also supports the serialization and deserialization of time zone information.

In some cases, taking full advantage of the `TimeZoneInfo` class may require further development work. If date and time values are not tightly coupled with the time zones to which they belong, further work is required. Unless your application provides some mechanism for linking a date and time with its associated time zone, it's easy for a particular date and time value to become disassociated from its time zone. One method

of linking this information is to define a class or structure that contains both the date and time value and its associated time zone object.

To take advantage of time zone support in .NET, you must know the time zone to which a date and time value belongs when that date and time object is instantiated. The time zone is often not known, particularly in web or network apps.

## See also

- [Dates, times, and time zones](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Work with calendars

Article • 08/27/2022

Although a date and time value represents a moment in time, its string representation is culture-sensitive and depends both on the conventions used for displaying date and time values by a specific culture and on the calendar used by that culture. This topic explores the support for calendars in .NET and discusses the use of the calendar classes when working with date values.

## Calendars in .NET

All calendars in .NET derive from the [System.Globalization.Calendar](#) class, which provides the base calendar implementation. One of the classes that inherits from the [Calendar](#) class is the [EastAsianLunisolarCalendar](#) class, which is the base class for all lunisolar calendars. .NET includes the following calendar implementations:

- [ChineseLunisolarCalendar](#), which represents the Chinese lunisolar calendar.
- [GregorianCalendar](#), which represents the Gregorian calendar. This calendar is further divided into subtypes (such as Arabic and Middle East French) that are defined by the [System.Globalization.GregorianCalendarTypes](#) enumeration. The [GregorianCalendar.CalendarType](#) property specifies the subtype of the Gregorian calendar.
- [HebrewCalendar](#), which represents the Hebrew calendar.
- [HijriCalendar](#), which represents the Hijri calendar.
- [JapaneseCalendar](#), which represents the Japanese calendar.
- [JapaneseLunisolarCalendar](#), which represents the Japanese lunisolar calendar.
- [JulianCalendar](#), which represents the Julian calendar.
- [KoreanCalendar](#), which represents the Korean calendar.
- [KoreanLunisolarCalendar](#), which represents the Korean lunisolar calendar.
- [PersianCalendar](#), which represents the Persian calendar.
- [TaiwanCalendar](#), which represents the Taiwan calendar.
- [TaiwanLunisolarCalendar](#), which represents the Taiwan lunisolar calendar.

- [ThaiBuddhistCalendar](#), which represents the Thai Buddhist calendar.
- [UmAlQuraCalendar](#), which represents the Um Al Qura calendar.

A calendar can be used in one of two ways:

- As the calendar used by a specific culture. Each [CultureInfo](#) object has a current calendar, which is the calendar that the object is currently using. The string representations of all date and time values automatically reflect the current culture and its current calendar. Typically, the current calendar is the culture's default calendar. [CultureInfo](#) objects also have optional calendars, which include additional calendars that the culture can use.
- As a standalone calendar independent of a specific culture. In this case, [Calendar](#) methods are used to express dates as values that reflect the calendar.

Note that six calendar classes – [ChineseLunisolarCalendar](#), [JapaneseLunisolarCalendar](#), [JulianCalendar](#), [KoreanLunisolarCalendar](#), [PersianCalendar](#), and [TaiwanLunisolarCalendar](#) – can be used only as standalone calendars. They are not used by any culture as either the default calendar or as an optional calendar.

## Calendars and cultures

Each culture has a default calendar, which is defined by the [CultureInfo.Calendar](#) property. The [CultureInfo.OptionalCalendars](#) property returns an array of [Calendar](#) objects that specifies all the calendars supported by a particular culture, including that culture's default calendar.

The following example illustrates the [CultureInfo.Calendar](#) and [CultureInfo.OptionalCalendars](#) properties. It creates [CultureInfo](#) objects for the Thai (Thailand) and Japanese (Japan) cultures and displays their default and optional calendars. Note that in both cases, the culture's default calendar is also included in the [CultureInfo.OptionalCalendars](#) collection.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        // Create a CultureInfo for Thai in Thailand.
        CultureInfo th = CultureInfo.CreateSpecificCulture("th-TH");
    }
}
```

```

        DisplayCalendars(th);

        // Create a CultureInfo for Japanese in Japan.
        CultureInfo ja = CultureInfo.CreateSpecificCulture("ja-JP");
        DisplayCalendars(ja);
    }

    static void DisplayCalendars(CultureInfo ci)
    {
        Console.WriteLine("Calendars for the {0} culture:", ci.Name);

        // Get the culture's default calendar.
        Calendar defaultCalendar = ci.Calendar;
        Console.Write("    Default Calendar: {0}",
        GetCalendarName(defaultCalendar));

        if (defaultCalendar is GregorianCalendar)
            Console.WriteLine(" ({0})",
                ((GregorianCalendar)
        defaultCalendar).CalendarType);
        else
            Console.WriteLine();

        // Get the culture's optional calendars.
        Console.WriteLine("    Optional Calendars:");
        foreach (var optionalCalendar in ci.OptionalCalendars) {
            Console.Write("{0,6}{1}", "", GetCalendarName(optionalCalendar));
            if (optionalCalendar is GregorianCalendar)
                Console.WriteLine(" ({0})",
                    ((GregorianCalendar)
        optionalCalendar).CalendarType);

            Console.WriteLine();
        }
        Console.WriteLine();
    }

    static string GetCalendarName(Calendar cal)
    {
        return cal.ToString().Replace("System.Globalization.", "");
    }
}

// The example displays the following output:
//     Calendars for the th-TH culture:
//         Default Calendar: ThaiBuddhistCalendar
//         Optional Calendars:
//             ThaiBuddhistCalendar
//             GregorianCalendar (Localized)
//
//     Calendars for the ja-JP culture:
//         Default Calendar: GregorianCalendar (Localized)
//         Optional Calendars:
//             GregorianCalendar (Localized)
//             JapaneseCalendar
//             GregorianCalendar (USEnglish)

```

The calendar currently in use by a particular [CultureInfo](#) object is defined by the culture's [DateTimeFormatInfo.Calendar](#) property. A culture's [DateTimeFormatInfo](#) object is returned by the [CultureInfo.DateTimeFormat](#) property. When a culture is created, its default value is the same as the value of the [CultureInfo.Calendar](#) property. However, you can change the culture's current calendar to any calendar contained in the array returned by the [CultureInfo.OptionalCalendars](#) property. If you try to set the current calendar to a calendar that is not included in the [CultureInfo.OptionalCalendars](#) property value, an [ArgumentException](#) is thrown.

The following example changes the calendar used by the Arabic (Saudi Arabia) culture. It first instantiates a [DateTime](#) value and displays it using the current culture - which, in this case, is English (United States) - and the current culture's calendar (which, in this case, is the Gregorian calendar). Next, it changes the current culture to Arabic (Saudi Arabia) and displays the date using its default Um Al-Qura calendar. It then calls the [CalendarExists](#) method to determine whether the Hijri calendar is supported by the Arabic (Saudi Arabia) culture. Because the calendar is supported, it changes the current calendar to Hijri and again displays the date. Note that in each case, the date is displayed using the current culture's current calendar.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
    public static void Main()
    {
        DateTime date1 = new DateTime(2011, 6, 20);

        DisplayCurrentInfo();
        // Display the date using the current culture and calendar.
        Console.WriteLine(date1.ToString("d"));
        Console.WriteLine();

        CultureInfo arSA = CultureInfo.CreateSpecificCulture("ar-SA");

        // Change the current culture to Arabic (Saudi Arabia).
        Thread.CurrentThread.CurrentCulture = arSA;
        // Display date and information about the current culture.
        DisplayCurrentInfo();
        Console.WriteLine(date1.ToString("d"));
        Console.WriteLine();

        // Change the calendar to Hijri.
        Calendar hijri = new HijriCalendar();
```

```

        if (CalendarExists(arSA, hijri)) {
            arSA.DateTimeFormat.Calendar = hijri;
            // Display date and information about the current culture.
            DisplayCurrentInfo();
            Console.WriteLine(date1.ToString("d"));
        }
    }

    private static void DisplayCurrentInfo()
    {
        Console.WriteLine("Current Culture: {0}",
                          CultureInfo.CurrentCulture.Name);
        Console.WriteLine("Current Calendar: {0}",
                          DateTimeFormatInfo.CurrentInfo.Calendar);
    }

    private static bool CalendarExists(CultureInfo culture, Calendar cal)
    {
        foreach (Calendar optionalCalendar in culture.OptionalCalendars)
            if (cal.ToString().Equals(optionalCalendar.ToString()))
                return true;

        return false;
    }
}

// The example displays the following output:
//   Current Culture: en-US
//   Current Calendar: System.Globalization.GregorianCalendar
//   6/20/2011
//
//   Current Culture: ar-SA
//   Current Calendar: System.Globalization.UmAlQuraCalendar
//   18/07/32
//
//   Current Culture: ar-SA
//   Current Calendar: System.Globalization.HijriCalendar
//   19/07/32

```

## Dates and calendars

With the exception of the constructors that include a parameter of type [Calendar](#) and allow the elements of a date (that is, the month, the day, and the year) to reflect values in a designated calendar, both [DateTime](#) and [DateTimeOffset](#) values are always based on the Gregorian calendar. This means, for example, that the [DateTime.Year](#) property returns the year in the Gregorian calendar, and the [DateTime.Day](#) property returns the day of the month in the Gregorian calendar.

 **Important**

It is important to remember that there is a difference between a date value and its string representation. The former is based on the Gregorian calendar; the latter is based on the current calendar of a specific culture.

The following example illustrates this difference between `DateTime` properties and their corresponding `Calendar` methods. In the example, the current culture is Arabic (Egypt), and the current calendar is Um Al Qura. A `DateTime` value is set to the fifteenth day of the seventh month of 2011. It is clear that this is interpreted as a Gregorian date, because these same values are returned by the `DateTime.ToString(String, IFormatProvider)` method when it uses the conventions of the invariant culture. The string representation of the date that is formatted using the conventions of the current culture is 14/08/32, which is the equivalent date in the Um Al Qura calendar. Next, members of `DateTime` and `Calendar` are used to return the day, the month, and the year of the `DateTime` value. In each case, the values returned by `DateTime` members reflect values in the Gregorian calendar, whereas values returned by `UmAlQuraCalendar` members reflect values in the Umm al-Qura calendar.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
    public static void Main()
    {
        // Make Arabic (Egypt) the current culture
        // and Umm al-Qura calendar the current calendar.
        CultureInfo arEG = CultureInfo.CreateSpecificCulture("ar-EG");
        Calendar cal = new UmAlQuraCalendar();
        arEG.DateTimeFormat.Calendar = cal;
        Thread.CurrentThread.CurrentCulture = arEG;

        // Display information on current culture and calendar.
        DisplayCurrentInfo();

        // Instantiate a date object.
        DateTime date1 = new DateTime(2011, 7, 15);

        // Display the string representation of the date.
        Console.WriteLine("Date: {0:d}", date1);
        Console.WriteLine("Date in the Invariant Culture: {0}",
            date1.ToString("d", CultureInfo.InvariantCulture));
        Console.WriteLine();

        // Compare DateTime properties and Calendar methods.
        Console.WriteLine("DateTime.Month property: {0}", date1.Month);
```

```

        Console.WriteLine("UmAlQura.GetMonth: {0}",
                           cal.GetMonth(date1));
        Console.WriteLine();

        Console.WriteLine("DateTime.Day property: {0}", date1.Day);
        Console.WriteLine("UmAlQura.GetDayOfMonth: {0}",
                           cal.GetDayOfMonth(date1));
        Console.WriteLine();

        Console.WriteLine("DateTime.Year property: {0:D4}", date1.Year);
        Console.WriteLine("UmAlQura.GetYear: {0}",
                           cal.GetYear(date1));
        Console.WriteLine();
    }

    private static void DisplayCurrentInfo()
    {
        Console.WriteLine("Current Culture: {0}",
                           CultureInfo.CurrentCulture.Name);
        Console.WriteLine("Current Calendar: {0}",
                           DateTimeFormatInfo.CurrentInfo.Calendar);
    }
}

// The example displays the following output:
//   Current Culture: ar-EG
//   Current Calendar: System.Globalization.UmAlQuraCalendar
//   Date: 14/08/32
//   Date in the Invariant Culture: 07/15/2011
//
//   DateTime.Month property: 7
//   UmAlQura.GetMonth: 8
//
//   DateTime.Day property: 15
//   UmAlQura.GetDayOfMonth: 14
//
//   DateTime.Year property: 2011
//   UmAlQura.GetYear: 1432

```

## Instantiate dates based on a calendar

Because [DateTime](#) and [DateTimeOffset](#) values are based on the Gregorian calendar, you must call an overloaded constructor that includes a parameter of type [Calendar](#) to instantiate a date value if you want to use the day, month, or year values from a different calendar. You can also call one of the overloads of a specific calendar's [Calendar.ToDateTime](#) method to instantiate a [DateTime](#) object based on the values of a particular calendar.

The following example instantiates one [DateTime](#) value by passing a [HebrewCalendar](#) object to a [DateTime](#) constructor, and instantiates a second [DateTime](#) value by calling the [HebrewCalendar.ToDateTime\(Int32, Int32, Int32, Int32, Int32, Int32, Int32\)](#)

method. Because the two values are created with identical values from the Hebrew calendar, the call to the `DateTime.Equals` method shows that the two `DateTime` values are equal.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        HebrewCalendar hc = new HebrewCalendar();

        DateTime date1 = new DateTime(5771, 6, 1, hc);
        DateTime date2 = hc.ToDateTime(5771, 6, 1, 0, 0, 0);

        Console.WriteLine("{0:d} (Gregorian) = {1:d2}/{2:d2}/{3:d4} ({4}):{5}",
            date1,
            hc.GetMonth(date2),
            hc.GetDayOfMonth(date2),
            hc.GetYear(date2),
            GetCalendarName(hc),
            date1.Equals(date2));
    }

    private static string GetCalendarName(Calendar cal)
    {
        return cal.ToString().Replace("System.Globalization.", "").Replace("Calendar", "");
    }
}

// The example displays the following output:
// 2/5/2011 (Gregorian) = 06/01/5771 (Hebrew): True
```

## Represent dates in the current calendar

Date and time formatting methods always use the current calendar when converting dates to strings. This means that the string representation of the year, the month, and the day of the month reflect the current calendar, and do not necessarily reflect the Gregorian calendar.

The following example shows how the current calendar affects the string representation of a date. It changes the current culture to Chinese (Traditional, Taiwan), and instantiates a date value. It then displays the current calendar and the date, changes the current calendar to `TaiwanCalendar`, and displays the current calendar and date once again. The

first time the date is displayed, it is represented as a date in the Gregorian calendar. The second time it is displayed, it is represented as a date in the Taiwan calendar.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
    public static void Main()
    {
        // Change the current culture to zh-TW.
        CultureInfo zhTW = CultureInfo.CreateSpecificCulture("zh-TW");
        Thread.CurrentThread.CurrentCulture = zhTW;
        // Define a date.
        DateTime date1 = new DateTime(2011, 1, 16);

        // Display the date using the default (Gregorian) calendar.
        Console.WriteLine("Current calendar: {0}",
                          zhTW.DateTimeFormat.Calendar);
        Console.WriteLine(date1.ToString("d"));

        // Change the current calendar and display the date.
        zhTW.DateTimeFormat.Calendar = new TaiwanCalendar();
        Console.WriteLine("Current calendar: {0}",
                          zhTW.DateTimeFormat.Calendar);
        Console.WriteLine(date1.ToString("d"));
    }
}

// The example displays the following output:
//   Current calendar: System.Globalization.GregorianCalendar
//   2011/1/16
//   Current calendar: System.Globalization.TaiwanCalendar
//   100/1/16
```

## Represent dates in a non-current calendar

To represent a date using a calendar that is not the current calendar of a particular culture, you must call methods of that [Calendar](#) object. For example, the [Calendar.GetYear](#), [Calendar.GetMonth](#), and [Calendar.GetDayOfMonth](#) methods convert the year, month, and day to values that reflect a particular calendar.

### Warning

Because some calendars are not optional calendars of any culture, representing dates in these calendars always requires that you call calendar methods. This is true

of all calendars that derive from the [EastAsianLunisolarCalendar](#), [JulianCalendar](#), and [PersianCalendar](#) classes.

The following example uses a [JulianCalendar](#) object to instantiate a date, January 9, 1905, in the Julian calendar. When this date is displayed using the default (Gregorian) calendar, it is represented as January 22, 1905. Calls to individual [JulianCalendar](#) methods enable the date to be represented in the Julian calendar.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        JulianCalendar julian = new JulianCalendar();
        DateTime date1 = new DateTime(1905, 1, 9, julian);

        Console.WriteLine("Date ({0}): {1:d}",
            CultureInfo.CurrentCulture.Calendar,
            date1);
        Console.WriteLine("Date in Julian calendar: {0:d2}/{1:d2}/{2:d4}",
            julian.GetMonth(date1),
            julian.GetDayOfMonth(date1),
            julian.GetYear(date1));
    }
}

// The example displays the following output:
//      Date (System.Globalization.GregorianCalendar): 1/22/1905
//      Date in Julian calendar: 01/09/1905
```

## Calendars and date ranges

The earliest date supported by a calendar is indicated by that calendar's [Calendar.MinSupportedDateTime](#) property. For the [GregorianCalendar](#) class, that date is January 1, 0001 C.E. Most of the other calendars in .NET support a later date. Trying to work with a date and time value that precedes a calendar's earliest supported date throws an [ArgumentOutOfRangeException](#) exception.

However, there is one important exception. The default (uninitialized) value of a [DateTime](#) object and a [DateTimeOffset](#) object is equal to the [GregorianCalendar.MinSupportedDateTime](#) value. If you try to format this date in a calendar that does not support January 1, 0001 C.E. and you do not provide a format specifier, the formatting method uses the "s" (sortable date/time pattern) format

specifier instead of the "G" (general date/time pattern) format specifier. As a result, the formatting operation does not throw an [ArgumentOutOfRangeException](#) exception. Instead, it returns the unsupported date. This is illustrated in the following example, which displays the value of [DateTime.MinValue](#) when the current culture is set to Japanese (Japan) with the Japanese calendar, and to Arabic (Egypt) with the Um Al Qura calendar. It also sets the current culture to English (United States) and calls the [DateTime.ToString\(IFormatProvider\)](#) method with each of these [CultureInfo](#) objects. In each case, the date is displayed by using the sortable date/time pattern.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
    public static void Main()
    {
        DateTime dat = DateTime.MinValue;

        // Change the current culture to ja-JP with the Japanese Calendar.
        CultureInfo jaJP = CultureInfo.CreateSpecificCulture("ja-JP");
        jaJP.DateTimeFormat.Calendar = new JapaneseCalendar();
        Thread.CurrentThread.CurrentCulture = jaJP;
        Console.WriteLine("Earliest supported date by {1} calendar: {0:d}",
                          jaJP.DateTimeFormat.Calendar.MinSupportedDateTime,
                          GetCalendarName(jaJP));
        // Attempt to display the date.
        Console.WriteLine(dat.ToString());
        Console.WriteLine();

        // Change the current culture to ar-EG with the Um Al Qura calendar.
        CultureInfo arEG = CultureInfo.CreateSpecificCulture("ar-EG");
        arEG.DateTimeFormat.Calendar = new UmAlQuraCalendar();
        Thread.CurrentThread.CurrentCulture = arEG;
        Console.WriteLine("Earliest supported date by {1} calendar: {0:d}",
                          arEG.DateTimeFormat.Calendar.MinSupportedDateTime,
                          GetCalendarName(arEG));
        // Attempt to display the date.
        Console.WriteLine(dat.ToString());
        Console.WriteLine();

        // Change the current culture to en-US.
        Thread.CurrentThread.CurrentCulture =
        CultureInfo.CreateSpecificCulture("en-US");
        Console.WriteLine(dat.ToString(jaJP));
        Console.WriteLine(dat.ToString(arEG));
        Console.WriteLine(dat.ToString("d"));
    }

    private static string GetCalendarName(CultureInfo culture)
```

```
    {
        Calendar cal = culture.DateTimeFormat.Calendar;
        return cal.GetType().Name.Replace("System.Globalization.",
        "").Replace("Calendar", "");
    }
}

// The example displays the following output:
//      Earliest supported date by Japanese calendar: 明治 1/9/8
//      0001-01-01T00:00:00
//
//      Earliest supported date by UmAlQura calendar: 01/01/18
//      0001-01-01T00:00:00
//
//      0001-01-01T00:00:00
//      0001-01-01T00:00:00
//      1/1/0001
```

## Work with eras

Calendars typically divide dates into eras. However, the [Calendar](#) classes in .NET do not support every era defined by a calendar, and most of the [Calendar](#) classes support only a single era. Only the [JapaneseCalendar](#) and [JapaneseLunisolarCalendar](#) classes support multiple eras.

### Important

The Reiwa era, a new era in the [JapaneseCalendar](#) and [JapaneseLunisolarCalendar](#), begins on May 1, 2019. This change affects all applications that use these calendars. See the following articles for more information:

- [Handling a new era in the Japanese calendar in .NET](#), which documents features added to .NET to support calendars with multiple eras and discusses best practices to use when handling multi-era calendars.
- [Prepare your application for the Japanese era change](#), which provides information on testing your applications on Windows to ensure their readiness for the era change.
- [Summary of new Japanese Era updates for .NET Framework](#), which lists .NET Framework updates for individual Windows versions that are related to the new Japanese calendar era, notes new .NET Framework features for multi-era support, and includes things to look for in testing your applications.

An era in most calendars denotes an extremely long time period. In the Gregorian calendar, for example, the current era spans more than two millennia. For the [JapaneseCalendar](#) and the [JapaneseLunisolarCalendar](#), the two calendars that support multiple eras, this is not the case. An era corresponds to the period of an emperor's reign. Support for multiple eras, particularly when the upper limit of the current era is unknown, poses special challenges.

## Eras and era names

In .NET, integers that represent the eras supported by a particular calendar implementation are stored in reverse order in the [Calendar.Eras](#) array. The current era (which is the era with the latest time range) is at index zero, and for [Calendar](#) classes that support multiple eras, each successive index reflects the previous era. The static [Calendar.CurrentEra](#) property defines the index of the current era in the [Calendar.Eras](#) array; it is a constant whose value is always zero. Individual [Calendar](#) classes also include static fields that return the value of the current era. They are listed in the following table.

Calendar class	Current era field
<a href="#">ChineseLunisolarCalendar</a>	<a href="#">ChineseEra</a>
<a href="#">GregorianCalendar</a>	<a href="#">ADEra</a>
<a href="#">HebrewCalendar</a>	<a href="#">HebrewEra</a>
<a href="#">HijriCalendar</a>	<a href="#">HijriEra</a>
<a href="#">JapaneseLunisolarCalendar</a>	<a href="#">JapaneseEra</a>
<a href="#">JulianCalendar</a>	<a href="#">JulianEra</a>
<a href="#">KoreanCalendar</a>	<a href="#">KoreanEra</a>
<a href="#">KoreanLunisolarCalendar</a>	<a href="#">GregorianEra</a>
<a href="#">PersianCalendar</a>	<a href="#">PersianEra</a>
<a href="#">ThaiBuddhistCalendar</a>	<a href="#">ThaiBuddhistEra</a>
<a href="#">UmAlQuraCalendar</a>	<a href="#">UmAlQuraEra</a>

The name that corresponds to a particular era number can be retrieved by passing the era number to the [DateTimeFormatInfo.GetEraName](#) or [DateTimeFormatInfo.GetAbbreviatedEraName](#) method. The following example calls these methods to retrieve information about era support in the [GregorianCalendar](#) class. It displays the Gregorian calendar date that corresponds to January 1 of the second year

of the current era, as well as the Gregorian calendar date that corresponds to January 1 of the second year of each supported Japanese calendar era.

```
C#
```

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        int year = 2;
        int month = 1;
        int day = 1;
        Calendar cal = new JapaneseCalendar();

        Console.WriteLine("\nDate instantiated without an era:");
        DateTime date1 = new DateTime(year, month, day, 0, 0, 0, 0, cal);
        Console.WriteLine("{0}/{1}/{2} in Japanese Calendar -> {3:d} in
Gregorian",
                           cal.GetMonth(date1), cal.GetDayOfMonth(date1),
                           cal.GetYear(date1), date1);

        Console.WriteLine("\nDates instantiated with eras:");
        foreach (int era in cal.Eras) {
            DateTime date2 = cal.ToDateTime(year, month, day, 0, 0, 0, 0, era);
            Console.WriteLine("{0}/{1}/{2} era {3} in Japanese Calendar ->
{4:d} in Gregorian",
                           cal.GetMonth(date2), cal.GetDayOfMonth(date2),
                           cal.GetYear(date2), cal.GetEra(date2), date2);
        }
    }
}
```

In addition, the "g" custom date and time format string includes a calendar's era name in the string representation of a date and time. For more information, see [Custom date and time format strings](#).

## Instantiate a date with an era

For the two [Calendar](#) classes that support multiple eras, a date that consists of a particular year, month, and day of the month value can be ambiguous. For example, all eras supported by the [JapaneseCalendar](#) have years whose number is 1. Ordinarily, if an era is not specified, both date and time and calendar methods assume that values belong to the current era. This is true of the [DateTime](#) and [DateTimeOffset](#) constructors that include parameters of type [Calendar](#), as well as the [JapaneseCalendar.ToDateTime](#) and [JapaneseLunisolarCalendar.ToDateTime](#) methods. The following example

instantiates a date that represents January 1 of the second year of an unspecified era. If you execute the example when the Reiwa era is the current era, the date is interpreted as the second year of the Reiwa era. The era, 令和, precedes the year in the string returned by the [DateTime.ToString\(String, IFormatProvider\)](#) method and corresponds to January 1, 2020, in the Gregorian calendar. (The Reiwa era begins in the year 2019 of the Gregorian calendar.)

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        var japaneseCal = new JapaneseCalendar();
        var jaJp = new CultureInfo("ja-JP");
        jaJp.DateTimeFormat.Calendar = japaneseCal;

        var date = new DateTime(2, 1, 1, japaneseCal);
        Console.WriteLine($"Gregorian calendar date: {date:d}");
        Console.WriteLine($"Japanese calendar date: {date.ToString("d",
jaJp)}");
    }
}
```

However, if the era changes, the intent of this code becomes ambiguous. Is the date intended to represent the second year of the current era, or is it intended to represent the second year of the Heisei era? There are two ways to avoid this ambiguity:

- Instantiate the date and time value using the default [GregorianCalendar](#) class. You can then use the Japanese calendar or the Japanese Lunisolar calendar for the string representation of dates, as the following example shows.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        var japaneseCal = new JapaneseCalendar();
        var jaJp = new CultureInfo("ja-JP");
        jaJp.DateTimeFormat.Calendar = japaneseCal;

        var date = new DateTime(1905, 2, 12);
    }
}
```

```

Console.WriteLine($"Gregorian calendar date: {date:d}");

        // Call the ToString(IFormatProvider) method.
        Console.WriteLine($"Japanese calendar date: {date.ToString("d",
jaJp)}");

        // Use a FormattableString object.
        FormattableString fmt = $"{date:d}";
        Console.WriteLine($"Japanese calendar date:
{fmt.ToString(jaJp)}");

        // Use the JapaneseCalendar object.
        Console.WriteLine($"Japanese calendar date:
{jaJp.DateTimeFormat.GetEraName(japaneseCal.GetEra(date))}" +
$" {japaneseCal.GetYear(date)}/{japaneseCal.GetMonth(date)}/{japaneseCal.G
etDayOfMonth(date)}");

        // Use the current culture.
        CultureInfo.CurrentCulture = jaJp;
        Console.WriteLine($"Japanese calendar date: {date:d}");
    }
}

// The example displays the following output:
//  Gregorian calendar date: 2/12/1905
//  Japanese calendar date: 明治38/2/12

```

- Call a date and time method that explicitly specifies an era. This includes the following methods:
  - The [ToDateTime\(Int32, Int32, Int32, Int32, Int32, Int32, Int32, Int32\)](#) method of the [JapaneseCalendar](#) or [JapaneseLunisolarCalendar](#) class.
  - A [DateTime](#) or [DateTimeOffset](#) parsing method, such as [Parse](#), [TryParse](#), [ParseExact](#), or [TryParseExact](#), that includes the string to be parsed and optionally a [DateTimeStyles](#) argument if the current culture is Japanese-Japan ("ja-JP") and that culture's calendar is the [JapaneseCalendar](#). The string to be parsed must include the era.
  - A [DateTime](#) or [DateTimeOffset](#) parsing method that includes a `provider` parameter of type [IFormatProvider](#). `provider` must be either a [CultureInfo](#) object that represents the Japanese-Japan ("ja-JP") culture whose current calendar is [JapaneseCalendar](#) or a [DateTimeFormatInfo](#) object whose [Calendar](#) property is [JapaneseCalendar](#). The string to be parsed must include the era.

The following example uses three of these methods to instantiate a date and time in the Meiji era, which began on September 8, 1868, and ended on July 29, 1912.

```
C#  
  
using System;  
using System.Globalization;  
  
public class Example  
{  
    public static void Main()  
    {  
        var japaneseCal = new JapaneseCalendar();  
        var jaJp = new CultureInfo("ja-JP");  
        jaJp.DateTimeFormat.Calendar = japaneseCal;  
  
        // We can get the era index by calling  
        DateTimeFormatInfo.GetEraName.  
        int eraIndex = 0;  
  
        for (int ctr = 0; ctr <  
jaJp.DateTimeFormat.Calendar.Eras.Length; ctr++)  
        {  
            if (jaJp.DateTimeFormat.GetEraName(ctr) == "明治")  
                eraIndex = ctr;  
            var date1 = japaneseCal.ToDateTime(23, 9, 8, 0, 0, 0, 0,  
eraIndex);  
            Console.WriteLine($"{date1.ToString("d", jaJp)} (Gregorian  
{date1:d})");  
  
            try {  
                var date2 = DateTime.Parse("明治23/9/8", jaJp);  
                Console.WriteLine($"{date2.ToString("d", jaJp)} (Gregorian  
{date2:d})");  
            }  
            catch (FormatException)  
            {  
                Console.WriteLine("The parsing operation failed.");  
            }  
  
            try {  
                var date3 = DateTime.ParseExact("明治23/9/8", "gyy/M/d",  
jaJp);  
                Console.WriteLine($"{date3.ToString("d", jaJp)} (Gregorian  
{date3:d})");  
            }  
            catch (FormatException)  
            {  
                Console.WriteLine("The parsing operation failed.");  
            }  
        }  
        // The example displays the following output:  
        // 明治23/9/8 (Gregorian 9/8/1890)  
    }  
}
```

```
// 明治23/9/8 (Gregorian 9/8/1890)
// 明治23/9/8 (Gregorian 9/8/1890)
```

### Tip

When working with calendars that support multiple eras, *always* use the Gregorian date to instantiate a date, or specify the era when you instantiate a date and time based on that calendar.

In specifying an era to the [ToDateTime\(Int32, Int32, Int32, Int32, Int32, Int32, Int32, Int32\)](#) method, you provide the index of the era in the calendar's [Eras](#) property. For calendars whose eras are subject to change, however, these indexes are not constant values; the current era is at index 0, and the oldest era is at index `Eras.Length - 1`. When a new era is added to a calendar, the indexes of the previous eras increase by one. You can supply the appropriate era index as follows:

- For dates in the current era, always use the calendar's [CurrentEra](#) property.
- For dates in a specified era, use the [DateTimeFormatInfo.GetEraName](#) method to retrieve the index that corresponds to a specified era name. This requires that the [JapaneseCalendar](#) be the current calendar of the [CultureInfo](#) object that represents the ja-JP culture. (This technique works for the [JapaneseLunisolarCalendar](#) as well, since it supports the same eras as the [JapaneseCalendar](#).) The previous example illustrates this approach.

## Calendars, eras, and date ranges: Relaxed range checks

Very much like individual calendars have supported date ranges, eras in the [JapaneseCalendar](#) and [JapaneseLunisolarCalendar](#) classes also have supported ranges. Previously, .NET used strict era range checks to ensure that an era-specific date was within the range of that era. That is, if a date is outside of the range of the specified era, the method throws an [ArgumentOutOfRangeException](#). Currently, .NET uses relaxed ranged checking by default. Updates to all versions of .NET introduced relaxed era range checks; the attempt to instantiate an era-specific date that is outside the range of the specified era "overflows" into the following era, and no exception is thrown.

The following example attempts to instantiate a date in the 65th year of the Showa era, which began on December 25, 1926 and ended on January 7, 1989. This date corresponds to January 9, 1990, which is outside the range of the Showa era in the [JapaneseCalendar](#). As the output from the example illustrates, the date displayed by the example is January 9, 1990, in the second year of the Heisei era.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        var jaJp = new CultureInfo("ja-JP");
        var cal = new JapaneseCalendar();
        jaJp.DateTimeFormat.Calendar = cal;
        string showaEra = "昭和";

        var dt = cal.ToDateTime(65, 1, 9, 15, 0, 0, 0, GetEraIndex(showaEra));
        FormattableString fmt = $"{dt:d}";

        Console.WriteLine($"Japanese calendar date: {fmt.ToString(jaJp)}");
        Console.WriteLine($"Gregorian calendar date: {fmt}");

        int GetEraIndex(string eraName)
        {
            foreach (var ctr in cal.Eras)
                if (jaJp.DateTimeFormat.GetEraName(ctr) == eraName)
                    return ctr;

            return 0;
        }
    }
}
// The example displays the following output:
//   Japanese calendar date: 平成2/1/9
//   Gregorian calendar date: 1/9/1990
```

If relaxed range checks are undesirable, you can restore strict range checks in a number of ways, depending on the version of .NET on which your application is running:

- **.NET Core:** Add the following to the `.netcore.runtime.json` config file:

JSON

```
"runtimeOptions": {
    "configProperties": {
        "Switch.System.Globalization.EnforceJapaneseEraYearRanges": true
    }
}
```

- **.NET Framework 4.6 or later:** Set the following `AppContext` switch in the `app.config` file:

## XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <AppContextSwitchOverrides
      value="Switch.System.Globalization.EnforceJapaneseEraYearRanges=true"
    />
  </runtime>
</configuration>
```

- **.NET Framework 4.5.2 or earlier:** Set the following registry value:

Value	
<b>Key</b>	HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework\AppContext
<b>Entry</b>	Switch.System.Globalization.EnforceJapaneseEraYearRanges
<b>Type</b>	REG_SZ
<b>Value</b>	true

With strict range checks enabled, the previous example throws an [ArgumentOutOfRangeException](#) and displays the following output:

### Console

```
Unhandled Exception: System.ArgumentOutOfRangeException: Valid values are
between 1 and 64, inclusive.
Parameter name: year
  at System.Globalization.GregorianCalendarHelper.GetYearOffset(Int32 year,
Int32 era, Boolean throwOnError)
  at System.Globalization.GregorianCalendarHelper.ToDateTime(Int32 year,
Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32
millisecond, Int32 era)
  at Example.Main()
```

## Represent dates in calendars with multiple eras

If a [Calendar](#) object supports eras and is the current calendar of a [CultureInfo](#) object, the era is included in the string representation of a date and time value for the full date and time, long date, and short date patterns. The following example displays these date patterns when the current culture is Japan (Japanese) and the current calendar is the Japanese calendar.

### C#

```

using System;
using System.Globalization;
using System.IO;
using System.Threading;

public class Example
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\eras.txt");
        DateTime dt = new DateTime(2012, 5, 1);

        CultureInfo culture = CultureInfo.CreateSpecificCulture("ja-JP");
        DateTimeFormatInfo dtfi = culture.DateTimeFormat;
        dtfi.Calendar = new JapaneseCalendar();
        Thread.CurrentThread.CurrentCulture = culture;

        sw.WriteLine("\n{0,-43} {1}", "Full Date and Time Pattern:",
        dtfi.FullDateTimePattern);
        sw.WriteLine(dt.ToString("F"));
        sw.WriteLine();

        sw.WriteLine("\n{0,-43} {1}", "Long Date Pattern:",
        dtfi.LongDatePattern);
        sw.WriteLine(dt.ToString("D"));

        sw.WriteLine("\n{0,-43} {1}", "Short Date Pattern:",
        dtfi.ShortDatePattern);
        sw.WriteLine(dt.ToString("d"));
        sw.Close();
    }
}

// The example writes the following output to a file:
//   Full Date and Time Pattern:          gg y'年'M'月'd'日' H:mm:ss
//   平成 24年5月1日 0:00:00
//
//   Long Date Pattern:                  gg y'年'M'月'd'日'
//   平成 24年5月1日
//
//   Short Date Pattern:                gg y/M/d
//   平成 24/5/1

```

## ⚠ Warning

The **JapaneseCalendar** class is the only calendar class in .NET that both supports dates in more than one era and that can be the current calendar of a **CultureInfo** object - specifically, of a **CultureInfo** object that represents the Japanese (Japan) culture.

For all calendars, the "g" custom format specifier includes the era in the result string. The following example uses the "MM-dd-yyyy g" custom format string to include the era in the result string when the current calendar is the Gregorian calendar.

C#

```
DateTime dat = new DateTime(2012, 5, 1);
Console.WriteLine("{0:MM-dd-yyyy g}", dat);
// The example displays the following output:
//      05-01-2012 A.D.
```

In cases where the string representation of a date is expressed in a calendar that is not the current calendar, the [Calendar](#) class includes a [Calendar.GetEra](#) method that can be used along with the [Calendar.GetYear](#), [Calendar.GetMonth](#), and [Calendar.GetDayOfMonth](#) methods to unambiguously indicate a date as well as the era to which it belongs. The following example uses the [JapaneseLunisolarCalendar](#) class to provide an illustration. However, note that including a meaningful name or abbreviation instead of an integer for the era in the result string requires that you instantiate a [DateTimeFormatInfo](#) object and make [JapaneseCalendar](#) its current calendar. (The [JapaneseLunisolarCalendar](#) calendar cannot be the current calendar of any culture, but in this case the two calendars share the same eras.)

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        DateTime date1 = new DateTime(2011, 8, 28);
        Calendar cal = new JapaneseLunisolarCalendar();

        Console.WriteLine("{0} {1:d4}/{2:d2}/{3:d2}",
                          cal.GetEra(date1),
                          cal.GetYear(date1),
                          cal.GetMonth(date1),
                          cal.GetDayOfMonth(date1));

        // Display eras
        CultureInfo culture = CultureInfo.CreateSpecificCulture("ja-JP");
        DateTimeFormatInfo dtfi = culture.DateTimeFormat;
        dtfi.Calendar = new JapaneseCalendar();

        Console.WriteLine("{0} {1:d4}/{2:d2}/{3:d2}",
                          dtfi.GetAbbreviatedEraName(cal.GetEra(date1)),
                          cal.GetYear(date1),
                          cal.GetMonth(date1),
```

```

        cal.GetDayOfMonth(date1));
    }
}

// The example displays the following output:
//      4 0023/07/29
//      平 0023/07/29

```

In the Japanese calendars, the first year of an era is called Gannen (元年). For example, instead of Heisei 1, the first year of the Heisei era can be described as Heisei Gannen. .NET adopts this convention in formatting operations for dates and times formatted with the following standard or custom date and time format strings when they are used with a [CultureInfo](#) object that represents the Japanese-Japan ("ja-JP") culture with the [JapaneseCalendar](#) class:

- [The long date pattern](#), indicated by the "D" standard date and time format string.
- [The full date long time pattern](#), indicated by the "F" standard date and time format string.
- [The full date short time pattern](#), indicated by the "f" standard date and time format string.
- [The year/month pattern](#), indicated by the "Y" or "y" standard date and time format string.
- The "ggy'年'" or "ggy年" [custom date and time format string](#).

For example, the following example displays a date in the first year of the Heisei era in the [JapaneseCalendar](#).

C#

```

using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        var enUs = new CultureInfo("en-US");
        var japaneseCal = new JapaneseCalendar();
        var jaJp = new CultureInfo("ja-JP");
        jaJp.DateTimeFormat.Calendar = japaneseCal;
        string heiseiEra = "平成";

        var date = japaneseCal.ToDateTime(1, 8, 18, 0, 0, 0, 0,
GetEraIndex(heiseiEra));
        FormattableString fmt = $"{date:D}";
        Console.WriteLine($"Japanese calendar date: {fmt.ToString(jaJp)}
(Gregorian: {fmt.ToString(enUs)})");

        int GetEraIndex(string eraName)
    }
}

```

```

    {
        foreach (var ctr in japaneseCal.Eras)
            if (jaJp.DateTimeFormat.GetEraName(ctr) == eraName)
                return ctr;

        return 0;
    }
}

// The example displays the following output:
//   Japanese calendar date: 平成元年8月18日 (Gregorian: Friday, August 18,
1989)

```

If this behavior is undesirable in formatting operations, you can restore the previous behavior, which always represents the first year of an era as "1" rather than "Gannen", by doing the following, depending on the version of .NET:

- **.NET Core:** Add the following to the `.netcore.runtime.json` config file:

JSON

```

"runtimeOptions": {
    "configProperties": {
        "Switch.System.Globalization.FormatJapaneseFirstYearAsANumber": true
    }
}

```

- **.NET Framework 4.6 or later:** Set the following `AppContext` switch in the `app.config` file:

XML

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <runtime>
        <AppContextSwitchOverrides
value="Switch.System.Globalization.FormatJapaneseFirstYearAsANumber=true" />
    </runtime>
</configuration>

```

- **.NET Framework 4.5.2 or earlier:** Set the following registry value:

Value
Key
HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework\AppContext

Value	
<b>Entry</b>	Switch.System.Globalization.FormatJapaneseFirstYearAsANumber
<b>Type</b>	REG_SZ
<b>Value</b>	true

With gannen support in formatting operations disabled, the previous example displays the following output:

Console

```
Japanese calendar date: 平成1年8月18日 (Gregorian: Friday, August 18, 1989)
```

.NET has also been updated so that date and time parsing operations support strings that contain the year represented as either "1" or Gannen. Although you should not need to do this, you can restore the previous behavior to recognize only "1" as the first year of an era. You can do this as follows, depending on the version of .NET:

- **.NET Core:** Add the following to the `.netcore.runtime.json` config file:

JSON

```
"runtimeOptions": {
  "configProperties": {
    "Switch.System.Globalization.EnforceLegacyJapaneseDateParsing": true
  }
}
```

- **.NET Framework 4.6 or later:** Set the following `AppContext` switch in the `app.config` file:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <AppContextSwitchOverrides
      value="Switch.System.Globalization.EnforceLegacyJapaneseDateParsing=true" />
  </runtime>
</configuration>
```

- **.NET Framework 4.5.2 or earlier:** Set the following registry value:

Value	
Key	HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework\ ApplicationContext
Entry	Switch.System.Globalization.EnforceLegacyJapaneseDateParsing
Type	REG_SZ
Value	true

## See also

- [How to: Display dates in non-Gregorian calendars](#)
- [Calendar class](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to use the DateOnly and TimeOnly structures

Article • 01/12/2023

The [DateOnly](#) and [TimeOnly](#) structures were introduced with .NET 6 and represent a specific date or time-of-day, respectively. Prior to .NET 6, and always in .NET Framework, developers used the [DateTime](#) type (or some other alternative) to represent one of the following:

- A whole date and time.
- A date, disregarding the time.
- A time, disregarding the date.

`DateOnly` and `TimeOnly` are types that represent those particular portions of a `DateTime` type.

## ⓘ Important

[DateOnly](#) and [TimeOnly](#) types aren't available in .NET Framework.

## The DateOnly structure

The [DateOnly](#) structure represents a specific date, without time. Since it has no time component, it represents a date from the start of the day to the end of the day. This structure is ideal for storing specific dates, such as a birth date, an anniversary date, or business-related dates.

Although you could use `DateTime` while ignoring the time component, there are a few benefits to using `DateOnly` over `DateTime`:

- The `DateTime` structure may roll into the previous or next day if it's offset by a time zone. `DateOnly` can't be offset by a time zone, and it always represents the date that was set.
- Serializing a `DateTime` structure includes the time component, which may obscure the intent of the data. Also, `DateOnly` serializes less data.
- When code interacts with a database, such as SQL Server, whole dates are generally stored as the `date` data type, which doesn't include a time. `DateOnly`

matches the database type better.

`DateOnly` has a range from 0001-01-01 through 9999-12-31, just like `DateTime`. You can specify a specific calendar in the `DateOnly` constructor. However, a `DateOnly` object always represents a date in the proleptic Gregorian calendar, regardless of which calendar was used to construct it. For example, you can build the date from a Hebrew calendar, but the date is converted to Gregorian:

C#

```
var hebrewCalendar = new System.Globalization.HebrewCalendar();
var theDate = new DateOnly(5776, 2, 8, hebrewCalendar); // 8 Cheshvan 5776

Console.WriteLine(theDate);

/* This example produces the following output:
 *
 * 10/21/2015
 */
```

## DateOnly examples

Use the following examples to learn about `DateOnly`:

- [Convert `DateTime` to `DateOnly`](#)
- [Add or subtract days, months, years](#)
- [Parse and format `DateOnly`](#)
- [Compare `DateOnly`](#)

## Convert `DateTime` to `DateOnly`

Use the `DateOnly.FromDateTime` static method to create a `DateOnly` type from a `DateTime` type, as demonstrated in the following code:

C#

```
var today = DateOnly.FromDateTime(DateTime.Now);
Console.WriteLine($"Today is {today}");

/* This example produces output similar to the following:
 *
 * Today is 12/28/2022
 */
```

## Add or subtract days, months, years

There are three methods used to adjust a [DateOnly](#) structure: [AddDays](#), [AddMonths](#), and [AddYears](#). Each method takes an integer parameter, and increases the date by that measurement. If a negative number is provided, the date is decreased by that measurement. The methods return a new instance of [DateOnly](#), as the structure is immutable.

C#

```
var theDate = new DateOnly(2015, 10, 21);

var nextDay = theDate.AddDays(1);
var previousDay = theDate.AddDays(-1);
var decadeLater = theDate.AddYears(10);
var lastMonth = theDate.AddMonths(-1);

Console.WriteLine($"Date: {theDate}");
Console.WriteLine($" Next day: {nextDay}");
Console.WriteLine($" Previous day: {previousDay}");
Console.WriteLine($" Decade later: {decadeLater}");
Console.WriteLine($" Last month: {lastMonth}");

/* This example produces the following output:
 *
 * Date: 10/21/2015
 * Next day: 10/22/2015
 * Previous day: 10/20/2015
 * Decade later: 10/21/2025
 * Last month: 9/21/2015
 */
```

## Parse and format DateOnly

[DateOnly](#) can be parsed from a string, just like the [DateTime](#) structure. All of the standard .NET date-based parsing tokens work with [DateOnly](#). When converting a [DateOnly](#) type to a string, you can use standard .NET date-based formatting patterns too. For more information about formatting strings, see [Standard date and time format strings](#).

C#

```
var theDate = DateOnly.ParseExact("21 Oct 2015", "dd MMM yyyy",
CultureInfo.InvariantCulture); // Custom format
var theDate2 = DateOnly.Parse("October 21, 2015",
CultureInfo.InvariantCulture);
```

```
Console.WriteLine(theDate.ToString("m", CultureInfo.InvariantCulture));
// Month day pattern
Console.WriteLine(theDate2.ToString("o", CultureInfo.InvariantCulture));
// ISO 8601 format
Console.WriteLine(theDate2.ToString("o", CultureInfo.InvariantCulture));
/* This example produces the following output:
 *
 * October 21
 * 2015-10-21
 * Wednesday, October 21, 2015
 */

```

## Compare DateOnly

`DateOnly` can be compared with other instances. For example, you can check if a date is before or after another, or if a date today matches a specific date.

C#

```
var theDate = DateOnly.ParseExact("21 Oct 2015", "dd MMM yyyy",
CultureInfo.InvariantCulture); // Custom format
var theDate2 = DateOnly.Parse("October 21, 2015",
CultureInfo.InvariantCulture);
var dateLater = theDate.AddMonths(6);
var dateBefore = theDate.AddDays(-10);

Console.WriteLine($"Consider {theDate}...");
Console.WriteLine($" Is '{nameof(theDate2)}' equal? {theDate == theDate2}");
Console.WriteLine($" Is {dateLater} after? {dateLater > theDate} ");
Console.WriteLine($" Is {dateLater} before? {dateLater < theDate} ");
Console.WriteLine($" Is {dateBefore} after? {dateBefore > theDate} ");
Console.WriteLine($" Is {dateBefore} before? {dateBefore < theDate} ");

/* This example produces the following output:
*
* Consider 10/21/2015
* Is 'theDate2' equal? True
* Is 4/21/2016 after? True
* Is 4/21/2016 before? False
* Is 10/11/2015 after? False
* Is 10/11/2015 before? True
*/

```

## The TimeOnly structure

The `TimeOnly` structure represents a time-of-day value, such as a daily alarm clock or what time you eat lunch each day. `TimeOnly` is limited to the range of 00:00:00.0000000

- 23:59:59.9999999, a specific time of day.

Prior to the `TimeOnly` type being introduced, programmers typically used either the `DateTime` type or the `TimeSpan` type to represent a specific time. However, using these structures to simulate a time without a date may introduce some problems, which `TimeOnly` solves:

- `TimeSpan` represents elapsed time, such as time measured with a stopwatch. The upper range is more than 29,000 years, and its value can be negative to indicate moving backwards in time. A negative `TimeSpan` doesn't indicate a specific time of the day.
- If `TimeSpan` is used as a time of day, there's a risk that it could be manipulated to a value outside of the 24-hour day. `TimeOnly` doesn't have this risk. For example, if an employee's work shift starts at 18:00 and lasts for 8 hours, adding 8 hours to the `TimeOnly` structure rolls over to 2:00
- Using `DateTime` for a time of day requires that an arbitrary date be associated with the time, and then later disregarded. It's common practice to choose `DateTime.MinValue` (0001-01-01) as the date, however, if hours are subtracted from the `DateTime` value, an `OutOfRangeException` exception might occur. `TimeOnly` doesn't have this problem as the time rolls forwards and backwards around the 24-hour timeframe.
- Serializing a `DateTime` structure includes the date component, which may obscure the intent of the data. Also, `TimeOnly` serializes less data.

## TimeOnly examples

Use the following examples to learn about `TimeOnly`:

- [Convert DateTime to TimeOnly](#)
- [Add or subtract time](#)
- [Parse and format TimeOnly](#)
- [Work with TimeSpan and DateTime](#)
- [Arithmetic operators and comparing TimeOnly](#)

## Convert DateTime to TimeOnly

Use the `TimeOnly.FromDateTime` static method to create a `TimeOnly` type from a `DateTime` type, as demonstrated in the following code:

C#

```
var now = TimeOnly.FromDateTime(DateTime.Now);
Console.WriteLine($"It is {now} right now");

/* This example produces output similar to the following:
 *
 * It is 2:01 PM right now
 */
```

## Add or subtract time

There are three methods used to adjust a `TimeOnly` structure: `AddHours`, `AddMinutes`, and `Add`. Both `AddHours` and `AddMinutes` take an integer parameter, and adjust the value accordingly. You can use a negative value to subtract and a positive value to add. The methods return a new instance of `TimeOnly` is returned, as the structure is immutable. The `Add` method takes a `TimeSpan` parameter and adds or subtracts the value from the `TimeOnly` value.

Because `TimeOnly` only represents a 24-hour period, it rolls over forwards or backwards appropriately when adding values supplied to those three methods. For example, if you use a value of `01:30:00` to represent 1:30 AM, then add -4 hours from that period, it rolls backwards to `21:30:00`, which is 9:30 PM. There are method overloads for `AddHours`, `AddMinutes`, and `Add` that capture the number of days rolled over.

C#

```
var theTime = new TimeOnly(7, 23, 11);

var hourLater = theTime.AddHours(1);
var minutesBefore = theTime.AddMinutes(-12);
var secondsAfter = theTime.Add(TimeSpan.FromSeconds(10));
var daysLater = theTime.Add(new TimeSpan(hours: 21, minutes: 200, seconds: 83), out int wrappedDays);
var daysBehind = theTime.AddHours(-222, out int wrappedDaysFromHours);

Console.WriteLine($"Time: {theTime}");
Console.WriteLine($" Hours later: {hourLater}");
Console.WriteLine($" Minutes before: {minutesBefore}");
Console.WriteLine($" Seconds after: {secondsAfter}");
Console.WriteLine($" {daysLater} is the time, which is {wrappedDays} days later");
Console.WriteLine($" {daysBehind} is the time, which is {wrappedDaysFromHours} days prior");

/* This example produces the following output:
 *
```

```
* Time: 7:23 AM
* Hours later: 8:23 AM
* Minutes before: 7:11 AM
* Seconds after: 7:23 AM
* 7:44 AM is the time, which is 1 days later
* 1:23 AM is the time, which is -9 days prior
*/
```

## Parse and format `TimeOnly`

`TimeOnly` can be parsed from a string, just like the `DateTime` structure. All of the standard .NET time-based parsing tokens work with `TimeOnly`. When converting a `TimeOnly` type to a string, you can use standard .NET date-based formatting patterns too. For more information about formatting strings, see [Standard date and time format strings](#).

C#

```
var theTime = TimeOnly.ParseExact("5:00 pm", "h:mm tt",
CultureInfo.InvariantCulture); // Custom format
var theTime2 = TimeOnly.Parse("17:30:25", CultureInfo.InvariantCulture);

Console.WriteLine(theTime.ToString("o", CultureInfo.InvariantCulture));
// Round-trip pattern.
Console.WriteLine(theTime2.ToString("t", CultureInfo.InvariantCulture));
// Long time format
Console.WriteLine(theTime2.ToString("o", CultureInfo.InvariantCulture));

/* This example produces the following output:
*
* 17:00:00.0000000
* 17:30
* 5:30:25 PM
*/
```

## Serialize `DateOnly` and `TimeOnly` types

With .NET 7+, `System.Text.Json` supports serializing and deserializing `DateOnly` and `TimeOnly` types. Consider the following object:

C#

```
sealed file record Appointment(
    Guid Id,
    string Description,
    DateOnly Date,
```

```
TimeOnly StartTime,  
TimeOnly EndTime);
```

The following example serializes an `Appointment` object, displays the resulting JSON, and then deserializes it back into a new instance of the `Appointment` type. Finally, the original and newly deserialized instances are compared for equality and the results are written to the console:

C#

```
Appointment originalAppointment = new(  
    Id: Guid.NewGuid(),  
    Description: "Take dog to veterinarian.",  
    Date: new DateOnly(2002, 1, 13),  
    StartTime: new TimeOnly(5,15),  
    EndTime: new TimeOnly(5, 45));  
string serialized = JsonSerializer.Serialize(originalAppointment);  
  
Console.WriteLine($"Resulting JSON: {serialized}");  
  
Appointment deserializedAppointment =  
    JsonSerializer.Deserialize<Appointment>(serialized)!;  
  
bool valuesAreTheSame = originalAppointment == deserializedAppointment;  
Console.WriteLine($"""  
    Original record has the same values as the deserialized record:  
{valuesAreTheSame}  
"");
```

In the preceding code:

- An `Appointment` object is instantiated and assigned to the `appointment` variable.
- The `appointment` instance is serialized to JSON using `JsonSerializer.Serialize`.
- The resulting JSON is written to the console.
- The JSON is deserialized back into a new instance of the `Appointment` type using `JsonSerializer.Deserialize`.
- The original and newly deserialized instances are compared for equality.
- The result of the comparison is written to the console.

For more information, see [How to serialize and deserialize JSON in .NET](#).

## Work with `TimeSpan` and `DateTime`

`TimeOnly` can be created from and converted to a `TimeSpan`. Also, `TimeOnly` can be used with a `DateTime`, either to create the `TimeOnly` instance, or to create a `DateTime` instance as long as a date is provided.

The following example creates a `TimeOnly` object from a `TimeSpan`, and then converts it back:

C#

```
// TimeSpan must be in the range of 00:00:00.0000000 to 23:59:59.9999999
var theTime = TimeOnly.FromTimeSpan(new TimeSpan(23, 59, 59));
var theTimeSpan = theTime.ToDateTime();

Console.WriteLine($"Variable '{nameof(theTime)}' is {theTime}");
Console.WriteLine($"Variable '{nameof(theTimeSpan)}' is {theTimeSpan}");

/* This example produces the following output:
 *
 * Variable 'theTime' is 11:59 PM
 * Variable 'theTimeSpan' is 23:59:59
 */
```

The following example creates a `DateTime` from a `TimeOnly` object, with an arbitrary date chosen:

C#

```
var theTime = new TimeOnly(11, 25, 46); // 11:25 PM and 46 seconds
var theDate = new DateOnly(2015, 10, 21); // October 21, 2015
var theDateTime = theDate.ToDateTime(theTime);
var reverseTime = TimeOnly.FromDateTime(theDateTime);

Console.WriteLine($"Date only is {theDate}");
Console.WriteLine($"Time only is {theTime}");
Console.WriteLine();
Console.WriteLine($"Combined to a DateTime type, the value is
{theDateTime}");
Console.WriteLine($"Converted back from DateTime, the time is
{reverseTime}");

/* This example produces the following output:
 *
 * Date only is 10/21/2015
 * Time only is 11:25 AM
 *
 * Combined to a DateTime type, the value is 10/21/2015 11:25:46 AM
 * Converted back from DateTime, the time is 11:25 AM
 */
```

## Arithmetic operators and comparing `TimeOnly`

Two `TimeOnly` instances can be compared with one another, and you can use the `IsBetween` method to check if a time is between two other times. When an addition or

subtraction operator is used on a `TimeOnly`, a `TimeSpan` is returned, representing a duration of time.

C#

```
var start = new TimeOnly(10, 12, 01); // 10:12:01 AM
var end = new TimeOnly(14, 00, 53); // 02:00:53 PM

var outside = start.AddMinutes(-3);
var inside = start.AddMinutes(120);

Console.WriteLine($"Time starts at {start} and ends at {end}");
Console.WriteLine($" Is {outside} between the start and end?
{outside.IsBetween(start, end)}");
Console.WriteLine($" Is {inside} between the start and end?
{inside.IsBetween(start, end)}");
Console.WriteLine($" Is {start} less than {end}? {start < end}");
Console.WriteLine($" Is {start} greater than {end}? {start > end}");
Console.WriteLine($" Does {start} equal {end}? {start == end}");
Console.WriteLine($" The time between {start} and {end} is {end - start}");

/* This example produces the following output:
 *
 * Time starts at 10:12 AM and ends at 2:00 PM
 * Is 10:09 AM between the start and end? False
 * Is 12:12 PM between the start and end? True
 * Is 10:12 AM less than 2:00 PM? True
 * Is 10:12 AM greater than 2:00 PM? False
 * Does 10:12 AM equal 2:00 PM? False
 * The time between 10:12 AM and 2:00 PM is 03:48:52
*/
```

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Performing arithmetic operations with dates and times

Article • 09/15/2021

Although both the [DateTime](#) and the [DateTimeOffset](#) structures provide members that perform arithmetic operations on their values, the results of arithmetic operations are very different. This article examines those differences, relates them to degrees of time zone awareness in date and time data, and discusses how to perform fully time zone aware operations using date and time data.

## Comparisons and arithmetic operations with DateTime values

The [DateTime.Kind](#) property allows a [DateTimeKind](#) value to be assigned to the date and time to indicate whether it represents local time, Coordinated Universal Time (UTC), or the time in an unspecified time zone. However, this limited time zone information is ignored when comparing or performing date and time arithmetic on [DateTimeKind](#) values. The following example, which compares the current local time with the current UTC time, illustrates how the time zone information is ignored.

C#

```
using System;

public enum TimeComparison
{
    EarlierThan = -1,
    TheSameAs = 0,
    LaterThan = 1
}

public class DateManipulation
{
    public static void Main()
    {
        DateTime localTime = DateTime.Now;
        DateTime utcTime = DateTime.UtcNow;

        Console.WriteLine("Difference between {0} and {1} time: {2}:{3} hours",
            localTime.Kind,
            utcTime.Kind,
            (localTime - utcTime).Hours,
            (localTime - utcTime).Minutes);
    }
}
```

```

        Console.WriteLine("The {0} time is {1} the {2} time.",
                           localTime.Kind,
                           Enum.GetName(typeof(TimeComparison),
                           localTime.CompareTo(utcTime)),
                           utcTime.Kind);
    }
}

// If run in the U.S. Pacific Standard Time zone, the example displays
// the following output to the console:
//   Difference between Local and Utc time: -7:0 hours
//   The Local time is EarlierThan the Utc time.

```

The [CompareTo\(DateTime\)](#) method reports that the local time is earlier than (or less than) the UTC time, and the subtraction operation indicates that the difference between UTC and the local time for a system in the U.S. Pacific Standard Time zone is seven hours. But because these two values provide different representations of a single point in time, it's clear in this case that the time interval is completely attributable to the local time zone's offset from UTC.

More generally, the [DateTime.Kind](#) property does not affect the results returned by [Kind](#) comparison and arithmetic methods (as the comparison of two identical points in time indicates), although it can affect the interpretation of those results. For example:

- The result of any arithmetic operation performed on two date and time values whose [DateTime.Kind](#) properties both equal [DateTimeKind](#) reflects the actual time interval between the two values. Similarly, the comparison of two such date and time values accurately reflects the relationship between times.
- The result of any arithmetic or comparison operation performed on two date and time values whose [DateTime.Kind](#) properties both equal [DateTimeKind](#) or on two date and time values with different [DateTime.Kind](#) property values reflects the difference in clock time between the two values.
- Arithmetic or comparison operations on local date and time values do not consider whether a particular value is ambiguous or invalid, nor do they take account of the effect of any adjustment rules that result from the local time zone's transition to or from daylight saving time.
- Any operation that compares or calculates the difference between UTC and a local time includes a time interval equal to the local time zone's offset from UTC in the result.
- Any operation that compares or calculates the difference between an unspecified time and either UTC or the local time reflects simple clock time. Time zone

differences are not considered, and the result does not reflect the application of time zone adjustment rules.

- Any operation that compares or calculates the difference between two unspecified times may include an unknown interval that reflects the difference between the time in two different time zones.

There are many scenarios in which time zone differences do not affect date and time calculations (for a discussion of some of these scenarios, see [Choosing between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo](#)) or in which the context of the date and time data defines the meaning of comparison or arithmetic operations.

## Comparisons and arithmetic operations with DateTimeOffset values

A [DateTimeOffset](#) value includes not only a date and time, but also an offset that unambiguously defines that date and time relative to UTC. This offset makes it possible to define equality differently than for [DateTime](#) values. Whereas [DateTime](#) values are equal if they have the same date and time value, [DateTimeOffset](#) values are equal if they both refer to the same point in time. When used in comparisons and in most arithmetic operations that determine the interval between two dates and times, a [DateTimeOffset](#) value is more accurate and less in need of interpretation. The following example, which is the [DateTimeOffset](#) equivalent to the previous example that compared local and UTC [DateTimeOffset](#) values, illustrates this difference in behavior.

```
C#  
  
using System;  
  
public enum TimeComparison  
{  
    EarlierThan = -1,  
    TheSameAs = 0,  
    LaterThan = 1  
}  
  
public class DateTimeOffsetManipulation  
{  
    public static void Main()  
    {  
        DateTimeOffset localTime = DateTimeOffset.Now;  
        DateTimeOffset utcTime = DateTimeOffset.UtcNow;  
  
        Console.WriteLine("Difference between local time and UTC: {0}:{1:D2}  
hours",  
            (localTime - utcTime).Hours,
```

```

        (localTime - utcTime).Minutes);
    Console.WriteLine("The local time is {0} UTC.",
        Enum.GetName(typeof(TimeComparison),
    localTime.CompareTo(utcTime)));
}
}
// Regardless of the local time zone, the example displays
// the following output to the console:
//    Difference between local time and UTC: 0:00 hours.
//    The local time is TheSameAs UTC.

```

In this example, the `CompareTo` method indicates that the current local time and the current UTC time are equal, and subtraction of `CompareTo(DateTimeOffset)` values indicates that the difference between the two times is `TimeSpan.Zero`.

The chief limitation of using `DateTimeOffset` values in date and time arithmetic is that although `DateTimeOffset` values have some time zone awareness, they are not fully time zone aware. Although the `DateTimeOffset` value's offset reflects a time zone's offset from UTC when a `DateTimeOffset` variable is first assigned a value, it becomes disassociated from the time zone thereafter. Because it is no longer directly associated with an identifiable time, the addition and subtraction of date and time intervals does not consider a time zone's adjustment rules.

To illustrate, the transition to daylight saving time in the U.S. Central Standard Time zone occurs at 2:00 A.M. on March 9, 2008. With that in mind, adding a two and a half hour interval to a Central Standard time of 1:30 A.M. on March 9, 2008, should produce a date and time of 5:00 A.M. on March 9, 2008. However, as the following example shows, the result of the addition is 4:00 A.M. on March 9, 2008. The result of this operation does represent the correct point in time, although it is not the time in the time zone in which we are interested (that is, it does not have the expected time zone offset).

C#

```

using System;

public class IntervalArithmetic
{
    public static void Main()
    {
        DateTime generalTime = new DateTime(2008, 3, 9, 1, 30, 0);
        const string tzName = "Central Standard Time";
        TimeSpan twoAndAHalfHours = new TimeSpan(2, 30, 0);

        // Instantiate DateTimeOffset value to have correct CST offset
        try
        {
            DateTimeOffset centralTime1 = new DateTimeOffset(generalTime,

```

```

TimeZoneInfo.FindSystemTimeZoneById(tzName).GetUtcOffset(generalTime));

    // Add two and a half hours
    DateTimeOffset centralTime2 = centralTime1.Add(twoAndAHalfHours);
    // Display result
    Console.WriteLine("{0} + {1} hours = {2}", centralTime1,
                      twoAndAHalfHours.ToString(),
                      centralTime2);
}
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("Unable to retrieve Central Standard Time zone
information.");
}
}

// The example displays the following output to the console:
//      3/9/2008 1:30:00 AM -06:00 + 02:30:00 hours = 3/9/2008 4:00:00 AM
-06:00

```

## Arithmetic operations with times in time zones

The [TimeZoneInfo](#) class includes conversion methods that automatically apply adjustments when they convert times from one time zone to another. These conversion methods include:

- The [ConvertTime](#) and [ConvertTimeBySystemTimeZoneId](#) methods, which convert times between any two time zones.
- The [ConvertTimeFromUtc](#) and [ConvertTimeToUtc](#) methods, which convert the time in a particular time zone to UTC, or convert UTC to the time in a particular time zone.

For details, see [Converting times between time zones](#).

The [TimeZoneInfo](#) class does not provide any methods that automatically apply adjustment rules when you perform date and time arithmetic. However, you can apply adjustment rules by converting the time in a time zone to UTC, performing the arithmetic operation, and then converting from UTC back to the time in the time zone.

For details, see [How to: Use time zones in date and time arithmetic](#).

For example, the following code is similar to the previous code that added two-and-a-half hours to 2:00 A.M. on March 9, 2008. However, because it converts a Central Standard time to UTC before it performs date and time arithmetic, and then converts the

result from UTC back to Central Standard time, the resulting time reflects the Central Standard Time Zone's transition to daylight saving time.

```
C#
```

```
using System;

public class TimeZoneAwareArithmetic
{
    public static void Main()
    {
        const string tzName = "Central Standard Time";

        DateTime generalTime = new DateTime(2008, 3, 9, 1, 30, 0);
        TimeZoneInfo cst = TimeZoneInfo.FindSystemTimeZoneById(tzName);
        TimeSpan twoAndAHalfHours = new TimeSpan(2, 30, 0);

        // Instantiate DateTimeOffset value to have correct CST offset
        try
        {
            DateTimeOffset centralTime1 = new DateTimeOffset(generalTime,
                cst.GetUtcOffset(generalTime));

            // Add two and a half hours
            DateTimeOffset utcTime = centralTime1.ToUniversalTime();
            utcTime += twoAndAHalfHours;

            DateTimeOffset centralTime2 = TimeZoneInfo.ConvertTime(utcTime,
                cst);
            // Display result
            Console.WriteLine("{0} + {1} hours = {2}", centralTime1,
                twoAndAHalfHours.ToString(),
                centralTime2);
        }
        catch (TimeZoneNotFoundException)
        {
            Console.WriteLine("Unable to retrieve Central Standard Time zone
information.");
        }
    }
}

// The example displays the following output to the console:
// 3/9/2008 1:30:00 AM -06:00 + 02:30:00 hours = 3/9/2008 5:00:00 AM
-05:00
```

## See also

- [Dates, times, and time zones](#)
- [How to: Use time zones in date and time arithmetic](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# DateTime and DateTimeOffset support in System.Text.Json

Article • 01/12/2023

The `System.Text.Json` library parses and writes `DateTime` and `DateTimeOffset` values according to the ISO 8601-1:2019 extended profile. `Converters` provide custom support for serializing and deserializing with `JsonSerializer`. You can also use `Utf8JsonReader` and `Utf8JsonWriter` to implement custom support.

## Support for the ISO 8601-1:2019 format

The `JsonSerializer`, `Utf8JsonReader`, `Utf8JsonWriter`, and `JsonElement` types parse and write `DateTime` and `DateTimeOffset` text representations according to the extended profile of the ISO 8601-1:2019 format. For example, `2019-07-26T16:59:57-05:00`.

`DateTime` and `DateTimeOffset` data can be serialized with `JsonSerializer`:

C#

```
using System.Text.Json;

public class Example
{
    private class Product
    {
        public string? Name { get; set; }
        public DateTime ExpiryDate { get; set; }
    }

    public static void Main(string[] args)
    {
        Product p = new Product();
        p.Name = "Banana";
        p.ExpiryDate = new DateTime(2019, 7, 26);

        string json = JsonSerializer.Serialize(p);
        Console.WriteLine(json);
    }
}

// The example displays the following output:
// {"Name":"Banana","ExpiryDate":"2019-07-26T00:00:00"}
```

`DateTime` and `DateTimeOffset` can also be deserialized with `JsonSerializer`:

C#

```
using System.Text.Json;

public class Example
{
    private class Product
    {
        public string? Name { get; set; }
        public DateTime ExpiryDate { get; set; }
    }

    public static void Main(string[] args)
    {
        string json = @"{""Name"":""Banana"" , ""ExpiryDate"":""2019-07-26T00:00:00""}";
        Product p = JsonSerializer.Deserialize<Product>(json)!;
        Console.WriteLine(p.Name);
        Console.WriteLine(p.ExpiryDate);
    }
}

// The example displays output similar to the following:
// Banana
// 7/26/2019 12:00:00 AM
```

With default options, input `DateTime` and `DateTimeOffset` text representations must conform to the extended ISO 8601-1:2019 profile. Attempting to deserialize representations that don't conform to the profile will cause `JsonSerializer` to throw a `JsonException`:

C#

```
using System.Text.Json;

public class Example
{
    private class Product
    {
        public string? Name { get; set; }
        public DateTime ExpiryDate { get; set; }
    }

    public static void Main(string[] args)
    {
        string json = @"
{""Name"":""Banana"" , ""ExpiryDate"":""26/07/2019""}";
        try
        {
            Product _ = JsonSerializer.Deserialize<Product>(json)!;
        }
        catch (JsonException e)
```

```

        {
            Console.WriteLine(e.Message);
        }
    }

// The example displays the following output:
// The JSON value could not be converted to System.DateTime. Path:
$.ExpiryDate | LineNumber: 0 | BytePositionInLine: 42.

```

The [JsonDocument](#) provides structured access to the contents of a JSON payload, including [DateTime](#) and [DateTimeOffset](#) representations. The following example shows how to calculate the average temperature on Mondays from a collection of temperatures:

C#

```

using System.Text.Json;

public class Example
{
    private static double ComputeAverageTemperatures(string json)
    {
        JsonDocumentOptions options = new JsonDocumentOptions
        {
            AllowTrailingCommas = true
        };

        using (JsonDocument document = JsonDocument.Parse(json, options))
        {
            int sumOfAllTemperatures = 0;
            int count = 0;

            foreach (JsonElement element in
document.RootElement.EnumerateArray())
            {
                DateTimeOffset date =
element.GetProperty("date").GetDateTimeOffset();

                if (date.DayOfWeek == DayOfWeek.Monday)
                {
                    int temp = element.GetProperty("temp").GetInt32();
                    sumOfAllTemperatures += temp;
                    count++;
                }
            }

            double averageTemp = (double)sumOfAllTemperatures / count;
            return averageTemp;
        }
    }
}

```

```

public static void Main(string[] args)
{
    string json =
        @"[ " +
        @"{ " +
            @"""date"": """2013-01-07T00:00:00Z""", " +
            @"""temp"": 23, " +
        @"}, " +
        @"{ " +
            @"""date"": """2013-01-08T00:00:00Z""", " +
            @"""temp"": 28, " +
        @"}, " +
        @"{ " +
            @"""date"": """2013-01-14T00:00:00Z""", " +
            @"""temp"": 8, " +
        @"}, " +
        @"]";

    Console.WriteLine(ComputeAverageTemperatures(json));
}

// The example displays the following output:
// 15.5

```

Attempting to compute the average temperature given a payload with non-compliant `DateTime` representations will cause `JsonDocument` to throw a [FormatException](#):

C#

```

using System.Text.Json;

public class Example
{
    private static double ComputeAverageTemperatures(string json)
    {
        JsonDocumentOptions options = new JsonDocumentOptions
        {
            AllowTrailingCommas = true
        };

        using (JsonDocument document = JsonDocument.Parse(json, options))
        {
            int sumOfAllTemperatures = 0;
            int count = 0;

            foreach (JsonElement element in
document.RootElement.EnumerateArray())
            {
                DateTimeOffset date =
element.GetProperty("date").GetDateTimeOffset();

                if (date.DayOfWeek == DayOfWeek.Monday)

```

```

        {
            int temp = element.GetProperty("temp").GetInt32();
            sumOfAllTemperatures += temp;
            count++;
        }
    }

    double averageTemp = (double)sumOfAllTemperatures / count;
    return averageTemp;
}

public static void Main(string[] args)
{
    // Computing the average temperatures will fail because the
    DateTimeOffset
    // values in the payload do not conform to the extended ISO 8601-
    1:2019 profile.
    string json =
        @"[ " +
        @"{ " +
        @"""date"": ""2013/01/07 00:00:00Z""", " +
        @"""temp"": 23, " +
        @"}, " +
        @"{ " +
        @"""date"": ""2013/01/08 00:00:00Z""", " +
        @"""temp"": 28, " +
        @"}, " +
        @"{ " +
        @"""date"": ""2013/01/14 00:00:00Z""", " +
        @"""temp"": 8, " +
        @"}, " +
        @"]";

    Console.WriteLine(ComputeAverageTemperatures(json));
}
}

// The example displays the following output:
// Unhandled exception.System.FormatException: One of the identified items
// was in an invalid format.
//     at System.Text.Json.JsonElement.GetDateTimeOffset()

```

The lower level [Utf8JsonWriter](#) writes [DateTime](#) and [DateTimeOffset](#) data:

```

C#

using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)

```

```

    {
        JsonWriterOptions options = new JsonWriterOptions
        {
            Indented = true
        };

        using (MemoryStream stream = new MemoryStream())
        {
            using (Utf8JsonWriter writer = new Utf8JsonWriter(stream,
options))
            {
                writer.WriteStartObject();
                writer.WriteString("date", DateTimeOffset.UtcNow);
                writer.WriteNumber("temp", 42);
                writer.WriteEndObject();
            }

            string json = Encoding.UTF8.GetString(stream.ToArray());
            Console.WriteLine(json);
        }
    }

    // The example output similar to the following:
    // {
    //     "date": "2019-07-26T00:00:00+00:00",
    //     "temp": 42
    // }
}

```

[Utf8JsonReader](#) parses [DateTime](#) and [DateTimeOffset](#) data:

C#

```

using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        byte[] utf8Data = Encoding.UTF8.GetBytes(@"""2019-07-
26T00:00:00""");

        Utf8JsonReader json = new Utf8JsonReader(utf8Data);
        while (json.Read())
        {
            if (json.TokenType == JsonTokenType.String)
            {
                Console.WriteLine(json.TryGetDateTime(out DateTime
datetime));
                Console.WriteLine(datetime);
                Console.WriteLine(json.GetDateTime());
            }
        }
    }
}

```

```

        }
    }

// The example displays output similar to the following:
// True
// 7/26/2019 12:00:00 AM
// 7/26/2019 12:00:00 AM

```

Attempting to read non-compliant formats with [Utf8JsonReader](#) will cause it to throw a [FormatException](#):

C#

```

using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        byte[] utf8Data = Encoding.UTF8.GetBytes(@"""2019/07/26
00:00:00""");

        Utf8JsonReader json = new Utf8JsonReader(utf8Data);
        while (json.Read())
        {
            if (json.TokenType == JsonTokenType.String)
            {
                Console.WriteLine(json.TryGetDateTime(out DateTime
datetime));
                Console.WriteLine(datetime);

                DateTime _ = json.GetDateTime();
            }
        }
    }

// The example displays the following output:
// False
// 1/1/0001 12:00:00 AM
// Unhandled exception. System.FormatException: The JSON value is not in a
supported DateTime format.
//      at System.Text.Json.Utf8JsonReader.GetDateTime()

```

## Serialize DateOnly and TimeOnly properties

With .NET 7+, `System.Text.Json` supports serializing and deserializing [DateOnly](#) and [TimeOnly](#) types. Consider the following object:

C#

```
sealed file record Appointment(
    Guid Id,
    string Description,
    DateOnly Date,
    TimeOnly StartTime,
    TimeOnly EndTime);
```

The following example serializes an `Appointment` object, displays the resulting JSON, and then deserializes it back into a new instance of the `Appointment` type. Finally, the original and newly deserialized instances are compared for equality and the results are written to the console:

C#

```
Appointment originalAppointment = new(
    Id: Guid.NewGuid(),
    Description: "Take dog to veterinarian.",
    Date: new DateOnly(2002, 1, 13),
    StartTime: new TimeOnly(5,15),
    EndTime: new TimeOnly(5, 45));
string serialized = JsonSerializer.Serialize(originalAppointment);

Console.WriteLine($"Resulting JSON: {serialized}");

Appointment deserializedAppointment =
    JsonSerializer.Deserialize<Appointment>(serialized)!;

bool valuesAreTheSame = originalAppointment == deserializedAppointment;
Console.WriteLine($"""
    Original record has the same values as the deserialized record:
{valuesAreTheSame}
""");
```

In the preceding code:

- An `Appointment` object is instantiated and assigned to the `appointment` variable.
- The `appointment` instance is serialized to JSON using `JsonSerializer.Serialize`.
- The resulting JSON is written to the console.
- The JSON is deserialized back into a new instance of the `Appointment` type using `JsonSerializer.Deserialize`.
- The original and newly deserialized instances are compared for equality.
- The result of the comparison is written to the console.

# Custom support for `DateTime` and `DateTimeOffset`

## When using `JsonSerializer`

If you want the serializer to perform custom parsing or formatting, you can implement [custom converters](#). Here are a few examples:

### `DateTimeOffset.Parse` and `DateTimeOffset.ToString`

If you can't determine the formats of your input `DateTime` or `DateTimeOffset` text representations, you can use the `DateTimeOffset.Parse` method in your converter read logic. This method allows you to use .NET's extensive support for parsing various `DateTime` and `DateTimeOffset` text formats, including non-ISO 8601 strings and ISO 8601 formats that don't conform to the extended ISO 8601-1:2019 profile. This approach is less performant than using the serializer's native implementation.

For serializing, you can use the `DateTimeOffset.ToString` method in your converter write logic. This method allows you to write `DateTime` and `DateTimeOffset` values using any of the [standard date and time formats](#), and the [custom date and time formats](#). This approach is also less performant than using the serializer's native implementation.

C#

```
using System.Diagnostics;
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Text.RegularExpressions;

namespace DateTimeConverterExamples;

public class DateTimeConverterUsingDateTimeParse : JsonConverter<DateTime>
{
    public override DateTime Read(ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
    {
        Debug.Assert(typeToConvert == typeof(DateTime));
        return DateTime.Parse(reader.GetString() ?? string.Empty);
    }

    public override void Write(Utf8JsonWriter writer, DateTime value, JsonSerializerOptions options)
    {
        writer.WriteStringValue(value.ToString());
    }
}
```

```
class Program
{
    private static void ParseDateTimeWithDefaultOptions()
    {
        DateTime _ = JsonSerializer.Deserialize<DateTime>(@"""04-10-2008
6:30 AM""");
    }

    private static void FormatDateTimeWithDefaultOptions()
    {
        Console.WriteLine(JsonSerializer.Serialize(DateTime.Parse("04-10-
2008 6:30 AM -4")));
    }

    private static void ProcessDateTimeWithCustomConverter()
    {
        JsonSerializerOptions options = new JsonSerializerOptions();
        options.Converters.Add(new DateTimeConverterUsingDateTimeParse());

        string testDateTimeStr = "04-10-2008 6:30 AM";
        string testDateTimeJson = @""" + testDateTimeStr + @"""";

        DateTime resultDateTime = JsonSerializer.Deserialize<DateTime>
(testDateTimeJson, options);
        Console.WriteLine(resultDateTime);

        string resultDateTimeJson =
JsonSerializer.Serialize(DateTime.Parse(testDateTimeStr), options);
        Console.WriteLine(Regex.Unescape(resultDateTimeJson));
    }

    static void Main(string[] args)
    {
        // Parsing non-compliant format as DateTime fails by default.
        try
        {
            ParseDateTimeWithDefaultOptions();
        }
        catch (JsonException e)
        {
            Console.WriteLine(e.Message);
        }

        // Formatting with default options prints according to extended ISO
        // 8601 profile.
        FormatDateTimeWithDefaultOptions();

        // Using converters gives you control over the serializers parsing
        // and formatting.
        ProcessDateTimeWithCustomConverter();
    }
}

// The example displays output similar to the following:
```

```
// The JSON value could not be converted to System.DateTime. Path: $ |  
LineNumber: 0 | BytePositionInLine: 20.  
// "2008-04-10T06:30:00-04:00"  
// 4/10/2008 6:30:00 AM  
// "4/10/2008 6:30:00 AM"
```

### ⓘ Note

When implementing `JsonConverter<T>`, and `T` is `DateTime`, the `typeToConvert` parameter will always be `typeof(DateTime)`. The parameter is useful for handling polymorphic cases and when using generics to get `typeof(T)` in a performant way.

## Utf8Parser and Utf8Formatter

You can use fast UTF-8-based parsing and formatting methods in your converter logic if your input `DateTime` or `DateTimeOffset` text representations are compliant with one of the "R", "I", "O", or "G" [standard date and time format strings](#), or you want to write according to one of these formats. This approach is much faster than using `S DateTime(Offset).Parse` and `DateTime(Offset).ToString`.

The following example shows a custom converter that serializes and deserializes `DateTime` values according to [the "R" standard format](#):

C#

```
using System.Buffers;  
using System.Buffers.Text;  
using System.Diagnostics;  
using System.Text.Json;  
using System.Text.Json.Serialization;  
  
namespace DateTimeConverterExamples;  
  
// This converter reads and writes DateTime values according to the "R"  
// standard format specifier:  
// https://learn.microsoft.com/dotnet/standard/base-types/standard-date-and-time-format-strings#the-rfc1123-r-r-format-specifier.  
public class DateTimeConverterForCustomStandardFormatR :  
JsonConverter<DateTime>  
{  
    public override DateTime Read(ref Utf8JsonReader reader, Type  
typeToConvert, JsonSerializerOptions options)  
    {  
        Debug.Assert(typeToConvert == typeof(DateTime));  
  
        if (Utf8Parser.TryParse(reader.ValueSpan, out DateTime value, out _,  
'R'))
```

```
        {
            return value;
        }

        throw new FormatException();
    }

    public override void Write(Utf8JsonWriter writer, DateTime value,
JsonSerializerOptions options)
{
    // The "R" standard format will always be 29 bytes.
    Span<byte> utf8Date = new byte[29];

    bool result = Utf8Formatter.TryFormat(value, utf8Date, out _, new
StandardFormat('R'));
    Debug.Assert(result);

    writer.WriteStringValue(utf8Date);
}
}

class Program
{
    private static void ParseDateTimeWithDefaultOptions()
    {
        DateTime _ = JsonSerializer.Deserialize<DateTime>(@"""Thu, 25 Jul
2019 13:36:07 GMT""");
    }

    private static void ProcessDateTimeWithCustomConverter()
    {
        JsonSerializerOptions options = new JsonSerializerOptions();
        options.Converters.Add(new
DateTimeConverterForCustomStandardFormatR());

        string testDateTimeStr = "Thu, 25 Jul 2019 13:36:07 GMT";
        string testDateTimeJson = @""" + testDateTimeStr + @"""";

        DateTime resultDateTime = JsonSerializer.Deserialize<DateTime>
(testDateTimeJson, options);
        Console.WriteLine(resultDateTime);

        Console.WriteLine(JsonSerializer.Serialize(DateTime.Parse(testDateTimeStr),
options));
    }

    static void Main(string[] args)
    {
        // Parsing non-compliant format as DateTime fails by default.
        try
        {
            ParseDateTimeWithDefaultOptions();
        }
        catch (JsonException e)
    }
}
```

```

    {
        Console.WriteLine(e.Message);
    }

    // Using converters gives you control over the serializers parsing
    // and formatting.
    ProcessDateTimeWithCustomConverter();
}
}

// The example displays output similar to the following:
// The JSON value could not be converted to System.DateTime.Path: $ | 
LineNumber: 0 | BytePositionInLine: 31.
// 7/25/2019 1:36:07 PM
// "Thu, 25 Jul 2019 09:36:07 GMT"

```

### ⓘ Note

The "R" standard format will always be 29 characters long.

The "l" (lowercase "L") format isn't documented with the other **standard date and time format strings** because it's supported only by the `Utf8Parser` and `Utf8Formatter` types. The format is lowercase RFC 1123 (a lowercase version of the "R" format). For example, "thu, 25 jul 2019 06:36:07 gmt".

## Use `DateTime(Offset).Parse` as a fallback

If you generally expect your input `DateTime` or `DateTimeOffset` data to conform to the extended ISO 8601-1:2019 profile, you can use the serializer's native parsing logic. You can also implement a fallback mechanism. The following example shows that, after failing to parse a `DateTime` text representation using `TryGetDateTime(DateTime)`, the converter successfully parses the data using `Parse(String)`:

C#

```

using System.Diagnostics;
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Text.RegularExpressions;

namespace DateTimeConverterExamples;

public class DateTimeConverterUsingDateTimeParseAsFallback :
JsonConverter<DateTime>
{
    public override DateTime Read(ref Utf8JsonReader reader, Type
typeToConvert, JsonSerializerOptions options)

```

```
    {
        Debug.Assert(typeToConvert == typeof(DateTime));

        if (!reader.TryGetDateTime(out DateTime value))
        {
            value = DateTime.Parse(reader.GetString()!);
        }

        return value;
    }

    public override void Write(Utf8JsonWriter writer, DateTime value,
JsonSerializerOptions options)
{
    writer.WriteStringValue(value.ToString("dd/MM/yyyy"));
}
}

class Program
{
    private static void ParseDateTimeWithDefaultOptions()
    {
        DateTime _ = JsonSerializer.Deserialize<DateTime>(@"""2019-07-16
16:45:27.4937872+00:00""");
    }

    private static void ProcessDateTimeWithCustomConverter()
    {
        JsonSerializerOptions options = new JsonSerializerOptions();
        options.Converters.Add(new
DateTimeConverterUsingDateTimeParseAsFallback());

        string testDateTimeStr = "2019-07-16 16:45:27.4937872+00:00";
        string testDateTimeJson = @""" + testDateTimeStr + @"""";

        DateTime resultDateTime = JsonSerializer.Deserialize<DateTime>
(testDateTimeJson, options);
        Console.WriteLine(resultDateTime);

        string resultDateTimeJson =
JsonSerializer.Serialize(DateTime.Parse(testDateTimeStr), options);
        Console.WriteLine(Regex.Unescape(resultDateTimeJson));
    }

    static void Main(string[] args)
{
    // Parsing non-compliant format as DateTime fails by default.
    try
    {
        ParseDateTimeWithDefaultOptions();
    }
    catch (JsonException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

```

        // Using converters gives you control over the serializers parsing
        // and formatting.
        ProcessDateTimeWithCustomConverter();
    }

}

// The example displays output similar to the following:
// The JSON value could not be converted to System.DateTime.Path: $ |
// LineNumber: 0 | BytePositionInLine: 35.
// 7/16/2019 4:45:27 PM
// "16/07/2019"

```

## Use Unix epoch date format

The following converters handle Unix epoch format with or without a time zone offset (values such as `/Date(1590863400000-0700)/` or `/Date(1590863400000)/`):

C#

```

sealed class UnixEpochDateTimeOffsetConverter :
JsonConverter<DateTimeOffset>
{
    static readonly DateTimeOffset s_epoch = new DateTimeOffset(1970, 1, 1,
0, 0, 0, TimeSpan.Zero);
    static readonly Regex s_regex = new Regex("^/Date\\\"(([+-]*)((\\d{2})\\(\\d{2})\\))\\\"/", RegexOptions.CultureInvariant);

    public override DateTimeOffset Read(ref Utf8JsonReader reader, Type
typeToConvert, JsonSerializerOptions options)
    {
        string formatted = reader.GetString()!;
        Match match = s_regex.Match(formatted);

        if (
            !match.Success
            || !long.TryParse(match.Groups[1].Value,
System.Globalization.NumberStyles.Integer, CultureInfo.InvariantCulture, out
long unixTime)
            || !int.TryParse(match.Groups[3].Value,
System.Globalization.NumberStyles.Integer, CultureInfo.InvariantCulture, out
int hours)
            || !int.TryParse(match.Groups[4].Value,
System.Globalization.NumberStyles.Integer, CultureInfo.InvariantCulture, out
int minutes))
        {
            throw new JsonException();
        }

        int sign = match.Groups[2].Value[0] == '+' ? 1 : -1;
        TimeSpan utcOffset = new TimeSpan(hours * sign, minutes * sign, 0);
    }
}

```

```

        return s_epoch.AddMilliseconds(unixTime).ToOffset(utcOffset);
    }

    public override void Write(Utf8JsonWriter writer, DateTimeOffset value,
JsonSerializerOptions options)
{
    long unixTime = Convert.ToInt64((value -
s_epoch).TotalMilliseconds);
    TimeSpan utcOffset = value.Offset;

    string formatted = string.Create(CultureInfo.InvariantCulture,
$""/Date({unixTime}{{(utcOffset >= TimeSpan.Zero ? "+" : "-")}{utcOffset:hhmm}})/");
    writer.WriteStringValue(formatted);
}
}

```

C#

```

sealed class UnixEpochDateTimeConverter : JsonConverter<DateTime>
{
    static readonly DateTime s_epoch = new DateTime(1970, 1, 1, 0, 0, 0);
    static readonly Regex s_regex = new Regex("^\u002fDate\\u0024(([-]+)*\\d+)\\u0024", RegexOptions.CultureInvariant);

    public override DateTime Read(ref Utf8JsonReader reader, Type
typeToConvert, JsonSerializerOptions options)
    {
        string formatted = reader.GetString()!;
        Match match = s_regex.Match(formatted);

        if (
            !match.Success
            || !long.TryParse(match.Groups[1].Value,
System.Globalization.NumberStyles.Integer, CultureInfo.InvariantCulture, out
long unixTime))
        {
            throw new JsonException();
        }

        return s_epoch.AddMilliseconds(unixTime);
    }

    public override void Write(Utf8JsonWriter writer, DateTimeOffset value,
JsonSerializerOptions options)
    {
        long unixTime = Convert.ToInt64((value -
s_epoch).TotalMilliseconds);

        string formatted = string.Create(CultureInfo.InvariantCulture,
$""/Date({unixTime})/");
    }
}

```

```
        writer.WriteStringValue(formatted);
    }
}
```

## When using [Utf8JsonWriter](#)

If you want to write a custom [DateTime](#) or [DateTimeOffset](#) text representation with [Utf8JsonWriter](#), you can format your custom representation to a [String](#), [ReadOnlySpan<Byte>](#), [ReadOnlySpan<Char>](#), or [JsonEncodedText](#), then pass it to the corresponding [Utf8JsonWriter.WriteStringValue](#) or [Utf8JsonWriter.WriteString](#) method.

The following example shows how a custom [DateTime](#) format can be created with [ToString\(String, IFormatProvider\)](#) and then written with the [WriteStringValue\(String\)](#) method:

C#

```
using System.Globalization;
using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        var options = new JsonWriterOptions
        {
            Indented = true
        };

        using (var stream = new MemoryStream())
        {
            using (var writer = new Utf8JsonWriter(stream, options))
            {
                string dateStr = DateTime.UtcNow.ToString("F",
CultureInfo.InvariantCulture);

                writer.WriteStartObject();
                writer.WriteString("date", dateStr);
                writer.WriteNumber("temp", 42);
                writer.WriteEndObject();
            }

            string json = Encoding.UTF8.GetString(stream.ToArray());
            Console.WriteLine(json);
        }
    }
}

// The example displays output similar to the following:
```

```
// {
//     "date": "Tuesday, 27 August 2019 19:21:44",
//     "temp": 42
// }
```

## When using [Utf8JsonReader](#)

If you want to read a custom [DateTime](#) or [DateTimeOffset](#) text representation with [Utf8JsonReader](#), you can get the value of the current JSON token as a [String](#) using [GetString\(\)](#) method, then parse the value using custom logic.

The following example shows how a custom [DateTimeOffset](#) text representation can be retrieved using the [GetString\(\)](#) method, then parsed using [ParseExact\(String, String, IFormatProvider\)](#):

C#

```
using System.Globalization;
using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        byte[] utf8Data = Encoding.UTF8.GetBytes(@"""Friday, 26 July 2019
00:00:00""");

        var json = new Utf8JsonReader(utf8Data);
        while (json.Read())
        {
            if (json.TokenType == JsonTokenType.String)
            {
                string value = json.GetString();
                DateTimeOffset dto = DateTimeOffset.ParseExact(value, "F",
CultureInfo.InvariantCulture);
                Console.WriteLine(dto);
            }
        }
    }
}

// The example displays output similar to the following:
// 7/26/2019 12:00:00 AM -04:00
```

## The extended ISO 8601-1:2019 profile in [System.Text.Json](#)

# Date and time components

The extended ISO 8601-1:2019 profile implemented in [System.Text.Json](#) defines the following components for date and time representations. These components are used to define various supported levels of granularity when parsing and formatting [DateTime](#) and [DateTimeOffset](#) representations.

Component	Format	Description
Year	"yyyy"	0001-9999
Month	"MM"	01-12
Day	"dd"	01-28, 01-29, 01-30, 01-31 based on month/year.
Hour	"HH"	00-23
Minute	"mm"	00-59
Second	"ss"	00-59
Second fraction	"FFFFFF"	Minimum of one digit, maximum of 16 digits.
Time offset	"K"	Either "Z" or "(+/-)HH:mm".
Partial time	"HH:mm:ss[FFFFFF]"	Time without UTC offset information.
Full date	"yyyy'-'MM'-'dd"	Calendar date.
Full time	"Partial time'K"	UTC of day or Local time of day with the time offset between local time and UTC.
Date time	"Full date"T"Full time"	Calendar date and time of day, for example, 2019-07-26T16:59:57-05:00.

## Support for parsing

The following levels of granularity are defined for parsing:

1. 'Full date'
  - a. "yyyy'-'MM'-'dd"
2. "'Full date"T"Hour":"Minute"
  - a. "yyyy'-'MM'-'dd'T'HH':'mm"
3. "'Full date"T"Partial time"
  - a. "yyyy'-'MM'-'dd'T'HH':'mm':'ss" ([The Sortable \("s"\) Format Specifier](#))

- b. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.FFFFFFFFFF"
- 4. ""Full date" "T" "Time hour" ":" "Minute" "Time offset" ""
  - a. "yyyy'-'MM'-'dd'T'HH':'mmZ"
  - b. "yyyy'-'MM'-'dd'T'HH':'mm('+'/-')HH':'mm"
- 5. 'Date time'
  - a. "yyyy'-'MM'-'dd'T'HH':'mm':'ssZ"
  - b. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.FFFFFFFFFFZ"
  - c. "yyyy'-'MM'-'dd'T'HH':'mm':'ss('+'/-')HH':'mm"
  - d. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.FFFFFFFFFF('+'/-')HH':'mm"

This level of granularity is compliant with [RFC 3339](#), a widely adopted profile of ISO 8601 used for interchanging date and time information. However, there are a few restrictions in the `System.Text.Json` implementation.

- RFC 3339 doesn't specify a maximum number of fractional-second digits, but specifies that at least one digit must follow the period, if a fractional-second section is present. The implementation in `System.Text.Json` allows up to 16 digits (to support interop with other programming languages and frameworks), but parses only the first seven. A `JsonException` will be thrown if there are more than 16 fractional second digits when reading `DateTime` and `DateTimeOffset` instances.
- RFC 3339 allows the "T" and "Z" characters to be "t" or "z" respectively, but allows applications to limit support to just the upper-case variants. The implementation in `System.Text.Json` requires them to be "T" and "Z". A `JsonException` will be thrown if input payloads contain "t" or "z" when reading `DateTime` and `DateTimeOffset` instances.
- RFC 3339 specifies that the date and time sections are separated by "T", but allows applications to separate them by a space (" ") instead. `System.Text.Json` requires date and time sections to be separated with "T". A `JsonException` will be thrown if input payloads contain a space (" ") when reading `DateTime` and `DateTimeOffset` instances.

If there are decimal fractions for seconds, there must be at least one digit. `2019-07-26T00:00:00.` isn't allowed. While up to 16 fractional digits are allowed, only the first seven are parsed. Anything beyond that is considered a zero. For example, `2019-07-26T00:00:00.1234567890` will be parsed as if it's `2019-07-26T00:00:00.1234567`. This approach maintains compatibility with the `DateTime` implementation, which is limited to this resolution.

Leap seconds aren't supported.

# Support for formatting

The following levels of granularity are defined for formatting:

1. "Full date" "Partial time"

a. "yyyy'-'MM'-'dd'T'HH':'mm':'ss" ([The Sortable \("s"\) Format Specifier](#))

Used to format a [DateTime](#) without fractional seconds and without offset information.

b. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.FFFFFFFFFF"

Used to format a [DateTime](#) with fractional seconds but without offset information.

2. 'Date time'

a. "yyyy'-'MM'-'dd'T'HH':'mm':'ssZ"

Used to format a [DateTime](#) without fractional seconds but with a UTC offset.

b. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.FFFFFFFFFFZ"

Used to format a [DateTime](#) with fractional seconds and with a UTC offset.

c. "yyyy'-'MM'-'dd'T'HH':'mm':'ss('+'/-')HH':'mm"

Used to format a [DateTime](#) or [DateTimeOffset](#) without fractional seconds but with a local offset.

d. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.FFFFFFFFFF('+'/-')HH':'mm"

Used to format a [DateTime](#) or [DateTimeOffset](#) with fractional seconds and with a local offset.

This level of granularity is compliant with [RFC 3339](#).

If the [round-trip format](#) representation of a [DateTime](#) or [DateTimeOffset](#) instance has trailing zeros in its fractional seconds, then [JsonSerializer](#) and [Utf8JsonWriter](#) will format a representation of the instance without trailing zeros. For example, a [DateTime](#) instance whose [round-trip format](#) representation is `2019-04-24T14:50:17.101000Z`, will be formatted as `2019-04-24T14:50:17.101Z` by [JsonSerializer](#) and [Utf8JsonWriter](#).

If the [round-trip format](#) representation of a [DateTime](#) or [DateTimeOffset](#) instance has all zeros in its fractional seconds, then [JsonSerializer](#) and [Utf8JsonWriter](#) will format a representation of the instance without fractional seconds. For example, a [DateTime](#)

instance whose [round-trip format](#) representation is `2019-04-24T14:50:17.0000000+02:00`, will be formatted as `2019-04-24T14:50:17+02:00` by [JsonSerializer](#) and [Utf8JsonWriter](#).

Truncating zeros in fractional-second digits allows the smallest output needed to preserve information on a round trip to be written.

A maximum of seven fractional-second digits are written. This maximum aligns with the [DateTime](#) implementation, which is limited to this resolution.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

## [.NET feedback](#)

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Time zone overview

Article • 02/15/2023

The [TimeZoneInfo](#) class simplifies the creation of time zone-aware applications. The [TimeZone](#) class supports working with the local time zone and Coordinated Universal Time (UTC). The [TimeZoneInfo](#) class supports both of these zones as well as any time zone about which information is predefined in the registry. You can also use [TimeZoneInfo](#) to define custom time zones that the system has no information about.

## Time zone essentials

A time zone is a geographical region in which the same time is used. Typically, but not always, adjacent time zones are one hour apart. The time in any of the world's time zones can be expressed as an offset from Coordinated Universal Time (UTC).

Many of the world's time zones support daylight saving time. Daylight saving time tries to maximize daylight hours by advancing the time forward by one hour in the spring or early summer, and returning to the normal (or standard) time in the late summer or fall. These changes to and from standard time are known as adjustment rules.

The transition to and from daylight saving time in a particular time zone can be defined either by a fixed or a floating adjustment rule. A fixed adjustment rule sets a particular date on which the transition to or from daylight saving time occurs each year. For example, a transition from daylight saving time to standard time that occurs each year on October 25 follows a fixed adjustment rule. Much more common are floating adjustment rules, which set a particular day of a particular week of a particular month for the transition to or from daylight saving time. For example, a transition from standard time to daylight saving time that occurs on the third Sunday of March follows a floating adjustment rule.

For time zones that support adjustment rules, the transition to and from daylight saving time creates two kinds of anomalous times: invalid times and ambiguous times. An invalid time is a nonexistent time created by the transition from standard time to daylight saving time. For example, if this transition occurs on a particular day at 2:00 A.M. and causes the time to change to 3:00 A.M., each time interval between 2:00 A.M. and 2:59:59 A.M. is invalid. An ambiguous time is a time that can be mapped to two different times in a single time zone. It is created by the transition from daylight saving time to standard time. For example, if this transition occurs on a particular day at 2:00 A.M. and causes the time to change to 1:00 A.M., each time interval between 1:00 A.M. and 1:59:59 A.M. can be interpreted as either a standard time or a daylight saving time.

# Time zone terminology

The following table defines terms commonly used when working with time zones and developing time zone-aware applications.

Term	Definition
Adjustment rule	A rule that defines when the transition from standard time to daylight saving time and back from daylight saving time to standard time occurs. Each adjustment rule has a start and end date that defines when the rule is in place (for example, the adjustment rule is in place from January 1, 1986, to December 31, 2006), a delta (the amount of time by which the standard time changes as a result of the application of the adjustment rule), and information about the specific date and time that the transitions are to occur during the adjustment period. Transitions can follow either a fixed rule or a floating rule.
Ambiguous time	A time that can be mapped to two different times in a single time zone. It occurs when the clock time is adjusted back in time, such as during the transition from a time zone's daylight saving time to its standard time. For example, if this transition occurs on a particular day at 2:00 A.M. and causes the time to change to 1:00 A.M., each time interval between 1:00 A.M. and 1:59:59 A.M. can be interpreted as either a standard time or a daylight saving time.
Fixed rule	An adjustment rule that sets a particular date for the transition to or from daylight saving time. For example, a transition from daylight saving time to standard time that occurs each year on October 25 follows a fixed adjustment rule.
Floating rule	An adjustment rule that sets a particular day of a particular week of a particular month for the transition to or from daylight saving time. For example, a transition from standard time to daylight saving time that occurs on the third Sunday of March follows a floating adjustment rule.
Invalid time	A nonexistent time that is an artifact of the transition from standard time to daylight saving time. It occurs when the clock time is adjusted forward in time, such as during the transition from a time zone's standard time to its daylight saving time. For example, if this transition occurs on a particular day at 2:00 A.M. and causes the time to change to 3:00 A.M., each time interval between 2:00 A.M. and 2:59:59 A.M. is invalid.
Transition time	Information about a specific time change, such as the change from daylight saving time to standard time or vice versa, in a particular time zone.

## Time zones and the `TimeZoneInfo` class

In .NET, a `TimeZoneInfo` object represents a time zone. The `TimeZoneInfo` class includes a `GetAdjustmentRules` method that returns an array of `TimeZoneInfo.AdjustmentRule` objects. Each element of this array provides information about the transition to and

from daylight saving time for a particular time period. (For time zones that do not support daylight saving time, the method returns an empty array.) Each [TimeZoneInfo.AdjustmentRule](#) object has a [DaylightTransitionStart](#) and a [DaylightTransitionEnd](#) property that defines the particular date and time of the transition to and from daylight saving time. The [IsFixedDateRule](#) property indicates whether that transition is fixed or floating.

.NET relies on time zone information provided by the Windows operating system and stored in the registry. Due to the number of the earth's time zones, not all existing time zones are represented in the registry. In addition, because the registry is a dynamic structure, predefined time zones can be added to or removed from it. Finally, the registry does not necessarily contain historic time zone data. For example, in Windows XP the registry contains data about only a single set of time zone adjustments. Windows Vista supports dynamic time zone data, which means that a single time zone can have multiple adjustment rules that apply to specific intervals of years. However, most time zones that are defined in the Windows Vista registry and support daylight saving time have only one or two predefined adjustment rules.

The dependence of the [TimeZoneInfo](#) class on the registry means that a time zone-aware application cannot be certain that a particular time zone is defined in the registry. As a result, the attempt to instantiate a specific time zone (other than the local time zone or the time zone that represents UTC) should use exception handling. It should also provide some method of letting the application to continue if a required [TimeZoneInfo](#) object cannot be instantiated from the registry.

To handle the absence of a required time zone, the [TimeZoneInfo](#) class includes a [CreateCustomTimeZone](#) method, which you can use to create custom time zones that are not found in the registry. For details on creating a custom time zone, see [How to: Create time zones without adjustment rules](#) and [How to: Create time zones with adjustment rules](#). In addition, you can use the [ToSerializedString](#) method to convert a newly created time zone to a string and save it in a data store (such as a database, a text file, the registry, or an application resource). You can then use the [FromSerializedString](#) method to convert this string back to a [TimeZoneInfo](#) object. For details, see [How to: Save time zones to an embedded resource](#) and [How to: Restore time zones from an embedded resource](#).

Because each time zone is characterized by a base offset from UTC, as well as by an offset from UTC that reflects any existing adjustment rules, a time in one time zone can be easily converted to the time in another time zone. For this purpose, the [TimeZoneInfo](#) object includes several conversion methods, including:

- [ConvertTimeFromUtc](#), which converts UTC to the time in a designated time zone.

- [ConvertTimeToUtc](#), which converts the time in a designated time zone to UTC.
- [ConvertTime](#), which converts the time in one designated time zone to the time in another designated time zone.
- [ConvertTimeBySystemTimeZoneId](#), which uses time zone identifiers (instead of [TimeZoneInfo](#) objects) as parameters to convert the time in one designated time zone to the time in another designated time zone.

For details on converting times between time zones, see [Converting times between time zones](#).

## See also

- [Dates, times, and time zones](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Use time zones in date and time arithmetic

Article • 09/15/2021

Ordinarily, when you perform date and time arithmetic using [DateTime](#) or [DateTimeOffset](#) values, the result does not reflect any time zone adjustment rules. This is true even when the time zone of the date and time value is clearly identifiable (for example, when the [Kind](#) property is set to [Local](#)). This topic shows how to perform arithmetic operations on date and time values that belong to a particular time zone. The results of the arithmetic operations will reflect the time zone's adjustment rules.

## To apply adjustment rules to date and time arithmetic

1. Implement some method of closely coupling a date and time value with the time zone to which it belongs. For example, declare a structure that includes both the date and time value and its time zone. The following example uses this approach to link a [DateTime](#) value with its time zone.

C#

```
// Define a structure for DateTime values for internal use only
internal struct TimeWithTimeZone
{
    TimeZoneInfo TimeZone;
    DateTime Time;
}
```

2. Convert a time to Coordinated Universal Time (UTC) by calling either the [ConvertTimeToUtc](#) method or the [ConvertTime](#) method.
3. Perform the arithmetic operation on the UTC time.
4. Convert the time from UTC to the original time's associated time zone by calling the [TimeZoneInfo.ConvertTime\(DateTime, TimeZoneInfo\)](#) method.

## Example

The following example adds two hours and thirty minutes to March 9, 2008, at 1:30 A.M. Central Standard Time. The time zone's transition to daylight saving time occurs thirty minutes later, at 2:00 A.M. on March 9, 2008. Because the example follows the four steps

listed in the previous section, it correctly reports the resulting time as 5:00 A.M. on March 9, 2008.

C#

```
using System;

public struct TimeZoneTime
{
    public TimeZoneInfo TimeZone;
    public DateTime Time;

    public TimeZoneTime(TimeZoneInfo tz, DateTime time)
    {
        if (tz == null)
            throw new ArgumentNullException("The time zone cannot be a null
reference.");

        this.TimeZone = tz;
        this.Time = time;
    }

    public TimeZoneTime AddTime(TimeSpan interval)
    {
        // Convert time to UTC
        DateTime utcTime = TimeZoneInfo.ConvertTimeToUtc(this.Time,
this.TimeZone);
        // Add time interval to time
        utcTime = utcTime.Add(interval);
        // Convert time back to time in time zone
        return new TimeZoneTime(this.TimeZone,
TimeZoneInfo.ConvertTime(utcTime,
TimeZoneInfo.Utc, this.TimeZone));
    }
}

public class TimeArithmetic
{
    public const string tzName = "Central Standard Time";

    public static void Main()
    {
        try
        {
            TimeZoneTime cstTime1, cstTime2;

            TimeZoneInfo cst = TimeZoneInfo.FindSystemTimeZoneById(tzName);
            DateTime time1 = new DateTime(2008, 3, 9, 1, 30, 0);
            TimeSpan twoAndAHalfHours = new TimeSpan(2, 30, 0);

            cstTime1 = new TimeZoneTime(cst, time1);
            cstTime2 = cstTime1.AddTime(twoAndAHalfHours);
            Console.WriteLine("{0} + {1} hours = {2}", cstTime1.Time,
```

```
twoAndAHalfHours.ToString(),
cstTime2.Time);
}
catch
{
    Console.WriteLine("Unable to find {0}.".tzName);
}
}
```

Both `DateTime` and `DateTimeOffset` values are disassociated from any time zone to which they might belong. To perform date and time arithmetic in a way that automatically applies a time zone's adjustment rules, the time zone to which any date and time value belongs must be immediately identifiable. This means that a date and time and its associated time zone must be tightly coupled. There are several ways to do this, which include the following:

- Assume that all times used in an application belong to a particular time zone. Although appropriate in some cases, this approach offers limited flexibility and possibly limited portability.
  - Define a type that tightly couples a date and time with its associated time zone by including both as fields of the type. This approach is used in the code example, which defines a structure to store the date and time and the time zone in two member fields.

The example illustrates how to perform arithmetic operations on `DateTime` values so that time zone adjustment rules are applied to the result. However, `DateTimeOffset` values can be used just as easily. The following example illustrates how the code in the original example might be adapted to use `DateTimeOffset` instead of `DateTime` values.

C#

```
using System;

public struct TimeZoneTime
{
    public TimeZoneInfo TimeZone;
    public DateTimeOffset Time;

    public TimeZoneTime(TimeZoneInfo tz, DateTimeOffset time)
    {
        if (tz == null)
            throw new ArgumentNullException("The time zone cannot be a null
reference.");
        this.TimeZone = tz;
        this.Time = time;
    }
}
```

```

}

public TimeZoneTime AddTime(TimeSpan interval)
{
    // Convert time to UTC
    DateTimeOffset utcTime = TimeZoneInfo.ConvertTime(this.Time,
TimeZoneInfo.Utc);
    // Add time interval to time
    utcTime = utcTime.Add(interval);
    // Convert time back to time in time zone
    return new TimeZoneTime(this.TimeZone,
TimeZoneInfo.ConvertTime(utcTime, this.TimeZone));
}

public class TimeArithmetic
{
    public const string tzName = "Central Standard Time";

    public static void Main()
    {
        try
        {
            TimeZoneTime cstTime1, cstTime2;

            TimeZoneInfo cst = TimeZoneInfo.FindSystemTimeZoneById(tzName);
            DateTime time1 = new DateTime(2008, 3, 9, 1, 30, 0);
            TimeSpan twoAndAHalfHours = new TimeSpan(2, 30, 0);

            cstTime1 = new TimeZoneTime(cst,
                new DateTimeOffset(time1, cst.GetUtcOffset(time1)));
            cstTime2 = cstTime1.AddTime(twoAndAHalfHours);
            Console.WriteLine("{0} + {1} hours = {2}", cstTime1.Time,
twoAndAHalfHours.ToString(),
cstTime2.Time);
        }
        catch
        {
            Console.WriteLine("Unable to find {0}.", tzName);
        }
    }
}

```

Note that if this addition is simply performed on the `DateTimeOffset` value without first converting it to UTC, the result reflects the correct point in time but its offset does not reflect that of the designated time zone for that time.

## Compiling the code

This example requires:

- That the `System` namespace be imported with the `using` statement (required in C# code).

## See also

- [Dates, times, and time zones](#)
- [Performing arithmetic operations with dates and times](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Converting between DateTime and DateTimeOffset

Article • 10/04/2022

Although the [DateTimeOffset](#) structure provides a greater degree of time zone awareness than the [DateTime](#) structure, [DateTime](#) parameters are used more commonly in method calls. Because of this approach, the ability to convert [DateTimeOffset](#) values to [DateTime](#) values and vice versa is important. This article shows how to perform these conversions in a way that preserves as much time zone information as possible.

## ⓘ Note

Both the [DateTime](#) and the [DateTimeOffset](#) types have some limitations when representing times in time zones. With its [Kind](#) property, [DateTime](#) is able to reflect only Coordinated Universal Time (UTC) and the system's local time zone. [DateTimeOffset](#) reflects a time's offset from UTC, but it doesn't reflect the actual time zone to which that offset belongs. For more information about time values and support for time zones, see [Choosing Between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo](#).

## Conversions from DateTime to DateTimeOffset

The [DateTimeOffset](#) structure provides two equivalent ways to perform [DateTime](#) to [DateTimeOffset](#) conversion that are suitable for most conversions:

- The [DateTimeOffset](#) constructor, which creates a new [DateTimeOffset](#) object based on a [DateTime](#) value.
- The implicit conversion operator, which allows you to assign a [DateTime](#) value to a [DateTimeOffset](#) object.

For UTC and local [DateTime](#) values, the [Offset](#) property of the resulting [DateTimeOffset](#) value accurately reflects the UTC or local time zone offset. For example, the following code converts a UTC time to its equivalent [DateTimeOffset](#) value:

C#

```
DateTime utcTime1 = new DateTime(2008, 6, 19, 7, 0, 0);
utcTime1 = DateTime.SpecifyKind(utcTime1, DateTimeKind.Utc);
DateTimeOffset utcTime2 = utcTime1;
```

```
Console.WriteLine("Converted {0} {1} to a DateTimeOffset value of {2}",
    utcTime1,
    utcTime1.Kind,
    utcTime2);
// This example displays the following output to the console:
//     Converted 6/19/2008 7:00:00 AM Utc to a DateTimeOffset value of
6/19/2008 7:00:00 AM +00:00
```

In this case, the offset of the `utcTime2` variable is 00:00. Similarly, the following code converts a local time to its equivalent [DateTimeOffset](#) value:

```
C#
DateTime localTime1 = new DateTime(2008, 6, 19, 7, 0, 0);
localTime1 = DateTime.SpecifyKind(localTime1, DateTimeKind.Local);
DateTimeOffset localTime2 = localTime1;
Console.WriteLine("Converted {0} {1} to a DateTimeOffset value of {2}",
    localTime1,
    localTime1.Kind,
    localTime2);
// This example displays the following output to the console:
//     Converted 6/19/2008 7:00:00 AM Local to a DateTimeOffset value of
6/19/2008 7:00:00 AM -07:00
```

However, for [DateTime](#) values whose [Kind](#) property is [DateTimeKind.Unspecified](#), these two conversion methods produce a [DateTimeOffset](#) value whose offset is that of the local time zone. The conversion is shown in the following example, which is run in the U.S. Pacific Standard Time zone:

```
C#
DateTime time1 = new DateTime(2008, 6, 19, 7, 0, 0); // Kind is
DateTimeKind.Unspecified
DateTimeOffset time2 = time1;
Console.WriteLine("Converted {0} {1} to a DateTimeOffset value of {2}",
    time1,
    time1.Kind,
    time2);
// This example displays the following output to the console:
//     Converted 6/19/2008 7:00:00 AM Unspecified to a DateTimeOffset value
of 6/19/2008 7:00:00 AM -07:00
```

If the [DateTime](#) value reflects the date and time in something other than the local time zone or UTC, you can convert it to a [DateTimeOffset](#) value and preserve its time zone information by calling the overloaded [DateTimeOffset](#) constructor. For example, the following example instantiates a [DateTimeOffset](#) object that reflects Central Standard Time:

C#

```
DateTime time1 = new DateTime(2008, 6, 19, 7, 0, 0);      // Kind is
DateTimeKind.Unspecified
try
{
    DateTimeOffset time2 = new DateTimeOffset(time1,
                                                TimeZoneInfo.FindSystemTimeZoneById("Central Standard
Time").GetUtcOffset(time1));
    Console.WriteLine("Converted {0} {1} to a DateTime value of {2}",
                      time1,
                      time1.Kind,
                      time2);
}
// Handle exception if time zone is not defined in registry
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("Unable to identify target time zone for conversion.");
}
// This example displays the following output to the console:
//   Converted 6/19/2008 7:00:00 AM Unspecified to a DateTime value of
6/19/2008 7:00:00 AM -05:00
```

The second parameter to this constructor overload is a [TimeSpan](#) object that represents the time's offset from UTC. Retrieve it by calling the [TimeZoneInfo.GetUtcOffset\(DateTime\)](#) method of the time's corresponding time zone. The method's single parameter is the [DateTime](#) value that represents the date and time to be converted. If the time zone supports daylight saving time, this parameter allows the method to determine the appropriate offset for that particular date and time.

## Conversions from DateTimeOffset to DateTime

The [DateTime](#) property is most commonly used to perform [DateTimeOffset](#) to [DateTime](#) conversion. However, it returns a [DateTime](#) value whose [Kind](#) property is [Unspecified](#), as the following example illustrates:

C#

```
DateTime baseTime = new DateTime(2008, 6, 19, 7, 0, 0);
DateTimeOffset sourceTime;
DateTime targetTime;

// Convert UTC to DateTime value
sourceTime = new DateTimeOffset(baseTime, TimeSpan.Zero);
targetTime = sourceTime.DateTime;
Console.WriteLine("{0} converts to {1} {2}",
                  sourceTime,
                  targetTime,
```

```

        targetTime.Kind);

// Convert local time to DateTime value
sourceTime = new DateTimeOffset(baseTime,
                                TimeZoneInfo.Local.GetUtcOffset(baseTime));
targetTime = sourceTime.DateTime;
Console.WriteLine("{0} converts to {1} {2}",
                 sourceTime,
                 targetTime,
                 targetTime.Kind);

// Convert Central Standard Time to a DateTime value
try
{
    TimeSpan offset = TimeZoneInfo.FindSystemTimeZoneById("Central Standard
Time").GetUtcOffset(baseTime);
    sourceTime = new DateTimeOffset(baseTime, offset);
    targetTime = sourceTime.DateTime;
    Console.WriteLine("{0} converts to {1} {2}",
                     sourceTime,
                     targetTime,
                     targetTime.Kind);
}
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("Unable to create DateTimeOffset based on U.S. Central
Standard Time.");
}
// This example displays the following output to the console:
//   6/19/2008 7:00:00 AM +00:00 converts to 6/19/2008 7:00:00 AM
Unspecified
//   6/19/2008 7:00:00 AM -07:00 converts to 6/19/2008 7:00:00 AM
Unspecified
//   6/19/2008 7:00:00 AM -05:00 converts to 6/19/2008 7:00:00 AM
Unspecified

```

The preceding example shows that any information about the [DateTimeOffset](#) value's relationship to UTC is lost by the conversion when the [DateTime](#) property is used. This behavior also affects [DateTimeOffset](#) values that correspond to UTC time or the system's local time because the [DateTime](#) structure reflects only those two time zones in its [Kind](#) property.

To preserve as much time zone information as possible when converting a [DateTimeOffset](#) to a [DateTime](#) value, you can use the [DateTimeOffset.UtcDateTime](#) and [DateTimeOffset.LocalDateTime](#) properties.

## Converting a UTC time

To indicate that a converted [DateTime](#) value is the UTC time, you can retrieve the value of the [DateTimeOffset.UtcDateTime](#) property. It differs from the [DateTime](#) property in two ways:

- It returns a [DateTime](#) value whose [Kind](#) property is [Utc](#).
- If the [Offset](#) property value doesn't equal [TimeSpan.Zero](#), it converts the time to UTC.

#### ⓘ Note

If your application requires that converted [DateTime](#) values unambiguously identify a single point in time, you should consider using the [DateTimeOffset.UtcDateTime](#) property to handle all [DateTimeOffset](#) to [DateTime](#) conversions.

The following code uses the [UtcDateTime](#) property to convert a [DateTimeOffset](#) value whose offset equals [TimeSpan.Zero](#) to a [DateTime](#) value:

C#

```
DateTimeOffset utcTime1 = new DateTimeOffset(2008, 6, 19, 7, 0, 0,
TimeSpan.Zero);
DateTime utcTime2 = utcTime1.UtcDateTime;
Console.WriteLine("{0} converted to {1} {2}",
    utcTime1,
    utcTime2,
    utcTime2.Kind);
// The example displays the following output to the console:
// 6/19/2008 7:00:00 AM +00:00 converted to 6/19/2008 7:00:00 AM Utc
```

The following code uses the [UtcDateTime](#) property to perform both a time zone conversion and a type conversion on a [DateTimeOffset](#) value:

C#

```
DateTimeOffset originalTime = new DateTimeOffset(2008, 6, 19, 7, 0, 0, new
TimeSpan(5, 0, 0));
DateTime utcTime = originalTime.UtcDateTime;
Console.WriteLine("{0} converted to {1} {2}",
    originalTime,
    utcTime,
    utcTime.Kind);
// The example displays the following output to the console:
// 6/19/2008 7:00:00 AM +05:00 converted to 6/19/2008 2:00:00 AM Utc
```

## Converting a local time

To indicate that a [DateTimeOffset](#) value represents the local time, you can pass the [DateTime](#) value returned by the [DateTimeOffset.DateTime](#) property to the `static` (`Shared` in Visual Basic) [SpecifyKind](#) method. The method returns the date and time passed to it as its first parameter but sets the [Kind](#) property to the value specified by its second parameter. The following code uses the [SpecifyKind](#) method when converting a [DateTimeOffset](#) value whose offset corresponds to that of the local time zone:

C#

```
DateTime sourceDate = new DateTime(2008, 6, 19, 7, 0, 0);
DateTimeOffset utcTime1 = new DateTimeOffset(sourceDate,
                                             TimeZoneInfo.Local.GetUtcOffset(sourceDate));
DateTime utcTime2 = utcTime1.DateTime;
if
(utcTime1.Offset.Equals(TimeZoneInfo.Local.GetUtcOffset(utcTime1.DateTime)))
    utcTime2 = DateTime.SpecifyKind(utcTime2, DateTimeKind.Local);

Console.WriteLine("{0} converted to {1} {2}",
                  utcTime1,
                  utcTime2,
                  utcTime2.Kind);
// The example displays the following output to the console:
// 6/19/2008 7:00:00 AM -07:00 converted to 6/19/2008 7:00:00 AM Local
```

You can also use the [DateTimeOffset.LocalDateTime](#) property to convert a [DateTimeOffset](#) value to a local [DateTime](#) value. The [Kind](#) property of the returned [DateTime](#) value is [Local](#). The following code uses the [DateTimeOffset.LocalDateTime](#) property when converting a [DateTimeOffset](#) value whose offset corresponds to that of the local time zone:

C#

```
DateTime sourceDate = new DateTime(2008, 6, 19, 7, 0, 0);
DateTimeOffset localTime1 = new DateTimeOffset(sourceDate,
                                               TimeZoneInfo.Local.GetUtcOffset(sourceDate));
DateTime localTime2 = localTime1.LocalDateTime;

Console.WriteLine("{0} converted to {1} {2}",
                  localTime1,
                  localTime2,
                  localTime2.Kind);
// The example displays the following output to the console:
// 6/19/2008 7:00:00 AM -07:00 converted to 6/19/2008 7:00:00 AM Local
```

When you retrieve a `DateTime` value using the `DateTimeOffset.LocalDateTime` property, the property's `get` accessor first converts the `DateTimeOffset` value to UTC, then converts it to local time by calling the `ToLocalTime` method. This behavior means that you can retrieve a value from the `DateTimeOffset.LocalDateTime` property to perform a time zone conversion at the same time that you perform a type conversion. It also means that the local time zone's adjustment rules are applied in performing the conversion. The following code illustrates the use of the `DateTimeOffset.LocalDateTime` property to perform both a type and a time zone conversion. The sample output is for a machine set to the Pacific Time Zone (US and Canada). The November date is Pacific Standard Time, which is UTC-8, while the June date is Daylight Savings Time, which is UTC-7.

C#

```
DateTimeOffset originalDate;
DateTime localDate;

// Convert time originating in a different time zone
originalDate = new DateTimeOffset(2008, 6, 18, 7, 0, 0,
                                  new TimeSpan(-5, 0, 0));
localDate = originalDate.LocalDateTime;
Console.WriteLine("{0} converted to {1} {2}",
                  originalDate,
                  localDate,
                  localDate.Kind);

// Convert time originating in a different time zone
// so local time zone's adjustment rules are applied
originalDate = new DateTimeOffset(2007, 11, 4, 4, 0, 0,
                                  new TimeSpan(-5, 0, 0));
localDate = originalDate.LocalDateTime;
Console.WriteLine("{0} converted to {1} {2}",
                  originalDate,
                  localDate,
                  localDate.Kind);

// The example displays the following output to the console,
// when you run it on a machine that is set to Pacific Time (US & Canada):
//       6/18/2008 7:00:00 AM -05:00 converted to 6/18/2008 5:00:00 AM Local
//       11/4/2007 4:00:00 AM -05:00 converted to 11/4/2007 1:00:00 AM Local
```

## A general-purpose conversion method

The following example defines a method named `ConvertFromDateTimeOffset` that converts `DateTimeOffset` values to `DateTime` values. Based on its offset, it determines whether the `DateTimeOffset` value is a UTC time, a local time, or some other time and defines the returned date and time value's `Kind` property accordingly.

C#

```
static DateTime ConvertFromDateTimeOffset(DateTimeOffset dateTime)
{
    if (dateTime.Offset.Equals(TimeSpan.Zero))
        return dateTime.UtcDateTime;
    else if
(dateTime.Offset.Equals(TimeZoneInfo.Local.GetUtcOffset(dateTime.DateTime)))
        return DateTime.SpecifyKind(dateTime.DateTime, DateTimeKind.Local);
    else
        return dateTime.DateTime;
}
```

The following example calls the `ConvertFromDateTimeOffset` method to convert `DateTimeOffset` values that represent a UTC time, a local time, and a time in the U.S. Central Standard Time zone.

C#

```
DateTime timeComponent = new DateTime(2008, 6, 19, 7, 0, 0);
DateTime returnedDate;

// Convert UTC time
DateTimeOffset utcTime = new DateTimeOffset(timeComponent, TimeSpan.Zero);
returnedDate = ConvertFromDateTimeOffset(utcTime);
Console.WriteLine("{0} converted to {1} {2}",
    utcTime,
    returnedDate,
    returnedDate.Kind);

// Convert local time
DateTimeOffset localTime = new DateTimeOffset(timeComponent,
    TimeZoneInfo.Local.GetUtcOffset(timeComponent));
returnedDate = ConvertFromDateTimeOffset(localTime);
Console.WriteLine("{0} converted to {1} {2}",
    localTime,
    returnedDate,
    returnedDate.Kind);

// Convert Central Standard Time
DateTimeOffset cstTime = new DateTimeOffset(timeComponent,
    TimeZoneInfo.FindSystemTimeZoneById("Central Standard
Time").GetUtcOffset(timeComponent));
returnedDate = ConvertFromDateTimeOffset(cstTime);
Console.WriteLine("{0} converted to {1} {2}",
    cstTime,
    returnedDate,
    returnedDate.Kind);

// The example displays the following output to the console:
// 6/19/2008 7:00:00 AM +00:00 converted to 6/19/2008 7:00:00 AM Utc
// 6/19/2008 7:00:00 AM -07:00 converted to 6/19/2008 7:00:00 AM Local
```

```
// 6/19/2008 7:00:00 AM -05:00 converted to 6/19/2008 7:00:00 AM
Unspecified
```

## ⓘ Note

The code makes the following two assumptions, depending on the application and the source of its date and time values, might not always be valid:

- It assumes that a date and time value whose offset is `TimeSpan.Zero` represents UTC. In fact, UTC isn't a time in a particular time zone, but the time in relation to which the times in the world's time zones are standardized. Time zones can also have an offset of `Zero`.
- It assumes that a date and time whose offset equals that of the local time zone represents the local time zone. Because date and time values are disassociated from their original time zone, this might not be the case; the date and time can have originated in another time zone with the same offset.

## See also

- [Dates, times, and time zones](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Converting times between time zones

Article • 09/08/2022

It's becoming increasingly important for any application that works with dates and times to handle differences between time zones. An application can no longer assume that all times can be expressed in the local time, which is the time available from the [DateTime](#) structure. For example, a web page that displays the current time in the eastern part of the United States will lack credibility to a customer in eastern Asia. This article explains how to convert times from one time zone to another and convert [DateTimeOffset](#) values that have limited time zone awareness.

## Converting to Coordinated Universal Time

Coordinated Universal Time (UTC) is a high-precision, atomic time standard. The world's time zones are expressed as positive or negative offsets from UTC. Thus, UTC provides a time-zone free or time-zone neutral time. The use of UTC is recommended when a date and time's portability across computers is important. For details and other best practices using dates and times, see [Coding best practices using DateTime in the .NET Framework](#). Converting individual time zones to UTC makes time comparisons easy.

### ⓘ Note

You can also serialize a [DateTimeOffset](#) structure to represent a single point in time unambiguously. Because [DateTimeOffset](#) objects store a date and time value along with its offset from UTC, they always represent a particular point in time in relation to UTC.

The easiest way to convert a time to UTC is to call the `static (Shared in Visual Basic)` [TimeZoneInfo.ConvertTimeToUtc\(DateTime\)](#) method. The exact conversion performed by the method depends on the value of the `dateTime` parameter's [Kind](#) property, as the following table shows:

<code>DateTime.Kind</code>	<b>Conversion</b>
<code>DateTimeKind.Local</code>	Converts local time to UTC.
<code>DateTimeKind.Unspecified</code>	Assumes the <code>dateTime</code> parameter is local time and converts local time to UTC.
<code>DateTimeKind.Utc</code>	Returns the <code>dateTime</code> parameter unchanged.

The following code converts the current local time to UTC and displays the result to the console:

```
C#
```

```
DateTime dateNow = DateTime.Now;
Console.WriteLine("The date and time are {0} UTC.",
    TimeZoneInfo.ConvertTimeToUtc(dateNow));
```

If the date and time value doesn't represent the local time or UTC, the [ToUniversalTime](#) method will likely return an erroneous result. However, you can use the [TimeZoneInfo.ConvertTimeToUtc](#) method to convert the date and time from a specified time zone. For details on retrieving a [TimeZoneInfo](#) object that represents the destination time zone, see [Finding the time zones defined on a local system](#). The following code uses the [TimeZoneInfo.ConvertTimeToUtc](#) method to convert Eastern Standard Time to UTC:

```
C#
```

```
DateTime easternTime = new DateTime(2007, 01, 02, 12, 16, 00);
string easternZoneId = "Eastern Standard Time";
try
{
    TimeZoneInfo easternZone =
    TimeZoneInfo.FindSystemTimeZoneById(easternZoneId);
    Console.WriteLine("The date and time are {0} UTC.",
        TimeZoneInfo.ConvertTimeToUtc(easternTime,
easternZone));
}
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("Unable to find the {0} zone in the registry.",
        easternZoneId);
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine("Registry data on the {0} zone has been corrupted.",
        easternZoneId);
}
```

The [TimeZoneInfo.ConvertTimeToUtc](#) method throws an [ArgumentException](#) if the [DateTime](#) object's [Kind](#) property and the time zone are mismatched. A mismatch occurs if the [Kind](#) property is [DateTimeKind.Local](#) but the [TimeZoneInfo](#) object doesn't represent the local time zone, or if the [Kind](#) property is [DateTimeKind.Utc](#) but the [TimeZoneInfo](#) object doesn't equal [TimeZoneInfo.Utc](#).

All of these methods take [DateTime](#) values as parameters and return a [DateTime](#) value. For [DateTimeOffset](#) values, the [DateTimeOffset](#) structure has a [ToUniversalTime](#) instance method that converts the date and time of the current instance to UTC. The following example calls the [ToUniversalTime](#) method to convert a local time and several other times to UTC:

C#

```
DateTimeOffset localTime, otherTime, universalTime;

// Define local time in local time zone
localTime = new DateTimeOffset(new DateTime(2007, 6, 15, 12, 0, 0));
Console.WriteLine("Local time: {0}", localTime);
Console.WriteLine();

// Convert local time to offset 0 and assign to otherTime
otherTime = localTime.ToOffset(TimeSpan.Zero);
Console.WriteLine("Other time: {0}", otherTime);
Console.WriteLine("{0} = {1}: {2}",
    localTime, otherTime,
    localTime.Equals(otherTime));
Console.WriteLine("{0} exactly equals {1}: {2}",
    localTime, otherTime,
    localTime.EqualsExact(otherTime));
Console.WriteLine();

// Convert other time to UTC
universalTime = localTime.ToUniversalTime();
Console.WriteLine("Universal time: {0}", universalTime);
Console.WriteLine("{0} = {1}: {2}",
    otherTime, universalTime,
    universalTime.Equals(otherTime));
Console.WriteLine("{0} exactly equals {1}: {2}",
    otherTime, universalTime,
    universalTime.EqualsExact(otherTime));
Console.WriteLine();

// The example produces the following output to the console:
// Local time: 6/15/2007 12:00:00 PM -07:00
//
// Other time: 6/15/2007 7:00:00 PM +00:00
// 6/15/2007 12:00:00 PM -07:00 = 6/15/2007 7:00:00 PM +00:00: True
// 6/15/2007 12:00:00 PM -07:00 exactly equals 6/15/2007 7:00:00 PM
// +00:00: False
//
// Universal time: 6/15/2007 7:00:00 PM +00:00
// 6/15/2007 7:00:00 PM +00:00 = 6/15/2007 7:00:00 PM +00:00: True
// 6/15/2007 7:00:00 PM +00:00 exactly equals 6/15/2007 7:00:00 PM
// +00:00: True
```

## Converting UTC to a designated time zone

To convert UTC to local time, see the [Converting UTC to local time](#) section that follows.

To convert UTC to the time in any time zone that you designate, call the

[ConvertTimeFromUtc](#) method. The method takes two parameters:

- The UTC to convert. This must be a [DateTime](#) value whose [Kind](#) property is set to [Unspecified](#) or [Utc](#).
- The time zone to convert the UTC to.

The following code converts UTC to Central Standard Time:

C#

```
DateTime timeUtc = DateTime.UtcNow;
try
{
    TimeZoneInfo cstZone = TimeZoneInfo.FindSystemTimeZoneById("Central
Standard Time");
    DateTime cstTime = TimeZoneInfo.ConvertTimeFromUtc(timeUtc, cstZone);
    Console.WriteLine("The date and time are {0} {1}.",
                      cstTime,
                      cstZone.IsDaylightSavingTime(cstTime) ?
                      cstZone.DaylightName : cstZone.StandardName);
}
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("The registry does not define the Central Standard Time
zone.");
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine("Registry data on the Central Standard Time zone has
been corrupted.");
}
```

## Converting UTC to local time

To convert UTC to local time, call the [ToLocalTime](#) method of the [DateTime](#) object whose time you want to convert. The exact behavior of the method depends on the value of the object's [Kind](#) property, as the following table shows:

<a href="#">DateTime.Kind</a>	<b>Conversion</b>
<a href="#">DateTimeKind.Local</a>	Returns the <a href="#">DateTime</a> value unchanged.
<a href="#">DateTimeKind.Unspecified</a>	Assumes that the <a href="#">DateTime</a> value is UTC and converts the UTC to local time.

<code>DateTime.Kind</code>	<b>Conversion</b>
<code>DateTimeKind.Utc</code>	Converts the <code>DateTime</code> value to local time.

### ① Note

The `TimeZone.ToLocalTime` method behaves identically to the `DateTime.ToLocalTime` method. It takes a single parameter, which is the date and time value, to convert.

You can also convert the time in any designated time zone to local time by using the `static` (Shared in Visual Basic) `TimeZoneInfo.ConvertTime` method. This technique is discussed in the next section.

## Converting between any two time zones

You can convert between any two time zones by using either of the following two `static` (Shared in Visual Basic) methods of the `TimeZoneInfo` class:

- [ConvertTime](#)

This method's parameters are the date and time value to convert, a `TimeZoneInfo` object that represents the time zone of the date and time value, and a `TimeZoneInfo` object that represents the time zone to convert the date and time value to.

- [ConvertTimeBySystemTimeZonId](#)

This method's parameters are the date and time value to convert, the identifier of the date and time value's time zone, and the identifier of the time zone to convert the date and time value to.

Both methods require that the `Kind` property of the date and time value to convert and the `TimeZoneInfo` object or time zone identifier that represents its time zone correspond to one another. Otherwise, an `ArgumentException` is thrown. For example, if the `Kind` property of the date and time value is `DateTimeKind.Local`, an exception is thrown if the `TimeZoneInfo` object passed as a parameter to the method isn't equal to `TimeZoneInfo.Local`. An exception is also thrown if the identifier passed as a parameter to the method isn't equal to `TimeZoneInfo.Local.Id`.

The following example uses the [ConvertTime](#) method to convert from Hawaiian Standard Time to local time:

```
C#  
  
DateTime hwTime = new DateTime(2007, 02, 01, 08, 00, 00);  
try  
{  
    TimeZoneInfo hwZone = TimeZoneInfo.FindSystemTimeZoneById("Hawaiian  
Standard Time");  
    Console.WriteLine("{0} {1} is {2} local time.",  
        hwTime,  
        hwZone.IsDaylightSavingTime(hwTime) ? hwZone.DaylightName :  
        hwZone.StandardName,  
        TimeZoneInfo.ConvertTime(hwTime, hwZone, TimeZoneInfo.Local));  
}  
catch (TimeZoneNotFoundException)  
{  
    Console.WriteLine("The registry does not define the Hawaiian Standard  
Time zone.");  
}  
catch (InvalidTimeZoneException)  
{  
    Console.WriteLine("Registry data on the Hawaiian Standard Time zone has  
been corrupted.");  
}
```

## Converting `DateTimeOffset` values

Date and time values represented by [DateTimeOffset](#) objects aren't fully time-zone aware because the object is disassociated from its time zone at the time it's instantiated. However, in many cases, an application simply needs to convert a date and time based on two different offsets from UTC rather than on time in particular time zones. To perform this conversion, you can call the current instance's [ToOffset](#) method. The method's single parameter is the offset of the new date and time value the method will return.

For example, if the date and time of a user request for a web page is known and is serialized as a string in the format MM/dd/yyyy hh:mm:ss zzzz, the following `ReturnTimeOnServer` method converts this date and time value to the date and time on the web server:

```
C#  
  
public DateTimeOffset ReturnTimeOnServer(string clientString)  
{  
    string format = @"M/d/yyyy H:m:s zzz";
```

```

    TimeSpan serverOffset =
TimeZoneInfo.Local.GetUtcOffset(DateTimeOffset.Now);

    try
    {
        DateTimeOffset clientTime = DateTimeOffset.ParseExact(clientString,
format, CultureInfo.InvariantCulture);
        DateTimeOffset serverTime = clientTime.ToOffset(serverOffset);
        return serverTime;
    }
    catch (FormatException)
    {
        return DateTimeOffset.MinValue;
    }
}

```

If the method passes the string "9/1/2007 5:32:07 -05:00," which represents the date and time in a time zone five hours earlier than UTC, it returns "9/1/2007 3:32:07 AM -07:00" for a server located in the U.S. Pacific Standard Time zone.

The [TimeZoneInfo](#) class also includes an overload of the [TimeZoneInfo.ConvertTime\(DateTimeOffset, TimeZoneInfo\)](#) method that performs time zone conversions with [ToOffset\(TimeSpan\)](#) values. The method's parameters are a [DateTimeOffset](#) value and a reference to the time zone to which the time is to be converted. The method call returns a [DateTimeOffset](#) value. For example, the [ReturnTimeOnServer](#) method in the previous example could be rewritten as follows to call the [ConvertTime\(DateTimeOffset, TimeZoneInfo\)](#) method.

C#

```

public DateTimeOffset ReturnTimeOnServer(string clientString)
{
    string format = @"M/d/yyyy H:m:s zzz";

    try
    {
        DateTimeOffset clientTime = DateTimeOffset.ParseExact(clientString,
format,
                           CultureInfo.InvariantCulture);
        DateTimeOffset serverTime = TimeZoneInfo.ConvertTime(clientTime,
TimeZoneInfo.Local);
        return serverTime;
    }
    catch (FormatException)
    {
        return DateTimeOffset.MinValue;
    }
}

```

# See also

- [TimeZoneInfo](#)
- [Dates, times, and time zones](#)
- [Finding the time zones defined on a local system](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Resolve ambiguous times

Article • 09/15/2021

An ambiguous time is a time that maps to more than one Coordinated Universal Time (UTC). It occurs when the clock time is adjusted back in time, such as during the transition from a time zone's daylight saving time to its standard time. When handling an ambiguous time, you can do one of the following:

- Make an assumption about how the time maps to UTC. For example, you can assume that an ambiguous time is always expressed in the time zone's standard time.
- If the ambiguous time is an item of data entered by the user, you can leave it to the user to resolve the ambiguity.

This topic shows how to resolve an ambiguous time by assuming that it represents the time zone's standard time.

## To map an ambiguous time to a time zone's standard time

1. Call the [IsAmbiguousTime](#) method to determine whether the time is ambiguous.
2. If the time is ambiguous, subtract the time from the [TimeSpan](#) object returned by the time zone's [BaseUtcOffset](#) property.
3. Call the `static` (`Shared` in Visual Basic .NET) [SpecifyKind](#) method to set the UTC date and time value's [Kind](#) property to [DateTimeKind.Utc](#).

## Example

The following example illustrates how to convert an ambiguous time to UTC by assuming that it represents the local time zone's standard time.

C#

```
private DateTime ResolveAmbiguousTime(DateTime ambiguousTime)
{
    // Time is not ambiguous
    if (!TimeZoneInfo.Local.IsAmbiguousTime(ambiguousTime))
    {
        return ambiguousTime;
    }
}
```

```

// Time is ambiguous
else
{
    DateTime utcTime = DateTime.SpecifyKind(ambiguousTime -
TimeZoneInfo.Local.BaseUtcOffset,
                                         DateTimeKind.Utc);
    Console.WriteLine("{0} local time corresponds to {1} {2}.",
                      ambiguousTime, utcTime, utcTime.Kind.ToString());
    return utcTime;
}
}

```

The example consists of a method named `ResolveAmbiguousTime` that determines whether the `DateTime` value passed to it is ambiguous. If the value is ambiguous, the method returns a `DateTime` value that represents the corresponding UTC time. The method handles this conversion by subtracting the value of the local time zone's `BaseUtcOffset` property from the local time.

Ordinarily, an ambiguous time is handled by calling the `GetAmbiguousTimeOffsets` method to retrieve an array of `TimeSpan` objects that contain the ambiguous time's possible UTC offsets. However, this example makes the arbitrary assumption that an ambiguous time should always be mapped to the time zone's standard time. The `BaseUtcOffset` property returns the offset between UTC and a time zone's standard time.

In this example, all references to the local time zone are made through the `TimeZoneInfo.Local` property; the local time zone is never assigned to an object variable. This is a recommended practice because a call to the `TimeZoneInfo.ClearCachedData` method invalidates any objects that the local time zone is assigned to.

## Compiling the code

This example requires:

- That the `System` namespace be imported with the `using` statement (required in C# code).

## See also

- [Dates, times, and time zones](#)
- [How to: Let users resolve ambiguous times](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Let users resolve ambiguous times

Article • 09/15/2021

An ambiguous time is a time that maps to more than one Coordinated Universal Time (UTC). It occurs when the clock time is adjusted back in time, such as during the transition from a time zone's daylight saving time to its standard time. When handling an ambiguous time, you can do one of the following:

- If the ambiguous time is an item of data entered by the user, you can leave it to the user to resolve the ambiguity.
- Make an assumption about how the time maps to UTC. For example, you can assume that an ambiguous time is always expressed in the time zone's standard time.

This topic shows how to let a user resolve an ambiguous time.

## To let a user resolve an ambiguous time

1. Get the date and time input by the user.
2. Call the [IsAmbiguousTime](#) method to determine whether the time is ambiguous.
3. If the time is ambiguous, call the [GetAmbiguousTimeOffsets](#) method to retrieve an array of [TimeSpan](#) objects. Each element in the array contains a UTC offset that the ambiguous time can map to.
4. Let the user select the desired offset.
5. Get the UTC date and time by subtracting the offset selected by the user from the local time.
6. Call the `static` (`Shared` in Visual Basic .NET) [SpecifyKind](#) method to set the UTC date and time value's [Kind](#) property to [DateTimeKind.Utc](#).

## Example

The following example prompts the user to enter a date and time and, if it is ambiguous, lets the user select the UTC time that the ambiguous time maps to.

C#

```
private void GetUserDateInput()
{
    // Get date and time from user
    DateTime inputDate = GetUserDateTime();
    DateTime utcDate;

    // Exit if date has no significant value
    if (inputDate == DateTime.MinValue) return;

    if (TimeZoneInfo.Local.IsAmbiguousTime(inputDate))
    {
        Console.WriteLine("The date you've entered is ambiguous.");
        Console.WriteLine("Please select the correct offset from Universal
Coordinated Time:");
        TimeSpan[] offsets =
        TimeZoneInfo.Local.GetAmbiguousTimeOffsets(inputDate);
        for (int ctr = 0; ctr < offsets.Length; ctr++)
        {
            Console.WriteLine("{0}.) {1} hours, {2} minutes", ctr,
offsets[ctr].Hours, offsets[ctr].Minutes);
        }
        Console.Write("> ");
        int selection = Convert.ToInt32(Console.ReadLine());

        // Convert local time to UTC, and set Kind property to
        DateTimeKind.Utc
        utcDate = DateTime.SpecifyKind(inputDate - offsets[selection],
DateTimeKind.Utc);

        Console.WriteLine("{0} local time corresponds to {1} {2}.",
inputDate,
utcDate, utcDate.Kind.ToString());
    }
    else
    {
        utcDate = inputDate.ToUniversalTime();
        Console.WriteLine("{0} local time corresponds to {1} {2}.",
inputDate,
utcDate, utcDate.Kind.ToString());
    }
}

private DateTime GetUserDateTime()
{
    bool exitFlag = false;           // flag to exit loop if date is valid
    string dateString;
    DateTime inputDate = DateTime.MinValue;

    Console.Write("Enter a local date and time: ");
    while (! exitFlag)
    {
        dateString = Console.ReadLine();
        if (dateString.ToUpper() == "E")
            exitFlag = true;

        if (DateTime.TryParse(dateString, out inputDate))

```

```

        exitFlag = true;
    else
        Console.WriteLine("Enter a valid date and time, or enter 'e' to exit:
");
    }

    return inputDate;
}

```

VB

```

Private Sub GetUserDateInput()
    ' Get date and time from user
    Dim inputDate As Date = GetUserDateTime()
    Dim utcDate As Date

    ' Exit if date has no significant value
    If inputDate = Date.MinValue Then Exit Sub

    If TimeZoneInfo.Local.IsAmbiguousTime(inputDate) Then
        Console.WriteLine("The date you've entered is ambiguous.")
        Console.WriteLine("Please select the correct offset from Universal
Coordinated Time:")
        Dim offsets() As TimeSpan =
TimeZoneInfo.Local.GetAmbiguousTimeOffsets(inputDate)
        For ctr As Integer = 0 to offsets.Length - 1
            Dim zoneDescription As String
            If offsets(ctr).Equals(TimeZoneInfo.Local.BaseUtcOffset) Then
                zoneDescription = TimeZoneInfo.Local.StandardName
            Else
                zoneDescription = TimeZoneInfo.Local.DaylightName
            End If
            Console.WriteLine("{0}.) {1} hours, {2} minutes ({3})",
                ctr, offsets(ctr).Hours, offsets(ctr).Minutes,
                zoneDescription)
        Next
        Console.Write("> ")
        Dim selection As Integer = CInt(Console.ReadLine())

        ' Convert local time to UTC, and set Kind property to
        DateTimeKind.Utc
        utcDate = Date.SpecifyKind(inputDate - offsets(selection),
        DateTimeKind.Utc)

        Console.WriteLine("{0} local time corresponds to {1} {2}.",
            inputDate, utcDate, utcDate.Kind.ToString())
    Else
        utcDate = inputDate.ToUniversalTime()
        Console.WriteLine("{0} local time corresponds to {1} {2}.",
            inputDate, utcDate, utcDate.Kind.ToString())
    End If
End Sub

```

```

Private Function GetUserDateTime() As Date
    Dim exitFlag As Boolean = False           ' flag to exit loop if date
    is valid
    Dim dateString As String
    Dim inputDate As Date = Date.MinValue

    Console.WriteLine("Enter a local date and time: ")
    Do While Not exitFlag
        dateString = Console.ReadLine()
        If dateString.ToUpper = "E" Then exitFlag = True
        If Date.TryParse(dateString, inputDate) Then
            exitFlag = true
        Else
            Console.WriteLine("Enter a valid date and time, or enter 'e' to
exit: ")
        End If
    Loop

    Return inputDate
End Function

```

The core of the example code uses an array of [TimeSpan](#) objects to indicate possible offsets of the ambiguous time from UTC. However, these offsets are unlikely to be meaningful to the user. To clarify the meaning of the offsets, the code also notes whether an offset represents the local time zone's standard time or its daylight saving time. The code determines which time is standard and which time is daylight by comparing the offset with the value of the [BaseUtcOffset](#) property. This property indicates the difference between the UTC and the time zone's standard time.

In this example, all references to the local time zone are made through the [TimeZoneInfo.Local](#) property; the local time zone is never assigned to an object variable. This is a recommended practice because a call to the [TimeZoneInfo.ClearCachedData](#) method invalidates any objects that the local time zone is assigned to.

## Compiling the code

This example requires:

- That the [System](#) namespace be imported with the `using` statement (required in C# code).

## See also

- [Dates, times, and time zones](#)
- [How to: Resolve ambiguous times](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Instantiating a DateTimeOffset object

Article • 01/03/2023

The [DateTimeOffset](#) structure offers a number of ways to create new [DateTimeOffset](#) values. Many of them correspond directly to the methods available for instantiating new [DateTime](#) values, with enhancements that allow you to specify the date and time value's offset from Coordinated Universal Time (UTC). In particular, you can instantiate a [DateTimeOffset](#) value in the following ways:

- By using a date and time literal.
- By calling a [DateTimeOffset](#) constructor.
- By implicitly converting a value to [DateTimeOffset](#) value.
- By parsing the string representation of a date and time.

This topic provides greater detail and code examples that illustrate these methods of instantiating new [DateTimeOffset](#) values.

## Date and time literals

For languages that support it, one of the most common ways to instantiate a [DateTime](#) value is to provide the date and time as a hard-coded literal value. For example, the following Visual Basic code creates a [DateTime](#) object whose value is May 1, 2008, at 8:06:32 AM.

VB

```
Dim literalDate1 As Date = #05/01/2008 8:06:32 AM#
Console.WriteLine(literalDate1.ToString())
' Displays:
'      5/1/2008 8:06:32 AM
```

[DateTimeOffset](#) values can also be initialized using date and time literals when using languages that support [DateTime](#) literals. For example, the following Visual Basic code creates a [DateTimeOffset](#) object.

VB

```
Dim literalDate As DateTimeOffset = #05/01/2008 8:06:32 AM#
Console.WriteLine(literalDate.ToString())
```

```
' Displays:  
' 5/1/2008 8:06:32 AM -07:00
```

As the console output shows, the [DateTimeOffset](#) value created in this way is assigned the offset of the local time zone. This means that a [DateTimeOffset](#) value assigned using a character literal does not identify a single point of time if the code is run on different computers.

## DateTimeOffset constructors

The [DateTimeOffset](#) type defines six constructors. Four of them correspond directly to [DateTime](#) constructors, with an additional parameter of type  [TimeSpan](#) that defines the date and time's offset from UTC. These allow you to define a [DateTimeOffset](#) value based on the value of its individual date and time components. For example, the following code uses these four constructors to instantiate [DateTimeOffset](#) objects with identical values of 5/1/2008 8:06:32 +01:00.

C#

```
DateTimeOffset dateAndTime;  
  
// Instantiate date and time using years, months, days,  
// hours, minutes, and seconds  
dateAndTime = new DateTimeOffset(2008, 5, 1, 8, 6, 32,  
                               new TimeSpan(1, 0, 0));  
Console.WriteLine(dateAndTime);  
// Instantiate date and time using years, months, days,  
// hours, minutes, seconds, and milliseconds  
dateAndTime = new DateTimeOffset(2008, 5, 1, 8, 6, 32, 545,  
                               new TimeSpan(1, 0, 0));  
Console.WriteLine("{0} {1}", dateAndTime.ToString("G"),  
                  dateAndTime.ToString("zzz"));  
  
// Instantiate date and time using Persian calendar with years,  
// months, days, hours, minutes, seconds, and milliseconds  
dateAndTime = new DateTimeOffset(1387, 2, 12, 8, 6, 32, 545,  
                               new PersianCalendar(),  
                               new TimeSpan(1, 0, 0));  
// Note that the console output displays the date in the Gregorian  
// calendar, not the Persian calendar.  
Console.WriteLine("{0} {1}", dateAndTime.ToString("G"),  
                  dateAndTime.ToString("zzz"));  
  
// Instantiate date and time using number of ticks  
// 05/01/2008 8:06:32 AM is 633,452,259,920,000,000 ticks  
dateAndTime = new DateTimeOffset(633452259920000000, new TimeSpan(1, 0, 0));  
Console.WriteLine(dateAndTime);  
// The example displays the following output to the console:  
//      5/1/2008 8:06:32 AM +01:00
```

```
//      5/1/2008 8:06:32 AM +01:00
//      5/1/2008 8:06:32 AM +01:00
//      5/1/2008 8:06:32 AM +01:00
```

Note that, when the value of the [DateTimeOffset](#) object instantiated using a [PersianCalendar](#) object as one of the arguments to its constructor is displayed to the console, it is expressed as a date in the Gregorian rather than the Persian calendar. To output a date using the Persian calendar, see the example in the [PersianCalendar](#) topic.

The other two constructors create a [DateTimeOffset](#) object from a [DateTime](#) value. The first of these has a single parameter, the [DateTime](#) value to convert to a [DateTimeOffset](#) value. The offset of the resulting [DateTimeOffset](#) value depends on the [Kind](#) property of the constructor's single parameter. If its value is [DateTimeKind.Utc](#), the offset is set equal to [TimeSpan.Zero](#). Otherwise, its offset is set equal to that of the local time zone. The following example illustrates the use of this constructor to instantiate [DateTimeOffset](#) objects representing UTC and the local time zone:

C#

```
// Declare date; Kind property is DateTimeKind.Unspecified
DateTime sourceDate = new DateTime(2008, 5, 1, 8, 30, 0);
DateTimeOffset targetTime;

// Instantiate a DateTimeOffset value from a UTC time
DateTime utcTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Utc);
targetTime = new DateTimeOffset(utcTime);
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM +00:00
// Because the Kind property is DateTimeKind.Utc,
// the offset is TimeSpan.Zero.

// Instantiate a DateTimeOffset value from a UTC time with a zero offset
targetTime = new DateTimeOffset(utcTime, TimeSpan.Zero);
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM +00:00
// Because the Kind property is DateTimeKind.Utc,
// the call to the constructor succeeds

// Instantiate a DateTimeOffset value from a UTC time with a negative offset
try
{
    targetTime = new DateTimeOffset(utcTime, new TimeSpan(-2, 0, 0));
    Console.WriteLine(targetTime);
}
catch (ArgumentException)
{
    Console.WriteLine("Attempt to create DateTimeOffset value from {0} failed.",
                    targetTime);
}
```

```
// Throws exception and displays the following to the console:  
//   Attempt to create DateTimeOffset value from 5/1/2008 8:30:00 AM +00:00  
// failed.  
  
// Instantiate a DateTimeOffset value from a local time  
DateTime localTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Local);  
targetTime = new DateTimeOffset(localTime);  
Console.WriteLine(targetTime);  
// Displays 5/1/2008 8:30:00 AM -07:00  
// Because the Kind property is DateTimeKind.Local,  
// the offset is that of the local time zone.  
  
// Instantiate a DateTimeOffset value from an unspecified time  
targetTime = new DateTimeOffset(sourceDate);  
Console.WriteLine(targetTime);  
// Displays 5/1/2008 8:30:00 AM -07:00  
// Because the Kind property is DateTimeKind.Unspecified,  
// the offset is that of the local time zone.
```

## ① Note

Calling the overload of the `DateTimeOffset` constructor that has a single `DateTime` parameter is equivalent to performing an implicit conversion of a `DateTime` value to a `DateTimeOffset` value.

The second constructor that creates a `DateTimeOffset` object from a `DateTime` value has two parameters: the `DateTime` value to convert, and a  `TimeSpan` value representing the date and time's offset from UTC. This offset value must correspond to the `Kind` property of the constructor's first parameter or an `ArgumentException` is thrown. If the `Kind` property of the first parameter is `DateTimeKind.Utc`, the value of the second parameter must be  `TimeSpan.Zero`. If the `Kind` property of the first parameter is `DateTimeKind.Local`, the value of the second parameter must be the offset of the local system's time zone. If the `Kind` property of the first parameter is `DateTimeKind.Unspecified`, the offset can be any valid value. The following code illustrates calls to this constructor to convert `DateTime` to `DateTimeOffset` values.

C#

```
DateTime sourceDate = new DateTime(2008, 5, 1, 8, 30, 0);  
DateTimeOffset targetTime;  
  
// Instantiate a DateTimeOffset value from a UTC time with a zero offset.  
DateTime utcTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Utc);  
targetTime = new DateTimeOffset(utcTime, TimeSpan.Zero);  
Console.WriteLine(targetTime);  
// Displays 5/1/2008 8:30:00 AM +00:00  
// Because the Kind property is DateTimeKind.Utc,
```

```

// the call to the constructor succeeds

// Instantiate a DateTimeOffset value from a UTC time with a non-zero
// offset.
try
{
    targetTime = new DateTimeOffset(utcTime, new TimeSpan(-2, 0, 0));
    Console.WriteLine(targetTime);
}
catch (ArgumentException)
{
    Console.WriteLine("Attempt to create DateTimeOffset value from {0} failed.",
                      utcTime);
}
// Throws exception and displays the following to the console:
//   Attempt to create DateTimeOffset value from 5/1/2008 8:30:00 AM failed.

// Instantiate a DateTimeOffset value from a local time with
// the offset of the local time zone
DateTime localTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Local);
targetTime = new DateTimeOffset(localTime,
                                TimeZoneInfo.Local.GetUtcOffset(localTime));
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -07:00
// Because the Kind property is DateTimeKind.Local and the offset matches
// that of the local time zone, the call to the constructor succeeds.

// Instantiate a DateTimeOffset value from a local time with a zero offset.
try
{
    targetTime = new DateTimeOffset(localTime, TimeSpan.Zero);
    Console.WriteLine(targetTime);
}
catch (ArgumentException)
{
    Console.WriteLine("Attempt to create DateTimeOffset value from {0} failed.",
                      localTime);
}
// Throws exception and displays the following to the console:
//   Attempt to create DateTimeOffset value from 5/1/2008 8:30:00 AM failed.

// Instantiate a DateTimeOffset value with an arbitrary time zone.
string timeZoneName = "Central Standard Time";
TimeSpan offset = TimeZoneInfo.FindSystemTimeZoneById(timeZoneName).
    GetUtcOffset(sourceDate);
targetTime = new DateTimeOffset(sourceDate, offset);
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -05:00

```

## Implicit type conversion

The `DateTimeOffset` type supports one *implicit* type conversion: from a `DateTime` value to a `DateTimeOffset` value. (An implicit type conversion is a conversion from one type to another that does not require an explicit cast (in C#) or conversion (in Visual Basic) and that does not lose information.) It makes code like the following possible.

C#

```
DateTimeOffset targetTime;

// The Kind property of sourceDate is DateTimeKind.Unspecified
DateTime sourceDate = new DateTime(2008, 5, 1, 8, 30, 0);
targetTime = sourceDate;
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -07:00

// define a UTC time (Kind property is DateTimeKind.Utc)
DateTime utcTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Utc);
targetTime = utcTime;
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM +00:00

// Define a local time (Kind property is DateTimeKind.Local)
DateTime localTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Local);
targetTime = localTime;
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -07:00
```

The offset of the resulting `DateTimeOffset` value depends on the `DateTime.Kind` property value. If its value is `DateTimeKind.Utc`, the offset is set equal to `TimeSpan.Zero`. If its value is either `DateTimeKind.Local` or `DateTimeKind.Unspecified`, the offset is set equal to that of the local time zone.

## Parsing the string representation of a date and time

The `DateTimeOffset` type supports four methods that allow you to convert the string representation of a date and time into a `DateTimeOffset` value:

- `Parse`, which tries to convert the string representation of a date and time to a `DateTimeOffset` value and throws an exception if the conversion fails.
- `TryParse`, which tries to convert the string representation of a date and time to a `DateTimeOffset` value and returns `false` if the conversion fails.
- `ParseExact`, which tries to convert the string representation of a date and time in a specified format to a `DateTimeOffset` value. The method throws an exception if the

conversion fails.

- [TryParseExact](#), which tries to convert the string representation of a date and time in a specified format to a [DateTimeOffset](#) value. The method returns `false` if the conversion fails.

The following example illustrates calls to each of these four string conversion methods to instantiate a [DateTimeOffset](#) value.

C#

```
string timeString;
DateTimeOffset targetTime;

timeString = "05/01/2008 8:30 AM +01:00";
try
{
    targetTime = DateTimeOffset.Parse(timeString);
    Console.WriteLine(targetTime);
}
catch (FormatException)
{
    Console.WriteLine("Unable to parse {0}.", timeString);
}

timeString = "05/01/2008 8:30 AM";
if (DateTimeOffset.TryParse(timeString, out targetTime))
    Console.WriteLine(targetTime);
else
    Console.WriteLine("Unable to parse {0}.", timeString);

timeString = "Thursday, 01 May 2008 08:30";
try
{
    targetTime = DateTimeOffset.ParseExact(timeString, "f",
        CultureInfo.InvariantCulture);
    Console.WriteLine(targetTime);
}
catch (FormatException)
{
    Console.WriteLine("Unable to parse {0}.", timeString);
}

timeString = "Thursday, 01 May 2008 08:30 +02:00";
string formatString;
formatString = CultureInfo.InvariantCulture.DateTimeFormat.LongDatePattern +
    " " +
    CultureInfo.InvariantCulture.DateTimeFormat.ShortTimePattern +
    " zzz";
if (DateTimeOffset.TryParseExact(timeString,
    formatString,
```

```
        CultureInfo.InvariantCulture,
        DateTimeStyles.AllowLeadingWhite,
        out targetTime))

    Console.WriteLine(targetTime);
else
    Console.WriteLine("Unable to parse {0}.", timeString);
// The example displays the following output to the console:
// 5/1/2008 8:30:00 AM +01:00
// 5/1/2008 8:30:00 AM -07:00
// 5/1/2008 8:30:00 AM -07:00
// 5/1/2008 8:30:00 AM +02:00
```

## See also

- [Dates, times, and time zones](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Create time zones without adjustment rules

Article • 09/15/2021

The precise time zone information that is required by an application may not be present on a particular system for several reasons:

- The time zone has never been defined in the local system's registry.
- Data about the time zone has been modified or removed from the registry.
- The time zone exists but does not have accurate information about time zone adjustments for a particular historic period.

In these cases, you can call the [CreateCustomTimeZone](#) method to define the time zone required by your application. You can use the overloads of this method to create a time zone with or without adjustment rules. If the time zone supports daylight saving time, you can define adjustments with either fixed or floating adjustment rules. (For definitions of these terms, see the "Time Zone Terminology" section in [Time zone overview](#).)

## ⓘ Important

Custom time zones created by calling the [CreateCustomTimeZone](#) method are not added to the registry. Instead, they can be accessed only through the object reference returned by the [CreateCustomTimeZone](#) method call.

This topic shows how to create a time zone without adjustment rules. To create a time zone that supports daylight saving time adjustment rules, see [How to: Create time zones with adjustment rules](#).

## To create a time zone without adjustment rules

1. Define the time zone's display name.

The display name follows a fairly standard format in which the time zone's offset from Coordinated Universal Time (UTC) is enclosed in parentheses and is followed by a string that identifies the time zone, one or more of the cities in the time zone, or one or more of the countries or regions in the time zone.

2. Define the name of the time zone's standard time. Typically, this string is also used as the time zone's identifier.
3. If you want to use a different identifier than the time zone's standard name, define the time zone identifier.
4. Instantiate a [TimeSpan](#) object that defines the time zone's offset from UTC. Time zones with times that are later than UTC have a positive offset. Time zones with times that are earlier than UTC have a negative offset.
5. Call the [TimeZoneInfo.CreateCustomTimeZone\(String, TimeSpan, String, String\)](#) method to instantiate the new time zone.

## Example

The following example defines a custom time zone for Mawson, Antarctica, which has no adjustment rules.

C#

```
string displayName = "(GMT+06:00) Antarctica/Mawson Time";
string standardName = "Mawson Time";
TimeSpan offset = new TimeSpan(06, 00, 00);
TimeZoneInfo mawson = TimeZoneInfo.CreateCustomTimeZone(standardName,
offset, displayName, standardName);
Console.WriteLine("The current time is {0} {1}",
TimeZoneInfo.ConvertTime(DateTime.Now, TimeZoneInfo.Local,
mawson),
mawson.StandardName);
```

The string assigned to the [DisplayName](#) property follows a standard format in which the time zone's offset from UTC is followed by a friendly description of the time zone.

## Compiling the code

This example requires:

- That the following namespaces be imported:

C#

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

# See also

- [Dates, times, and time zones](#)
- [Time zone overview](#)
- [How to: Create time zones with adjustment rules](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Create time zones with adjustment rules

Article • 09/15/2021

The precise time zone information that is required by an application may not be present on a particular system for several reasons:

- The time zone has never been defined in the local system's registry.
- Data about the time zone has been modified or removed from the registry.
- The time zone does not have accurate information about time zone adjustments for a particular historic period.

In these cases, you can call the [CreateCustomTimeZone](#) method to define the time zone required by your application. You can use the overloads of this method to create a time zone with or without adjustment rules. If the time zone supports daylight saving time, you can define adjustments with either fixed or floating adjustment rules. (For definitions of these terms, see the "Time Zone Terminology" section in [Time zone overview](#).)

## ⓘ Important

Custom time zones created by calling the [CreateCustomTimeZone](#) method are not added to the registry. Instead, they can be accessed only through the object reference returned by the [CreateCustomTimeZone](#) method call.

This topic shows how to create a time zone with adjustment rules. To create a time zone that does not support daylight saving time adjustment rules, see [How to: Create Time Zones Without Adjustment Rules](#).

## To create a time zone with floating adjustment rules

1. For each adjustment (that is, for each transition away from and back to standard time over a particular time interval), do the following:
  - a. Define the starting transition time for the time zone adjustment.

You must call the [TimeZoneInfo.TransitionTime.CreateFloatingDateRule](#) method and pass it a [DateTime](#) value that defines the time of the transition, an integer value that defines the month of the transition, an integer value that defines the

week on which the transition occurs, and a [DayOfWeek](#) value that defines the day of the week on which the transition occurs. This method call instantiates a [TimeZoneInfo.TransitionTime](#) object.

- b. Define the ending transition time for the time zone adjustment. This requires another call to the [TimeZoneInfo.TransitionTime.CreateFloatingDateRule](#) method. This method call instantiates a second [TimeZoneInfo.TransitionTime](#) object.
  - c. Call the [CreateAdjustmentRule](#) method and pass it the effective start and end dates of the adjustment, a [TimeSpan](#) object that defines the amount of time in the transition, and the two [TimeZoneInfo.TransitionTime](#) objects that define when the transitions to and from daylight saving time occur. This method call instantiates a [TimeZoneInfo.AdjustmentRule](#) object.
  - d. Assign the [TimeZoneInfo.AdjustmentRule](#) object to an array of [TimeZoneInfo.AdjustmentRule](#) objects.
2. Define the time zone's display name. The display name follows a fairly standard format in which the time zone's offset from Coordinated Universal Time (UTC) is enclosed in parentheses and is followed by a string that identifies the time zone, one or more of the cities in the time zone, or one or more of the countries or regions in the time zone.
  3. Define the name of the time zone's standard time. Typically, this string is also used as the time zone's identifier.
  4. Define the name of the time zone's daylight time.
  5. If you want to use a different identifier than the time zone's standard name, define the time zone identifier.
  6. Instantiate a [TimeSpan](#) object that defines the time zone's offset from UTC. Time zones with times that are later than UTC have a positive offset. Time zones with times that are earlier than UTC have a negative offset.
  7. Call the [TimeZoneInfo.CreateCustomTimeZone\(String, TimeSpan, String, String, TimeZoneInfo+AdjustmentRule\[\]\)](#) method to instantiate the new time zone.

## Example

The following example defines a Central Standard Time zone for the United States that includes adjustment rules for a variety of time intervals from 1918 to the present.

C#

```
TimeZoneInfo cst;
// Declare necessary TimeZoneInfo.AdjustmentRule objects for time zone
TimeSpan delta = new TimeSpan(1, 0, 0);
TimeZoneInfo.AdjustmentRule adjustment;
List<TimeZoneInfo.AdjustmentRule> adjustmentList = new
List<TimeZoneInfo.AdjustmentRule>();
// Declare transition time variables to hold transition time information
TimeZoneInfo.TransitionTime transitionRuleStart, transitionRuleEnd;

// Define new Central Standard Time zone 6 hours earlier than UTC
// Define rule 1 (for 1918-1919)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 03, 05, DayOfWeek.Sunday);
transitionRuleEnd = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 10, 05, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1918, 1, 1), new DateTime(1919, 12, 31), delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 2 (for 1942)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 02, 09);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1942, 1, 1), new DateTime(1942, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 3 (for 1945)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 23, 0, 0), 08, 14);
transitionRuleEnd = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 09, 30);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1945, 1, 1), new DateTime(1945, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define end rule (for 1967-2006)
transitionRuleEnd = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 10, 5, DayOfWeek.Sunday);
// Define rule 4 (for 1967-73)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 04, 05, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1967, 1, 1), new DateTime(1973, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 5 (for 1974 only)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 01, 06);
```

```

adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1974, 1, 1), new DateTime(1974, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 6 (for 1975 only)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 02, 23);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1975, 1, 1), new DateTime(1975, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 7 (1976-1986)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 04, 05, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1976, 1, 1), new DateTime(1986, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 8 (1987-2006)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 04, 01, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1987, 1, 1), new DateTime(2006, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 9 (2007- )
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 03, 02, DayOfWeek.Sunday);
transitionRuleEnd = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 11, 01, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(2007, 1, 1), DateTime.MaxValue.Date,
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);

// Convert list of adjustment rules to an array
TimeZoneInfo.AdjustmentRule[] adjustments = new
TimeZoneInfo.AdjustmentRule[adjustmentList.Count];
adjustmentList.CopyTo(adjustments);

cst = TimeZoneInfo.CreateCustomTimeZone("Central Standard Time", new
TimeSpan(-6, 0, 0),
"(GMT-06:00) Central Time (US Only)", "Central Standard Time",
"Central Daylight Time", adjustments);

```

The time zone created in this example has multiple adjustment rules. Care must be taken to ensure that the effective start and end dates of any adjustment rule do not

overlap with the dates of another adjustment rule. If there is an overlap, an [InvalidTimeZoneException](#) is thrown.

For floating adjustment rules, the value 5 is passed to the `week` parameter of the [CreateFloatingDateRule](#) method to indicate that the transition occurs on the last week of a particular month.

In creating the array of [TimeZoneInfo.AdjustmentRule](#) objects to use in the [TimeZoneInfo.CreateCustomTimeZone\(String, TimeSpan, String, String, String, TimeZoneInfo+AdjustmentRule\[\]\)](#) method call, the code could initialize the array to the size required by the number of adjustments to be created for the time zone. Instead, this code example calls the [Add](#) method to add each adjustment rule to a generic `List<T>` collection of [TimeZoneInfo.AdjustmentRule](#) objects. The code then calls the [CopyTo](#) method to copy the members of this collection to the array.

The example also uses the [CreateFixedDateRule](#) method to define fixed-date adjustments. This is similar to calling the [CreateFloatingDateRule](#) method, except that it requires only the time, month, and day of the transition parameters.

The example can be tested using code such as the following:

C#

```
TimeZoneInfo est = TimeZoneInfo.FindSystemTimeZoneById("Eastern Standard
Time");

DateTime pastDate1 = new DateTime(1942, 2, 11);
Console.WriteLine("Is {0} daylight saving time: {1}", pastDate1,
                  cst.IsDaylightSavingTime(pastDate1));

DateTime pastDate2 = new DateTime(1967, 10, 29, 1, 30, 00);
Console.WriteLine("Is {0} ambiguous: {1}", pastDate2,
                  cst.IsAmbiguousTime(pastDate2));

DateTime pastDate3 = new DateTime(1974, 1, 7, 2, 59, 00);
Console.WriteLine("{0} {1} is {2} {3}", pastDate3,
                  est.IsDaylightSavingTime(pastDate3) ?
                  est.DaylightName : est.StandardName,
                  TimeZoneInfo.ConvertTime(pastDate3, est, cst),

                  cst.IsDaylightSavingTime(TimeZoneInfo.ConvertTime(pastDate3, est, cst)) ?
                  cst.DaylightName : cst.StandardName);
// This code produces the following output to the console:
//
//    Is 2/11/1942 12:00:00 AM daylight saving time: True
//    Is 10/29/1967 1:30:00 AM ambiguous: True
```

```
// 1/7/1974 2:59:00 AM Eastern Standard Time is 1/7/1974 2:59:00 AM  
Central Daylight Time
```

# Compiling the code

This example requires:

- That the following namespaces be imported:

C#

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

## See also

- [Dates, times, and time zones](#)
- [Time zone overview](#)
- [How to: Create time zones without adjustment rules](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### [.NET feedback](#)

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Finding the time zones defined on a local system

Article • 09/15/2021

The [TimeZoneInfo](#) class does not expose a public constructor. As a result, the `new` keyword cannot be used to create a new [TimeZoneInfo](#) object. Instead, [TimeZoneInfo](#) objects are instantiated either by retrieving information on predefined time zones from the registry or by creating a custom time zone. This topic discusses instantiating a time zone from data stored in the registry. In addition, `static` (`shared` in Visual Basic) properties of the [TimeZoneInfo](#) class provide access to Coordinated Universal Time (UTC) and the local time zone.

## ⓘ Note

For time zones that are not defined in the registry, you can create custom time zones by calling the overloads of the [CreateCustomTimeZone](#) method. Creating a custom time zone is discussed in the [How to: Create time zones without adjustment rules](#) and [How to: Create time zones with adjustment rules](#) topics. In addition, you can instantiate a [TimeZoneInfo](#) object by restoring it from a serialized string with the [FromSerializedString](#) method. Serializing and deserializing a [TimeZoneInfo](#) object is discussed in the [How to: Save time zones to an embedded resource](#) and [How to: Restore Time Zones from an Embedded Resource](#) topics.

## Accessing individual time zones

The [TimeZoneInfo](#) class provides two predefined time zone objects that represent the UTC time and the local time zone. They are available from the [Utc](#) and [Local](#) properties, respectively. For instructions on accessing the UTC or local time zones, see [How to: Access the predefined UTC and local time zone objects](#).

You can also instantiate a [TimeZoneInfo](#) object that represents any time zone defined in the registry. For instructions on instantiating a specific time zone object, see [How to: Instantiate a TimeZoneInfo object](#).

## Time zone identifiers

The time zone identifier is a key field that uniquely identifies the time zone. While most keys are relatively short, the time zone identifier is comparatively long. In most cases, its

value corresponds to the [TimeZoneInfo.StandardName](#) property, which is used to provide the name of the time zone's standard time. However, there are exceptions. The best way to make sure that you supply a valid identifier is to enumerate the time zones available on your system and note their associated identifiers.

## See also

- [Dates, times, and time zones](#)
- [How to: Access the predefined UTC and local time zone objects](#)
- [How to: Instantiate a TimeZoneInfo object](#)
- [Converting times between time zones](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### [.NET feedback](#)

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Enumerate time zones present on a computer

Article • 09/15/2021

Successfully working with a designated time zone requires that information about that time zone be available to the system. The Windows XP and Windows Vista operating systems store this information in the registry. However, although the total number of time zones that exist throughout the world is large, the registry contains information about only a subset of them. In addition, the registry itself is a dynamic structure whose contents are subject to both deliberate and accidental change. As a result, an application cannot always assume that a particular time zone is defined and available on a system. The first step for many applications that use time zone information applications is to determine whether required time zones are available on the local system, or to give the user a list of time zones from which to select. This requires that an application enumerate the time zones defined on a local system.

## ⓘ Note

If an application relies on the presence of a particular time zone that may not be defined on a local system, the application can ensure its presence by serializing and deserializing information about the time zone. The time zone can then be added to a list control so that the application user can select it. For details, see [How to: Save Time Zones to an Embedded Resource](#) and [How to: Restore time zones from an embedded resource](#).

## To enumerate the time zones present on the local system

1. Call the [TimeZoneInfo.GetSystemTimeZones](#) method. The method returns a generic [ReadOnlyCollection<T>](#) collection of [TimeZoneInfo](#) objects. The entries in the collection are sorted by their [DisplayName](#) property. For example:

C#

```
ReadOnlyCollection<TimeZoneInfo> tzCollection;  
tzCollection = TimeZoneInfo.GetSystemTimeZones();
```

2. Enumerate the individual [TimeZoneInfo](#) objects in the collection by using a [foreach](#) loop (in C#) or a [For Each...Next](#) loop (in Visual Basic), and perform any necessary processing on each object. For example, the following code enumerates

the `ReadOnlyCollection<T>` collection of `TimeZoneInfo` objects returned in step 1 and lists the display name of each time zone on the console.

```
C#  
  
foreach (TimeZoneInfo timeZone in tzCollection)  
    Console.WriteLine("  {0}: {1}", timeZone.Id, timeZone.DisplayName);
```

## To present the user with a list of time zones present on the local system

1. Call the `TimeZoneInfo.GetSystemTimeZones` method. The method returns a generic `ReadOnlyCollection<T>` collection of `TimeZoneInfo` objects.
2. Assign the collection returned in step 1 to the `DataSource` property of a Windows forms or ASP.NET list control.
3. Retrieve the `TimeZoneInfo` object that the user has selected.

The example provides an illustration for a Windows application.

## Example

The example starts a Windows application that displays the time zones defined on a system in a list box. The example then displays a dialog box that contains the value of the `DisplayName` property of the time zone object selected by the user.

```
C#  
  
private void Form1_Load(object sender, EventArgs e)  
{  
    ReadOnlyCollection<TimeZoneInfo> tzCollection;  
    tzCollection = TimeZoneInfo.GetSystemTimeZones();  
    this.timeZoneList.DataSource = tzCollection;  
}  
  
private void OkButton_Click(object sender, EventArgs e)  
{  
    TimeZoneInfo selectedTimeZone = (TimeZoneInfo)  
    this.timeZoneList.SelectedItem;  
    MessageBox.Show("You selected the " + selectedTimeZone.ToString() + "  
time zone.");  
}
```

Most list controls (such as the [System.Windows.Forms.ListBox](#) or [System.Web.UI.WebControls.BulletedList](#) control) allow you to assign a collection of object variables to their `DataSource` property as long as that collection implements the [IEnumerable](#) interface. (The generic [ReadOnlyCollection<T>](#) class does this.) To display an individual object in the collection, the control calls that object's `ToString` method to extract the string that is used to represent the object. In the case of [TimeZoneInfo](#) objects, the `ToString` method returns the [TimeZoneInfo](#) object's display name (the value of its [DisplayName](#) property).

### Note

Because list controls call an object's `ToString` method, you can assign a collection of [TimeZoneInfo](#) objects to the control, have the control display a meaningful name for each object, and retrieve the [TimeZoneInfo](#) object that the user has selected. This eliminates the need to extract a string for each object in the collection, assign the string to a collection that is in turn assigned to the control's `DataSource` property, retrieve the string the user has selected, and then use this string to extract the object that it describes.

## Compiling the code

This example requires:

- That the following namespaces be imported:

[System](#) (in C# code)

[System.Collections.ObjectModel](#)

## See also

- [Dates, times, and time zones](#)
- [How to: Save time zones to an embedded resource](#)
- [How to: Restore time zones from an embedded resource](#)

 Collaborate with us on  
GitHub

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Access the predefined UTC and local time zone objects

Article • 09/15/2021

The [TimeZoneInfo](#) class provides two properties, [Utc](#) and [Local](#), that give your code access to predefined time zone objects. This topic discusses how to access the [TimeZoneInfo](#) objects returned by those properties.

## To access the Coordinated Universal Time (UTC) [TimeZoneInfo](#) object

1. Use the `static` (`Shared` in Visual Basic) [TimeZoneInfo.Utc](#) property to access Coordinated Universal Time.
2. Rather than assigning the [TimeZoneInfo](#) object returned by the property to an object variable, continue to access Coordinated Universal Time through the [TimeZoneInfo.Utc](#) property.

## To access the local time zone

1. Use the `static` (`Shared` in Visual Basic) [TimeZoneInfo.Local](#) property to access the local system time zone.
2. Rather than assigning the [TimeZoneInfo](#) object returned by the property to an object variable, continue to access the local time zone through the [TimeZoneInfo.Local](#) property.

## Example

The following code uses the [TimeZoneInfo.Local](#) and [TimeZoneInfo.Utc](#) properties to convert a time from the U.S. and Canadian Eastern Standard time zone, as well as to display the time zone name to the console.

C#

```
// Create Eastern Standard Time value and TimeZoneInfo object
DateTime estTime = new DateTime(2007, 1, 1, 00, 00, 00);
string timeZoneName = "Eastern Standard Time";
try
{
    TimeZoneInfo est = TimeZoneInfo.FindSystemTimeZoneById(timeZoneName);
```

```

// Convert EST to local time
DateTime localTime = TimeZoneInfo.ConvertTime(estTime, est,
TimeZoneInfo.Local);
Console.WriteLine("At {0} {1}, the local time is {2} {3}.",
    estTime,
    est,
    localTime,
    TimeZoneInfo.Local.IsDaylightSavingTime(localTime) ?
        TimeZoneInfo.Local.DaylightName :
        TimeZoneInfo.Local.StandardName);

// Convert EST to UTC
DateTime utcTime = TimeZoneInfo.ConvertTime(estTime, est,
TimeZoneInfo.Utc);
Console.WriteLine("At {0} {1}, the time is {2} {3}.",
    estTime,
    est,
    utcTime,
    TimeZoneInfo.Utc.StandardName);
}

catch (TimeZoneNotFoundException)
{
    Console.WriteLine("The {0} zone cannot be found in the registry.",
        timeZoneName);
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine("The registry contains invalid data for the {0} zone.",
        timeZoneName);
}

// The example produces the following output to the console:
//    At 1/1/2007 12:00:00 AM (UTC-05:00) Eastern Time (US & Canada), the
//    local time is 1/1/2007 12:00:00 AM Eastern Standard Time.
//    At 1/1/2007 12:00:00 AM (UTC-05:00) Eastern Time (US & Canada), the
//    time is 1/1/2007 5:00:00 AM UTC.

```

You should always access the local time zone through the [TimeZoneInfo.Local](#) property rather than assigning the local time zone to a [TimeZoneInfo](#) object variable. Similarly, you should always access Coordinated Universal Time through the [TimeZoneInfo.Utc](#) property rather than assigning the UTC zone to a [TimeZoneInfo](#) object variable. This prevents the [TimeZoneInfo](#) object variable from being invalidated by a call to the [TimeZoneInfo.ClearCachedData](#) method.

## Compiling the code

This example requires:

- That the `System` namespace be imported with the `using` statement (required in C# code).

## See also

- [Dates, times, and time zones](#)
- [Finding the time zones defined on a local system](#)
- [How to: Instantiate a `TimeZoneInfo` object](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### [.NET feedback](#)

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Instantiate a TimeZoneInfo object

Article • 09/15/2021

The most common way to instantiate a [TimeZoneInfo](#) object is to retrieve information about it from the registry. This topic discusses how to instantiate a [TimeZoneInfo](#) object from the local system registry.

## To instantiate a TimeZoneInfo object

1. Declare a [TimeZoneInfo](#) object.
2. Call the `static` (`Shared` in Visual Basic) [TimeZoneInfo.FindSystemTimeZoneById](#) method.
3. Handle any exceptions thrown by the method, particularly the [TimeZoneNotFoundException](#) that is thrown if the time zone is not defined in the registry.

## Example

The following code retrieves a [TimeZoneInfo](#) object that represents the Eastern Standard Time zone and displays the Eastern Standard time that corresponds to the local time.

C#

```
DateTime timeNow = DateTime.Now;
try
{
    TimeZoneInfo easternZone = TimeZoneInfo.FindSystemTimeZoneById("Eastern
Standard Time");
    DateTime easternTimeNow = TimeZoneInfo.ConvertTime(timeNow,
TimeZoneInfo.Local,
                                                    easternZone);
    Console.WriteLine("{0} {1} corresponds to {2} {3}.",
                      timeNow,
                      TimeZoneInfo.Local.IsDaylightSavingTime(timeNow) ?
                      TimeZoneInfo.Local.DaylightName :
                      TimeZoneInfo.Local.StandardName,
                      easternTimeNow,
                      easternZone.IsDaylightSavingTime(easternTimeNow) ?
                      easternZone.DaylightName :
                      easternZone.StandardName);
}
```

```

// Handle exception
//
// As an alternative to simply displaying an error message, an alternate
// Eastern
// Standard Time TimeZoneInfo object could be instantiated here either by
// restoring
// it from a serialized string or by providing the necessary data to the
// CreateCustomTimeZone method.
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("The Eastern Standard Time Zone cannot be found on the
local system.");
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine("The Eastern Standard Time Zone contains invalid or
missing data.");
}
catch (SecurityException)
{
    Console.WriteLine("The application lacks permission to read time zone
information from the registry.");
}
catch (OutOfMemoryException)
{
    Console.WriteLine("Not enough memory is available to load information on
the Eastern Standard Time zone.");
}
// If we weren't passing FindSystemTimeZoneById a literal string, we also
// would handle an ArgumentNullException.

```

The `TimeZoneInfo.FindSystemTimeZoneById` method's single parameter is the identifier of the time zone that you want to retrieve, which corresponds to the object's `TimeZoneInfo.Id` property. The time zone identifier is a key field that uniquely identifies the time zone. While most keys are relatively short, the time zone identifier is comparatively long. In most cases, its value corresponds to the `StandardName` property of a `TimeZoneInfo` object, which is used to provide the name of the time zone's standard time. However, there are exceptions. The best way to make sure that you supply a valid identifier is to enumerate the time zones available on your system and note the identifiers of the time zones present on them. For an illustration, see [How to: Enumerate time zones present on a computer](#). The [Finding the time zones defined on a local system](#) topic also contains a list of selected time zone identifiers.

If the time zone is found, the method returns its `TimeZoneInfo` object. If the time zone is not found, the method throws a `TimeZoneNotFoundException`. If the time zone is found but its data is corrupted or incomplete, the method throws an `InvalidTimeZoneException`.

If your application relies on a time zone that must be present, you should first call the [FindSystemTimeZoneById](#) method to retrieve the time zone information from the registry. If the method call fails, your exception handler should then either create a new instance of the time zone or re-create it by deserializing a serialized [TimeZoneInfo](#) object. See [How to: Restore time zones from an embedded resource](#) for an example.

## See also

- [Dates, times, and time zones](#)
- [Finding the time zones defined on a local system](#)
- [How to: Access the predefined UTC and local time zone objects](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### [.NET feedback](#)

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Saving and restoring time zones

Article • 09/15/2021

The [TimeZoneInfo](#) class relies on the registry to retrieve predefined time zone data. However, the registry is a dynamic structure. Additionally, the time zone information that the registry contains is used by the operating system primarily to handle time adjustments and conversions for the current year. This has two major implications for applications that rely on accurate time zone data:

- A time zone that is required by an application may not be defined in the registry, or it may have been renamed or removed from the registry.
- A time zone that is defined in the registry may lack information about the particular adjustment rules that are necessary for historical time zone conversions.

The [TimeZoneInfo](#) class addresses these limitations through its support for serialization (saving) and deserialization (restoring) of time zone data.

## Time zone serialization and deserialization

Saving and restoring a time zone by serializing and deserializing time zone data involves just two method calls:

- You can serialize a [TimeZoneInfo](#) object by calling that object's [ToSerializedString](#) method. The method takes no parameters and returns a string that contains time zone information.
- You can deserialize a [TimeZoneInfo](#) object from a serialized string by passing that string to the `static (Shared in Visual Basic)` [TimeZoneInfo.FromSerializedString](#) method.

## Serialization and deserialization scenarios

The ability to save (or serialize) a [TimeZoneInfo](#) object to a string and to restore (or deserialize) it for later use increases both the utility and the flexibility of the [TimeZoneInfo](#) class. This section examines some of the situations in which serialization and deserialization are most useful.

## Serializing and deserializing time zone data in an application

A serialized time zone can be restored from a string when it is needed. An application might do this if the time zone retrieved from the registry is unable to correctly convert a date and time within a particular date range. For example, time zone data in the Windows XP registry supports a single adjustment rule, while time zones defined in the Windows Vista registry typically provide information about two adjustment rules. This means that historical time conversions may be inaccurate. Serialization and deserialization of time zone data can handle this limitation.

In the following example, a custom `TimeZoneInfo` class that has no adjustment rules is defined to represent the U.S. Eastern Standard Time zone from 1883 to 1917, before the introduction of daylight saving time in the United States. The custom time zone is serialized in a variable that has global scope. The time zone conversion method, `ConvertUtcTime`, is passed Coordinated Universal Time (UTC) times to convert. If the date and time occurs in 1917 or earlier, the custom Eastern Standard Time zone is restored from a serialized string and replaces the time zone retrieved from the registry.

C#

```
using System;

public class TimeZoneSerialization
{
    static string serializedEst;

    public static void Main()
    {
        // Retrieve Eastern Standard Time zone from registry
        try
        {
            TimeZoneSerialization tzs = new TimeZoneSerialization();
            TimeZoneInfo est = TimeZoneInfo.FindSystemTimeZoneById("Eastern
Standard Time");
            // Create custom Eastern Time Zone for historical (pre-1918)
            // conversions
            CreateTimeZone();
            // Call conversion function with one current and one pre-1918 date
            and time
            Console.WriteLine(ConvertUtcTime(DateTime.UtcNow, est));
            Console.WriteLine(ConvertUtcTime(new DateTime(1900, 11, 15, 9, 32,
00, DateTimeKind.Utc), est));
        }
        catch (TimeZoneNotFoundException)
        {
            Console.WriteLine("The Eastern Standard Time zone is not in the
registry.");
        }
        catch (InvalidTimeZoneException)
        {
            Console.WriteLine("Data on the Eastern Standard Time Zone in the
registry is corrupted.");
        }
    }
}
```

```
        }

    }

    private static void CreateTimeZone()
    {
        // Create a simple Eastern Standard time zone
        // without adjustment rules for 1883-1918
        TimeZoneInfo earlyEstZone = TimeZoneInfo.CreateCustomTimeZone("Eastern
Standard Time",
            new TimeSpan(-5, 0, 0),
            " (GMT-05:00) Eastern Time (United
States)",
            "Eastern Standard Time");
        serializedEst = earlyEstZone.ToSerializedString();
    }

    private static DateTime ConvertUtcTime(DateTime utcDate, TimeZoneInfo tz)
    {
        // Use time zone object from registry
        if (utcDate.Year > 1917)
        {
            return TimeZoneInfo.ConvertTimeFromUtc(utcDate, tz);
        }
        // Handle dates before introduction of DST
        else
        {
            // Restore serialized time zone object
            tz = TimeZoneInfo.FromSerializedString(serializedEst);
            return TimeZoneInfo.ConvertTimeFromUtc(utcDate, tz);
        }
    }
}
```

## Handling time zone exceptions

Because the registry is a dynamic structure, its contents are subject to accidental or deliberate modification. This means that a time zone that should be defined in the registry and that is required for an application to execute successfully may be absent. Without support for time zone serialization and deserialization, you have little choice but to handle the resulting `TimeZoneNotFoundException` by ending the application. However, by using time zone serialization and deserialization, you can handle an unexpected `TimeZoneNotFoundException` by restoring the required time zone from a serialized string, and the application will continue to run.

The following example creates and serializes a custom Central Standard Time zone. It then tries to retrieve the Central Standard Time zone from the registry. If the retrieval operation throws either a [TimeZoneNotFoundException](#) or an [InvalidTimeZoneException](#), the exception handler deserializes the time zone.

C#

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

public class TimeZoneApplication
{
    // Define collection of custom time zones
    private Dictionary<string, string> customTimeZones = new
    Dictionary<string, string>();
    private TimeZoneInfo cst;

    public TimeZoneApplication()
    {
        // Create custom Central Standard Time
        //
        // Declare necessary TimeZoneInfo.AdjustmentRule objects for time zone
        TimeZoneInfo customTimeZone;
        TimeSpan delta = new TimeSpan(1, 0, 0);
        TimeZoneInfo.AdjustmentRule adjustment;
        List<TimeZoneInfo.AdjustmentRule> adjustmentList = new
        List<TimeZoneInfo.AdjustmentRule>();
        // Declare transition time variables to hold transition time
        // information
        TimeZoneInfo.TransitionTime transitionRuleStart, transitionRuleEnd;

        // Define end rule (for 1976-2006)
        transitionRuleEnd =
        TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
        0, 0), 10, 5, DayOfWeek.Sunday);
        // Define rule (1976-1986)
        transitionRuleStart =
        TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
        0, 0), 04, 05, DayOfWeek.Sunday);
        adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
        DateTime(1976, 1, 1), new DateTime(1986, 12, 31), delta,
        transitionRuleStart, transitionRuleEnd);
        adjustmentList.Add(adjustment);
        // Define rule (1987-2006)
        transitionRuleStart =
        TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
        0, 0), 04, 01, DayOfWeek.Sunday);
        adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
        DateTime(1987, 1, 1), new DateTime(2006, 12, 31), delta,
        transitionRuleStart, transitionRuleEnd);
        adjustmentList.Add(adjustment);
        // Define rule (2007- )
        transitionRuleStart =
        TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
        0, 0), 03, 02, DayOfWeek.Sunday);
        transitionRuleEnd =
        TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
        0, 0), 11, 01, DayOfWeek.Sunday);
        adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
```

```

        DateTime(2007, 01, 01), DateTime.MaxValue.Date, delta, transitionRuleStart,
        transitionRuleEnd);
        adjustmentList.Add(adjustment);

        // Create custom U.S. Central Standard Time zone
        customTimeZone = TimeZoneInfo.CreateCustomTimeZone("Central Standard
Time",
            new TimeSpan(-6, 0, 0),
            "(GMT-06:00) Central Time (US Only)", "Central
Standard Time",
            "Central Daylight Time", adjustmentList.ToArray());
        // Add time zone to collection
        customTimeZones.Add(customTimeZone.Id,
        customTimeZone.ToSerializedString());

        // Create any other required time zones
    }

    public static void Main()
{
    TimeZoneApplication tza = new TimeZoneApplication();
    tza.AppEntryPoint();
}

private void AppEntryPoint()
{
    try
    {
        cst = TimeZoneInfo.FindSystemTimeZoneById("Central Standard Time");
    }
    catch (TimeZoneNotFoundException)
    {
        if (customTimeZones.ContainsKey("Central Standard Time"))
            HandleTimeZoneException("Central Standard Time");
    }
    catch (InvalidTimeZoneException)
    {
        if (customTimeZones.ContainsKey("Central Standard Time"))
            HandleTimeZoneException("Central Standard Time");
    }
    if (cst == null)
    {
        Console.WriteLine("Unable to load Central Standard Time zone.");
        return;
    }
    DateTime currentTime = DateTime.Now;
    Console.WriteLine("The current {0} time is {1}.",
        TimeZoneInfo.Local.IsDaylightSavingTime(currentTime)
    ?
        TimeZoneInfo.Local.StandardName :
        TimeZoneInfo.Local.DaylightName,
        currentTime.ToString("f"));
    Console.WriteLine("The current {0} time is {1}.",
        cst.IsDaylightSavingTime(currentTime) ?
        cst.StandardName :

```

```
        cst.DaylightName,
        TimeZoneInfo.ConvertTime(currentTime,
TimeZoneInfo.Local, cst).ToString("F"));
    }

    private void HandleTimeZoneException(string timeZoneName)
    {
        string tzString = customTimeZones[timeZoneName];
        cst = TimeZoneInfo.FromSerializedString(tzString);
    }
}
```

## Storing a serialized string and restoring it when needed

The previous examples have stored time zone information to a string variable and restored it when needed. However, the string that contains serialized time zone information can itself be stored in some storage medium, such as an external file, a resource file embedded in the application, or the registry. (Note that information about custom time zones should be stored apart from the system's time zone keys in the registry.)

Storing a serialized time zone string in this manner also separates the time zone creation routine from the application itself. For example, a time zone creation routine can execute and create a data file that contains historical time zone information that an application can use. The data file can be then be installed with the application, and it can be opened and one or more of its time zones can be deserialized when the application requires them.

For an example that uses an embedded resource to store serialized time zone data, see [How to: Save time zones to an embedded resource](#) and [How to: Restore time zones from an embedded resource](#).

## See also

- [Dates, times, and time zones](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

# How to: Save time zones to an embedded resource

Article • 09/15/2021

A time zone-aware application often requires the presence of a particular time zone. However, because the availability of individual [TimeZoneInfo](#) objects depends on information stored in the local system's registry, even customarily available time zones may be absent. In addition, information about custom time zones instantiated by using the [CreateCustomTimeZone](#) method is not stored with other time zone information in the registry. To ensure that these time zones are available when they are needed, you can save them by serializing them, and later restore them by deserializing them.

Typically, serializing a [TimeZoneInfo](#) object occurs apart from the time zone-aware application. Depending on the data store used to hold serialized [TimeZoneInfo](#) objects, time zone data may be serialized as part of a setup or installation routine (for example, when the data is stored in an application key of the registry), or as part of a utility routine that runs before the final application is compiled (for example, when the serialized data is stored in a .NET XML resource (.resx) file).

In addition to a resource file that is compiled with the application, several other data stores can be used for time zone information. These include the following:

- The registry. Note that an application should use the subkeys of its own application key to store custom time zone data rather than using the subkeys of `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Time Zones`.
- Configuration files.
- Other system files.

## To save a time zone by serializing it to a .resx file

1. Retrieve an existing time zone or create a new time zone.

To retrieve an existing time zone, see [How to: Access the predefined UTC and local time zone objects](#) and [How to: Instantiate a TimeZoneInfo object](#).

To create a new time zone, call one of the overloads of the [CreateCustomTimeZone](#) method. For more information, see [How to: Create time zones without adjustment rules](#) and [How to: Create time zones with adjustment rules](#).

2. Call the [ToSerializedString](#) method to create a string that contains the time zone's data.
3. Instantiate a [StreamWriter](#) object by providing the name and optionally the path of the .resx file to the [StreamWriter](#) class constructor.
4. Instantiate a [ResXResourceWriter](#) object by passing the [StreamWriter](#) object to the [ResXResourceWriter](#) class constructor.
5. Pass the time zone's serialized string to the [ResXResourceWriter.AddResource](#) method.
6. Call the [ResXResourceWriter.Generate](#) method.
7. Call the [ResXResourceWriter.Close](#) method.
8. Close the [StreamWriter](#) object by calling its [Close](#) method.
9. Add the generated .resx file to the application's Visual Studio project.
10. Using the [Properties](#) window in Visual Studio, make sure that the .resx file's **Build Action** property is set to **Embedded Resource**.

## Example

The following example serializes a [TimeZoneInfo](#) object that represents Central Standard Time and a [TimeZoneInfo](#) object that represents the Palmer Station, Antarctica time to a .NET XML resource file that is named SerializedTimeZones.resx. Central Standard Time is typically defined in the registry; Palmer Station, Antarctica is a custom time zone.

C#

```
TimeZoneSerialization()
{
    TextWriter writeStream;
    Dictionary<string, string> resources = new Dictionary<string, string>();
    // Determine if .resx file exists
    if (File.Exists(resxName))
    {
        // Open reader
        TextReader readStream = new StreamReader(resxName);
        ResXResourceReader resReader = new ResXResourceReader(readStream);
        foreach (DictionaryEntry item in resReader)
        {
            if (! (((string) item.Key) == "CentralStandardTime" ||
                   ((string) item.Key) == "PalmerStandardTime" ))
                resources.Add((string) item.Key, (string) item.Value);
        }
    }
}
```

```

        readStream.Close();
        // Delete file, since write method creates duplicate xml headers
        File.Delete(resxName);
    }

    // Open stream to write to .resx file
    try
    {
        writeStream = new StreamWriter(resxName, true);
    }
    catch (FileNotFoundException e)
    {
        // Handle failure to find file
        Console.WriteLine("{0}: The file {1} could not be found.",
e.GetType().Name, resxName);
        return;
    }

    // Get resource writer
    ResXResourceWriter resWriter = new ResXResourceWriter(writeStream);

    // Add resources from existing file
    foreach (KeyValuePair<string, string> item in resources)
    {
        resWriter.AddResource(item.Key, item.Value);
    }

    // Serialize Central Standard Time
    try
    {
        TimeZoneInfo cst = TimeZoneInfo.FindSystemTimeZoneById("Central
Standard Time");
        resWriter.AddResource(cst.Id.Replace(" ", string.Empty),
cst.ToSerializedString());
    }
    catch (TimeZoneNotFoundException)
    {
        Console.WriteLine("The Central Standard Time zone could not be
found.");
    }

    // Create time zone for Palmer, Antarctica
    //
    // Define transition times to/from DST
    TimeZoneInfo.TransitionTime startTransition =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 4,
0, 0),

10, 2, DayOfWeek.Sunday);
    TimeZoneInfo.TransitionTime endTransition =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 3,
0, 0),

3, 2, DayOfWeek.Sunday);
    // Define adjustment rule

```

```

        TimeSpan delta = new TimeSpan(1, 0, 0);
        TimeZoneInfo.AdjustmentRule adjustment =
            TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new DateTime(1999, 10, 1),
                DateTime.MaxValue.Date, delta,
                startTransition, endTransition);
        // Create array for adjustment rules
        TimeZoneInfo.AdjustmentRule[] adjustments = {adjustment};
        // Define other custom time zone arguments
        string DisplayName = "(GMT-04:00) Antarctica/Palmer Time";
        string standardName = "Palmer Standard Time";
        string daylightName = "Palmer Daylight Time";
        TimeSpan offset = new TimeSpan(-4, 0, 0);
        TimeZoneInfo palmer = TimeZoneInfo.CreateCustomTimeZone(standardName,
        offset, DisplayName, standardName, daylightName, adjustments);
        resWriter.AddResource(palmer.Id.Replace(" ", String.Empty),
        palmer.ToSerializedString());

        // Save changes to .resx file
        resWriter.Generate();
        resWriter.Close();
        writeStream.Close();
    }
}

```

This example serializes [TimeZoneInfo](#) objects so that they are available in a resource file at compile time.

Because the [ResXResourceWriter.Generate](#) method adds complete header information to a .NET XML resource file, it cannot be used to add resources to an existing file. The example handles this by checking for the SerializedTimeZones.resx file and, if it exists, storing all of its resources other than the two serialized time zones to a generic [Dictionary<TKey, TValue>](#) object. The existing file is then deleted and the existing resources are added to a new SerializedTimeZones.resx file. The serialized time zone data is also added to this file.

The key (or [Name](#)) fields of resources should not contain embedded spaces. The [Replace\(String, String\)](#) method is called to remove all embedded spaces in the time zone identifiers before they are assigned to the resource file.

## Compiling the code

This example requires:

- That a reference to `System.Windows.Forms.dll` and `System.Core.dll` be added to the project.
- That the following namespaces be imported:

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Reflection;
using System.Resources;
using System.Windows.Forms;
```

## See also

- [Dates, times, and time zones](#)
- [Time zone overview](#)
- [How to: Restore time zones from an embedded resource](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# How to: Restore time zones from an embedded resource

Article • 09/15/2021

This topic describes how to restore time zones that have been saved in a resource file. For information and instructions about saving time zones, see [How to: Save time zones to an embedded resource](#).

## To deserialize a `TimeZoneInfo` object from an embedded resource

1. If the time zone to be retrieved is not a custom time zone, try to instantiate it by using the [FindSystemTimeZoneByIId](#) method.
2. Instantiate a [ResourceManager](#) object by passing the fully qualified name of the embedded resource file and a reference to the assembly that contains the resource file.

If you cannot determine the fully qualified name of the embedded resource file, use the [Ilasm.exe \(IL Disassembler\)](#) to examine the assembly's manifest. An `.mresource` entry identifies the resource. In the example, the resource's fully qualified name is `SerializeTimeZoneData.SerializedTimeZones`.

If the resource file is embedded in the same assembly that contains the time zone instantiation code, you can retrieve a reference to it by calling the `static` (Shared in Visual Basic) [GetExecutingAssembly](#) method.

3. If the call to the [FindSystemTimeZoneByIId](#) method fails, or if a custom time zone is to be instantiated, retrieve a string that contains the serialized time zone by calling the [ResourceManager.GetString](#) method.
4. Deserialize the time zone data by calling the [FromSerializedString](#) method.

## Example

The following example deserializes a `TimeZoneInfo` object stored in an embedded .NET XML resource file.

C#

```

private void DeserializeTimeZones()
{
    TimeZoneInfo cst, palmer;
    string timeZoneString;
    ResourceManager resMgr = new
    ResourceManager("SerializeTimeZoneData.SerializedTimeZones",
    this.GetType().Assembly);

    // Attempt to retrieve time zone from system
    try
    {
        cst = TimeZoneInfo.FindSystemTimeZoneById("Central Standard Time");
    }
    catch (TimeZoneNotFoundException)
    {
        // Time zone not in system; retrieve from resource
        timeZoneString = resMgr.GetString("CentralStandardTime");
        if (!String.IsNullOrEmpty(timeZoneString))
        {
            cst = TimeZoneInfo.FromSerializedString(timeZoneString);
        }
        else
        {
            MessageBox.Show("Unable to create Central Standard Time Zone.
Application must exit.", "Application Error");
            return;
        }
    }
    // Retrieve custom time zone
    try
    {
        timeZoneString = resMgr.GetString("PalmerStandardTime");
        palmer = TimeZoneInfo.FromSerializedString(timeZoneString);
    }
    catch (MissingManifestResourceException)
    {
        MessageBox.Show("Unable to retrieve the Palmer Standard Time Zone from
the resource file. Application must exit.");
        return;
    }
}

```

This code illustrates exception handling to ensure that a [TimeZoneInfo](#) object required by the application is present. It first tries to instantiate a [TimeZoneInfo](#) object by retrieving it from the registry using the [FindSystemTimeZoneById](#) method. If the time zone cannot be instantiated, the code retrieves it from the embedded resource file.

Because data for custom time zones (time zones instantiated by using the [CreateCustomTimeZone](#) method) are not stored in the registry, the code does not call the [FindSystemTimeZoneById](#) to instantiate the time zone for Palmer, Antarctica.

Instead, it immediately looks to the embedded resource file to retrieve a string that contains the time zone's data before it calls the [FromSerializedString](#) method.

## Compiling the code

This example requires:

- That a reference to `System.Windows.Forms.dll` and `System.Core.dll` be added to the project.
- That the following namespaces be imported:

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Reflection;
using System.Resources;
using System.Windows.Forms;
```

## See also

- [Dates, times, and time zones](#)
- [Time zone overview](#)
- [How to: Save time zones to an embedded resource](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

### [Open a documentation issue](#)

### [Provide product feedback](#)

# System.DateTime struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

## ⓘ Important

Eras in the Japanese calendars are based on the emperor's reign and are therefore expected to change. For example, May 1, 2019 marked the beginning of the Reiwa era in the [JapaneseCalendar](#) and [JapaneseLunisolarCalendar](#). Such a change of era affects all applications that use these calendars. For more information and to determine whether your applications are affected, see [Handling a new era in the Japanese calendar in .NET](#). For information on testing your applications on Windows systems to ensure their readiness for the era change, see [Prepare your application for the Japanese era change](#). For features in .NET that support calendars with multiple eras and for best practices when working with calendars that support multiple eras, see [Working with eras](#).

## Overview

The [DateTime](#) value type represents dates and times with values ranging from 00:00:00 (midnight), January 1, 0001 Anno Domini (Common Era) through 11:59:59 P.M., December 31, 9999 A.D. (C.E.) in the Gregorian calendar.

Time values are measured in 100-nanosecond units called ticks. A particular date is the number of ticks since 12:00 midnight, January 1, 0001 A.D. (C.E.) in the [GregorianCalendar](#) calendar. The number excludes ticks that would be added by leap seconds. For example, a ticks value of 31241376000000000L represents the date Friday, January 01, 0100 12:00:00 midnight. A [DateTime](#) value is always expressed in the context of an explicit or default calendar.

## ⓘ Note

If you're working with a ticks value that you want to convert to some other time interval, such as minutes or seconds, you should use the [TimeSpan.TicksPerDay](#), [TimeSpan.TicksPerHour](#), [TimeSpan.TicksPerMinute](#), [TimeSpan.TicksPerSecond](#), or [TimeSpan.TicksPerMillisecond](#) constant to perform the conversion. For example, to add the number of seconds represented by a specified number of ticks to the

Second component of a `DateTime` value, you can use the expression

```
dateValue.Second + nTicks/Timespan.TicksPerSecond.
```

You can view the source for the entire set of examples from this article in either [Visual Basic](#), [F#](#), or [C#](#).

#### ⓘ Note

An alternative to the `DateTime` structure for working with date and time values in particular time zones is the `DateTimeOffset` structure. The `DateTimeOffset` structure stores date and time information in a private `DateTime` field and the number of minutes by which that date and time differs from UTC in a private `Int16` field. This makes it possible for a `DateTimeOffset` value to reflect the time in a particular time zone, whereas a `DateTime` value can unambiguously reflect only UTC and the local time zone's time. For a discussion about when to use the `DateTime` structure or the `DateTimeOffset` structure when working with date and time values, see [Choosing Between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo](#).

## Quick links to example code

#### ⓘ Note

Some C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The [local time zone](#) of the [Try.NET](#) inline code runner and playground is Coordinated Universal Time, or UTC. This may affect the behavior and the output of examples that illustrate the `DateTime`, `DateTimeOffset`, and `TimeZoneInfo` types and their members.

This article includes several examples that use the `DateTime` type:

## Initialization examples

- [Invoke a constructor](#)

- Invoke the implicit parameterless constructor
- Assignment from return value
- Parsing a string that represents a date and time
- Visual Basic syntax to initialize a date and time

## Format `DateTime` objects as strings examples

- Use the default date time format
- Format a date and time using a specific culture
- Format a date time using a standard or custom format string
- Specify both a format string and a specific culture
- Format a date time using the ISO 8601 standard for web services

## Parse strings as `DateTime` objects examples

- Use Parse or TryParse to convert a string to a date and time
- Use ParseExact or TryParseExact to convert a string in a known format
- Convert from the ISO 8601 string representation to a date and time

## `DateTime` resolution examples

- Explore the resolution of date and time values
- Comparing for equality within a tolerance

## Culture and calendars examples

- Display date and time values using culture specific calendars
- Parse strings according to a culture specific calendar
- Initialize a date and time from a specific culture's calendar
- Accessing date and time properties using a specific culture's calendar
- Retrieving the week of the year using culture specific calendars

## Persistence examples

- Persisting date and time values as strings in the local time zone
- Persisting date and time values as strings in a culture and time invariant format
- Persisting date and time values as integers
- Persisting date and time values using the XmlSerializer

# Initialize a DateTime object

You can assign an initial value to a new `DateTime` value in many different ways:

- Calling a constructor, either one where you specify arguments for values, or use the implicit parameterless constructor.
- Assigning a `DateTime` to the return value of a property or method.
- Parsing a `DateTime` value from its string representation.
- Using Visual Basic-specific language features to instantiate a `DateTime`.

The following code snippets show examples of each.

## Invoke constructors

You call any of the overloads of the `DateTime` constructor that specify elements of the date and time value (such as the year, month, and day, or the number of ticks). The following code creates a specific date using the `DateTime` constructor specifying the year, month, day, hour, minute, and second.

```
C#
```

```
var date1 = new DateTime(2008, 5, 1, 8, 30, 52);
Console.WriteLine(date1);
```

You invoke the `DateTime` structure's implicit parameterless constructor when you want a `DateTime` initialized to its default value. (For details on the implicit parameterless constructor of a value type, see [Value Types](#).) Some compilers also support declaring a `DateTime` value without explicitly assigning a value to it. Creating a value without an explicit initialization also results in the default value. The following example illustrates the `DateTime` implicit parameterless constructor in C# and Visual Basic, as well as a `DateTime` declaration without assignment in Visual Basic.

```
C#
```

```
var dat1 = new DateTime();
// The following method call displays 1/1/0001 12:00:00 AM.
Console.WriteLine(dat1.ToString(System.Globalization.CultureInfo.InvariantCulture));
// The following method call displays True.
Console.WriteLine(dat1.Equals(DateTime.MinValue));
```

## Assign a computed value

You can assign the `DateTime` object a date and time value returned by a property or method. The following example assigns the current date and time, the current Coordinated Universal Time (UTC) date and time, and the current date to three new `DateTime` variables.

C#

```
DateTime date1 = DateTime.Now;  
DateTime date2 = DateTime.UtcNow;  
DateTime date3 = DateTime.Today;
```

## Parse a string that represents a `DateTime`

The `Parse`, `ParseExact`, `TryParse`, and `TryParseExact` methods all convert a string to its equivalent date and time value. The following examples use the `Parse` and `ParseExact` methods to parse a string and convert it to a `DateTime` value. The second format uses a form supported by the [ISO 8601](#) standard for a representing date and time in string format. This standard representation is often used to transfer date information in web services.

C#

```
var dateString = "5/1/2008 8:30:52 AM";  
DateTime date1 = DateTime.Parse(dateString,  
    System.Globalization.CultureInfo.InvariantCulture);  
var iso8601String = "20080501T08:30:52Z";  
DateTime dateISO8602 = DateTime.ParseExact(iso8601String,  
    "yyyyMMddTHH:mm:ssZ",  
    System.Globalization.CultureInfo.InvariantCulture);
```

The `TryParse` and `TryParseExact` methods indicate whether a string is a valid representation of a `DateTime` value and, if it is, performs the conversion.

## Language-specific syntax for Visual Basic

The following Visual Basic statement initializes a new `DateTime` value.

VB

```
Dim date1 As Date = #5/1/2008 8:30:52AM#
```

# DateTime values and their string representations

Internally, all [DateTime](#) values are represented as the number of ticks (the number of 100-nanosecond intervals) that have elapsed since 12:00:00 midnight, January 1, 0001. The actual [DateTime](#) value is independent of the way in which that value appears when displayed. The appearance of a [DateTime](#) value is the result of a formatting operation that converts a value to its string representation.

The appearance of date and time values is dependent on culture, international standards, application requirements, and personal preference. The [DateTime](#) structure offers flexibility in formatting date and time values through overloads of [ToString](#). The default [DateTime.ToString\(\)](#) method returns the string representation of a date and time value using the current culture's short date and long time pattern. The following example uses the default [DateTime.ToString\(\)](#) method. It displays the date and time using the short date and long time pattern for the current culture. The en-US culture is the current culture on the computer on which the example was run.

```
C#
```

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString());
// For en-US culture, displays 3/1/2008 7:00:00 AM
```

You may need to format dates in a specific culture to support web scenarios where the server may be in a different culture from the client. You specify the culture using the [DateTime.ToString\(IFormatProvider\)](#) method to create the short date and long time representation in a specific culture. The following example uses the [DateTime.ToString\(IFormatProvider\)](#) method to display the date and time using the short date and long time pattern for the fr-FR culture.

```
C#
```

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString(System.Globalization.CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 01/03/2008 07:00:00
```

Other applications may require different string representations of a date. The [DateTime.ToString\(String\)](#) method returns the string representation defined by a standard or custom format specifier using the formatting conventions of the current culture. The following example uses the [DateTime.ToString\(String\)](#) method to display the

full date and time pattern for the en-US culture, the current culture on the computer on which the example was run.

```
C#
```

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString("F"));
// Displays Saturday, March 01, 2008 7:00:00 AM
```

Finally, you can specify both the culture and the format using the [DateTime.ToString\(String, IFormatProvider\)](#) method. The following example uses the [DateTime.ToString\(String, IFormatProvider\)](#) method to display the full date and time pattern for the fr-FR culture.

```
C#
```

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString("F", new
System.Globalization.CultureInfo("fr-FR")));
// Displays samedi 1 mars 2008 07:00:00
```

The [DateTime.ToString\(String\)](#) overload can also be used with a custom format string to specify other formats. The following example shows how to format a string using the [ISO 8601](#) ↗ standard format often used for web services. The Iso 8601 format does not have a corresponding standard format string.

```
C#
```

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0, DateTimeKind.Utc);
Console.WriteLine(date1.ToString("yyyy-MM-ddTHH:mm:sszzz",
System.Globalization.CultureInfo.InvariantCulture));
// Displays 2008-03-01T07:00:00+00:00
```

For more information about formatting [DateTime](#) values, see [Standard Date and Time Format Strings](#) and [Custom Date and Time Format Strings](#).

## Parse DateTime values from strings

Parsing converts the string representation of a date and time to a [DateTime](#) value. Typically, date and time strings have two different usages in applications:

- A date and time takes a variety of forms and reflects the conventions of either the current culture or a specific culture. For example, an application allows a user whose current culture is en-US to input a date value as "12/15/2013" or

"December 15, 2013". It allows a user whose current culture is en-gb to input a date value as "15/12/2013" or "15 December 2013."

- A date and time is represented in a predefined format. For example, an application serializes a date as "20130103" independently of the culture on which the app is running. An application may require dates be input in the current culture's short date format.

You use the [Parse](#) or [TryParse](#) method to convert a string from one of the common date and time formats used by a culture to a [DateTime](#) value. The following example shows how you can use [TryParse](#) to convert date strings in different culture-specific formats to a [DateTime](#) value. It changes the current culture to English (United Kingdom) and calls the [GetDateTimeFormats\(\)](#) method to generate an array of date and time strings. It then passes each element in the array to the [TryParse](#) method. The output from the example shows the parsing method was able to successfully convert each of the culture-specific date and time strings.

C#

```
System.Threading.Thread.CurrentThread.CurrentCulture =
    System.Globalization.CultureInfo.CreateSpecificCulture("en-GB");

var date1 = new DateTime(2013, 6, 1, 12, 32, 30);
var badFormats = new List<String>();

Console.WriteLine("${"Date String",-37} {"Date",-19}\n");
foreach (var dateString in date1.GetDateTimeFormats())
{
    DateTime parsedDate;
    if (DateTime.TryParse(dateString, out parsedDate))
        Console.WriteLine($"{dateString,-37}
{DateTime.Parse(dateString),-19}");
    else
        badFormats.Add(dateString);
}

// Display strings that could not be parsed.
if (badFormats.Count > 0)
{
    Console.WriteLine("\nStrings that could not be parsed: ");
    foreach (var badFormat in badFormats)
        Console.WriteLine($"    {badFormat}");
}
// Press "Run" to see the output.
```

You use the [ParseExact](#) and [TryParseExact](#) methods to convert a string that must match a particular format or formats to a [DateTime](#) value. You specify one or more date and time format strings as a parameter to the parsing method. The following example uses the

`TryParseExact(String, String[], IFormatProvider, DateTimeStyles, DateTime)` method to convert strings that must be either in a "yyyyMMdd" format or a "HHmmss" format to `DateTime` values.

C#

```
string[] formats = { "yyyyMMdd", "HHmmss" };
string[] dateStrings = { "20130816", "20131608", " 20130816  ",
                        "115216", "521116", " 115216 " };
DateTime parsedDate;

foreach (var dateString in dateStrings)
{
    if (DateTime.TryParseExact(dateString, formats, null,
        System.Globalization.DateTimeStyles.AllowWhiteSpaces |
        System.Globalization.DateTimeStyles.AdjustToUniversal,
        out parsedDate))
        Console.WriteLine($"{dateString} --> {parsedDate:g}");
    else
        Console.WriteLine($"Cannot convert {dateString}");
}

// The example displays the following output:
//      20130816 --> 8/16/2013 12:00 AM
//      Cannot convert 20131608
//      20130816 --> 8/16/2013 12:00 AM
//      115216 --> 4/22/2013 11:52 AM
//      Cannot convert 521116
//      115216 --> 4/22/2013 11:52 AM
```

One common use for `ParseExact` is to convert a string representation from a web service, usually in [ISO 8601](#) standard format. The following code shows the correct format string to use:

C#

```
var iso8601String = "20080501T08:30:52Z";
DateTime dateISO8602 = DateTime.ParseExact(iso8601String,
    "yyyyMMddTHH:mm:ssZ",
    System.Globalization.CultureInfo.InvariantCulture);
Console.WriteLine($"{iso8601String} --> {dateISO8602:g}");
```

If a string cannot be parsed, the `Parse` and `ParseExact` methods throw an exception. The `TryParse` and `TryParseExact` methods return a `Boolean` value that indicates whether the conversion succeeded or failed. You should use the `TryParse` or `TryParseExact` methods in scenarios where performance is important. The parsing operation for date and time

strings tends to have a high failure rate, and exception handling is expensive. Use these methods if strings are input by users or coming from an unknown source.

For more information about parsing date and time values, see [Parsing Date and Time Strings](#).

## DateTime values

Descriptions of time values in the [DateTime](#) type are often expressed using the Coordinated Universal Time (UTC) standard. Coordinated Universal Time is the internationally recognized name for Greenwich Mean Time (GMT). Coordinated Universal Time is the time as measured at zero degrees longitude, the UTC origin point. Daylight saving time is not applicable to UTC.

Local time is relative to a particular time zone. A time zone is associated with a time zone offset. A time zone offset is the displacement of the time zone measured in hours from the UTC origin point. In addition, local time is optionally affected by daylight saving time, which adds or subtracts a time interval adjustment. Local time is calculated by adding the time zone offset to UTC and adjusting for daylight saving time if necessary. The time zone offset at the UTC origin point is zero.

UTC time is suitable for calculations, comparisons, and storing dates and time in files. Local time is appropriate for display in user interfaces of desktop applications. Time zone-aware applications (such as many Web applications) also need to work with a number of other time zones.

If the [Kind](#) property of a [DateTime](#) object is [DateTimeKind.Unspecified](#), it is unspecified whether the time represented is local time, UTC time, or a time in some other time zone.

## DateTime resolution

### Note

As an alternative to performing date and time arithmetic on [DateTime](#) values to measure elapsed time, you can use the [Stopwatch](#) class.

The [Ticks](#) property expresses date and time values in units of one ten-millionth of a second. The [Millisecond](#) property returns the thousandths of a second in a date and time value. Using repeated calls to the [DateTime.Now](#) property to measure elapsed time is dependent on the system clock. The system clock on Windows 7 and Windows 8

systems has a resolution of approximately 15 milliseconds. This resolution affects small time intervals less than 100 milliseconds.

The following example illustrates the dependence of current date and time values on the resolution of the system clock. In the example, an outer loop repeats 20 times, and an inner loop serves to delay the outer loop. If the value of the outer loop counter is 10, a call to the [Thread.Sleep](#) method introduces a five-millisecond delay. The following example shows the number of milliseconds returned by the [DateTime.NowMilliseconds](#) property changes only after the call to [Thread.Sleep](#).

C#

```
string output = "";
for (int ctr = 0; ctr <= 20; ctr++)
{
    output += String.Format("${DateTime.Now.Millisecond}\n");
    // Introduce a delay loop.
    for (int delay = 0; delay <= 1000; delay++)
    { }

    if (ctr == 10)
    {
        output += "Thread.Sleep called...\n";
        System.Threading.Thread.Sleep(5);
    }
}
Console.WriteLine(output);
// Press "Run" to see the output.
```

## DateTime operations

A calculation using a [DateTime](#) structure, such as [Add](#) or [Subtract](#), does not modify the value of the structure. Instead, the calculation returns a new [DateTime](#) structure whose value is the result of the calculation.

Conversion operations between time zones (such as between UTC and local time, or between one time zone and another) take daylight saving time into account, but arithmetic and comparison operations do not.

The [DateTime](#) structure itself offers limited support for converting from one time zone to another. You can use the [ToLocalTime](#) method to convert UTC to local time, or you can use the [ToUniversalTime](#) method to convert from local time to UTC. However, a full set of time zone conversion methods is available in the [TimeZoneInfo](#) class. You convert the time in any one of the world's time zones to the time in any other time zone using these methods.

Calculations and comparisons of [DateTime](#) objects are meaningful only if the objects represent times in the same time zone. You can use a [TimeZoneInfo](#) object to represent a [DateTime](#) value's time zone, although the two are loosely coupled. A [DateTime](#) object does not have a property that returns an object that represents that date and time value's time zone. The [Kind](#) property indicates if a [DateTime](#) represents UTC, local time, or is unspecified. In a time zone-aware application, you must rely on some external mechanism to determine the time zone in which a [DateTime](#) object was created. You could use a structure that wraps both the [DateTime](#) value and the [TimeZoneInfo](#) object that represents the [DateTime](#) value's time zone. For details on using UTC in calculations and comparisons with [DateTime](#) values, see [Performing Arithmetic Operations with Dates and Times](#).

Each [DateTime](#) member implicitly uses the Gregorian calendar to perform its operation. Exceptions are methods that implicitly specify a calendar. These include constructors that specify a calendar, and methods with a parameter derived from [IFormatProvider](#), such as [System.Globalization.DateTimeFormatInfo](#).

Operations by members of the [DateTime](#) type take into account details such as leap years and the number of days in a month.

## DateTime values and calendars

The .NET Class Library includes a number of calendar classes, all of which are derived from the [Calendar](#) class. They are:

- The [ChineseLunisolarCalendar](#) class.
- The [EastAsianLunisolarCalendar](#) class.
- The [GregorianCalendar](#) class.
- The [HebrewCalendar](#) class.
- The [HijriCalendar](#) class.
- The [JapaneseCalendar](#) class.
- The [JapaneseLunisolarCalendar](#) class.
- The [JulianCalendar](#) class.
- The [KoreanCalendar](#) class.
- The [KoreanLunisolarCalendar](#) class.
- The [PersianCalendar](#) class.
- The [TaiwanCalendar](#) class.
- The [TaiwanLunisolarCalendar](#) class.
- The [ThaiBuddhistCalendar](#) class.
- The [UmAlQuraCalendar](#) class.

## ⓘ Important

Eras in the Japanese calendars are based on the emperor's reign and are therefore expected to change. For example, May 1, 2019 marked the beginning of the Reiwa era in the [JapaneseCalendar](#) and [JapaneseLunisolarCalendar](#). Such a change of era affects all applications that use these calendars. For more information and to determine whether your applications are affected, see [Handling a new era in the Japanese calendar in .NET](#). For information on testing your applications on Windows systems to ensure their readiness for the era change, see [Prepare your application for the Japanese era change](#). For features in .NET that support calendars with multiple eras and for best practices when working with calendars that support multiple eras, see [Working with eras](#).

Each culture uses a default calendar defined by its read-only [CultureInfo.Calendar](#) property. Each culture may support one or more calendars defined by its read-only [CultureInfo.OptionalCalendars](#) property. The calendar currently used by a specific [CultureInfo](#) object is defined by its [DateTimeFormatInfo.Calendar](#) property. It must be one of the calendars found in the [CultureInfo.OptionalCalendars](#) array.

A culture's current calendar is used in all formatting operations for that culture. For example, the default calendar of the Thai Buddhist culture is the Thai Buddhist Era calendar, which is represented by the [ThaiBuddhistCalendar](#) class. When a [CultureInfo](#) object that represents the Thai Buddhist culture is used in a date and time formatting operation, the Thai Buddhist Era calendar is used by default. The Gregorian calendar is used only if the culture's [DateTimeFormatInfo.Calendar](#) property is changed, as the following example shows:

C#

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var value = new DateTime(2016, 5, 28);

Console.WriteLine(value.ToString(thTH));

thTH.DateTimeFormat.Calendar = new System.Globalization.GregorianCalendar();
Console.WriteLine(value.ToString(thTH));
// The example displays the following output:
//      28/5/2559 0:00:00
//      28/5/2016 0:00:00
```

A culture's current calendar is also used in all parsing operations for that culture, as the following example shows.

C#

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var value = DateTime.Parse("28/05/2559", thTH);
Console.WriteLine(value.ToString(thTH));

thTH.DateTimeFormat.Calendar = new System.Globalization.GregorianCalendar();
Console.WriteLine(value.ToString(thTH));
// The example displays the following output:
//      28/5/2559 0:00:00
//      28/5/2016 0:00:00
```

You instantiate a [DateTime](#) value using the date and time elements (number of the year, month, and day) of a specific calendar by calling a [DateTime constructor](#) that includes a `calendar` parameter and passing it a [Calendar](#) object that represents that calendar. The following example uses the date and time elements from the [ThaiBuddhistCalendar](#) calendar.

C#

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var dat = new DateTime(2559, 5, 28, thTH.DateTimeFormat.Calendar);
Console.WriteLine($"Thai Buddhist era date: {dat.ToString("d", thTH)}");
Console.WriteLine($"Gregorian date: {dat:d}");
// The example displays the following output:
//      Thai Buddhist Era Date: 28/5/2559
//      Gregorian Date: 28/05/2016
```

[DateTime](#) constructors that do not include a `calendar` parameter assume that the date and time elements are expressed as units in the Gregorian calendar.

All other [DateTime](#) properties and methods use the Gregorian calendar. For example, the [DateTime.Year](#) property returns the year in the Gregorian calendar, and the [DateTime.IsLeapYear\(Int32\)](#) method assumes that the `year` parameter is a year in the Gregorian calendar. Each [DateTime](#) member that uses the Gregorian calendar has a corresponding member of the [Calendar](#) class that uses a specific calendar. For example, the [Calendar.GetYear](#) method returns the year in a specific calendar, and the [Calendar.IsLeapYear](#) method interprets the `year` parameter as a year number in a specific calendar. The following example uses both the [DateTime](#) and the corresponding members of the [ThaiBuddhistCalendar](#) class.

C#

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var cal = thTH.DateTimeFormat.Calendar;
var dat = new DateTime(2559, 5, 28, cal);
```

```

Console.WriteLine("Using the Thai Buddhist Era calendar:");
Console.WriteLine($"Date: {dat.ToString("d", thTH)}");
Console.WriteLine($"Year: {cal.GetYear(dat)}");
Console.WriteLine($"Leap year: {cal.IsLeapYear(cal.GetYear(dat))}\n");

Console.WriteLine("Using the Gregorian calendar:");
Console.WriteLine($"Date: {dat:d}");
Console.WriteLine($"Year: {dat.Year}");
Console.WriteLine($"Leap year: {DateTime.IsLeapYear(dat.Year)}");
// The example displays the following output:
//      Using the Thai Buddhist Era calendar
//      Date : 28/5/2559
//      Year: 2559
//      Leap year : True
//
//      Using the Gregorian calendar
//      Date : 28/05/2016
//      Year: 2016
//      Leap year : True

```

The [DateTime](#) structure includes a [DayOfWeek](#) property that returns the day of the week in the Gregorian calendar. It does not include a member that allows you to retrieve the week number of the year. To retrieve the week of the year, call the individual calendar's [Calendar.GetWeekOfYear](#) method. The following example provides an illustration.

C#

```

var thTH = new System.Globalization.CultureInfo("th-TH");
var thCalendar = thTH.DateTimeFormat.Calendar;
var dat = new DateTime(1395, 8, 18, thCalendar);
Console.WriteLine("Using the Thai Buddhist Era calendar:");
Console.WriteLine($"Date: {dat.ToString("d", thTH)}");
Console.WriteLine($"Day of Week: {thCalendar.GetDayOfWeek(dat)}");
Console.WriteLine($"Week of year: {thCalendar.GetWeekOfYear(dat,
System.Globalization.CalendarWeekRule.FirstDay, DayOfWeek.Sunday)}\n");

var greg = new System.Globalization.GregorianCalendar();
Console.WriteLine("Using the Gregorian calendar:");
Console.WriteLine($"Date: {dat:d}");
Console.WriteLine($"Day of Week: {dat.DayOfWeek}");
Console.WriteLine($"Week of year: {greg.GetWeekOfYear(dat,
System.Globalization.CalendarWeekRule.FirstDay, DayOfWeek.Sunday)}");
// The example displays the following output:
//      Using the Thai Buddhist Era calendar
//      Date : 18/8/1395
//      Day of Week: Sunday
//      Week of year: 34
//
//      Using the Gregorian calendar
//      Date : 18/08/0852

```

```
//      Day of Week: Sunday
//      Week of year: 34
```

For more information on dates and calendars, see [Working with Calendars](#).

## Persist DateTime values

You can persist [DateTime](#) values in the following ways:

- [Convert them to strings](#) and persist the strings.
- [Convert them to 64-bit integer values](#) (the value of the [Ticks](#) property) and persist the integers.
- [Serialize the DateTime values](#).

You must ensure that the routine that restores the [DateTime](#) values doesn't lose data or throw an exception regardless of which technique you choose. [DateTime](#) values should round-trip. That is, the original value and the restored value should be the same. And if the original [DateTime](#) value represents a single instant of time, it should identify the same moment of time when it's restored.

## Persist values as strings

To successfully restore [DateTime](#) values that are persisted as strings, follow these rules:

- Make the same assumptions about culture-specific formatting when you restore the string as when you persisted it. To ensure that a string can be restored on a system whose current culture is different from the culture of the system it was saved on, call the [ToString](#) overload to save the string by using the conventions of the invariant culture. Call the [Parse\(String, IFormatProvider, DateTimeStyles\)](#) or [TryParse\(String, IFormatProvider, DateTimeStyles, DateTime\)](#) overload to restore the string by using the conventions of the invariant culture. Never use the [ToString\(\)](#), [Parse\(String\)](#), or [TryParse\(String, DateTime\)](#) overloads, which use the conventions of the current culture.
- If the date represents a single moment of time, ensure that it represents the same moment in time when it's restored, even on a different time zone. Convert the [DateTime](#) value to Coordinated Universal Time (UTC) before saving it or use [DateTimeOffset](#).

The most common error made when persisting [DateTime](#) values as strings is to rely on the formatting conventions of the default or current culture. Problems arise if the current culture is different when saving and restoring the strings. The following example

illustrates these problems. It saves five dates using the formatting conventions of the current culture, which in this case is English (United States). It restores the dates using the formatting conventions of a different culture, which in this case is English (United Kingdom). Because the formatting conventions of the two cultures are different, two of the dates can't be restored, and the remaining three dates are interpreted incorrectly. Also, if the original date and time values represent single moments in time, the restored times are incorrect because time zone information is lost.

C#

```
public static void PersistAsLocalStrings()
{
    SaveLocalDatesAsString();
    RestoreLocalDatesFromString();
}

private static void SaveLocalDatesAsString()
{
    DateTime[] dates = { new DateTime(2014, 6, 14, 6, 32, 0),
                        new DateTime(2014, 7, 10, 23, 49, 0),
                        new DateTime(2015, 1, 10, 1, 16, 0),
                        new DateTime(2014, 12, 20, 21, 45, 0),
                        new DateTime(2014, 6, 2, 15, 14, 0) };
    string? output = null;

    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    for (int ctr = 0; ctr < dates.Length; ctr++)
    {
        Console.WriteLine(dates[ctr].ToString("f"));
        output += dates[ctr].ToString() + (ctr != dates.Length - 1 ? " | " :
"");
    }
    var sw = new StreamWriter(filenameTxt);
    sw.Write(output);
    sw.Close();
    Console.WriteLine("Saved dates...");
}

private static void RestoreLocalDatesFromString()
{
    TimeZoneInfo.ClearCachedData();
    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
    StreamReader sr = new StreamReader(filenameTxt);
    string[] inputValues = sr.ReadToEnd().Split(new char[] { ' | ' },
StringSplitOptions.RemoveEmptyEntries);
}
```

```

sr.Close();
Console.WriteLine("The dates on an {0} system:",
                  Thread.CurrentThread.CurrentCulture.Name);
foreach (var inputValue in inputValues)
{
    DateTime dateValue;
    if (DateTime.TryParse(inputValue, out dateValue))
    {
        Console.WriteLine($"{inputValue} --> {dateValue:f}");
    }
    else
    {
        Console.WriteLine($"Cannot parse '{inputValue}'");
    }
}
Console.WriteLine("Restored dates...");
}

// When saved on an en-US system, the example displays the following output:
//   Current Time Zone: (UTC-08:00) Pacific Time (US & Canada)
//   The dates on an en-US system:
//   Saturday, June 14, 2014 6:32 AM
//   Thursday, July 10, 2014 11:49 PM
//   Saturday, January 10, 2015 1:16 AM
//   Saturday, December 20, 2014 9:45 PM
//   Monday, June 02, 2014 3:14 PM
//   Saved dates...
//
// When restored on an en-GB system, the example displays the following
// output:
//   Current Time Zone: (UTC) Dublin, Edinburgh, Lisbon, London
//   The dates on an en-GB system:
//   Cannot parse //6/14/2014 6:32:00 AM//  

//   //7/10/2014 11:49:00 PM// --> 07 October 2014 23:49
//   //1/10/2015 1:16:00 AM// --> 01 October 2015 01:16
//   Cannot parse //12/20/2014 9:45:00 PM//  

//   //6/2/2014 3:14:00 PM// --> 06 February 2014 15:14
//   Restored dates...

```

To round-trip [DateTime](#) values successfully, follow these steps:

1. If the values represent single moments of time, convert them from the local time to UTC by calling the [ToUniversalTime](#) method.
2. Convert the dates to their string representations by calling the [ToString\(String, IFormatProvider\)](#) or [String.Format\(IFormatProvider, String, Object\[\]\)](#) overload. Use the formatting conventions of the invariant culture by specifying [CultureInfo.InvariantCulture](#) as the `provider` argument. Specify that the value should round-trip by using the "O" or "R" standard format string.

To restore the persisted [DateTime](#) values without data loss, follow these steps:

1. Parse the data by calling the [ParseExact](#) or [TryParseExact](#) overload. Specify `CultureInfo.InvariantCulture` as the `provider` argument, and use the same standard format string you used for the `format` argument during conversion. Include the `DateTimeStyles.RoundtripKind` value in the `styles` argument.
2. If the `DateTime` values represent single moments in time, call the [ToLocalTime](#) method to convert the parsed date from UTC to local time.

The following example uses the invariant culture and the "O" standard format string to ensure that `DateTime` values saved and restored represent the same moment in time regardless of the system, culture, or time zone of the source and target systems.

C#

```
public static void PersistAsInvariantStrings()
{
    SaveDatesAsInvariantStrings();
    RestoreDatesAsInvariantStrings();
}

private static void SaveDatesAsInvariantStrings()
{
    DateTime[] dates = { new DateTime(2014, 6, 14, 6, 32, 0),
                         new DateTime(2014, 7, 10, 23, 49, 0),
                         new DateTime(2015, 1, 10, 1, 16, 0),
                         new DateTime(2014, 12, 20, 21, 45, 0),
                         new DateTime(2014, 6, 2, 15, 14, 0) };
    string? output = null;

    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    for (int ctr = 0; ctr < dates.Length; ctr++)
    {
        Console.WriteLine(dates[ctr].ToString("f"));
        output += dates[ctr].ToUniversalTime().ToString("O",
CultureInfo.InvariantCulture)
            + (ctr != dates.Length - 1 ? " | " : ""));
    }
    var sw = new StreamWriter(filenameTxt);
    sw.Write(output);
    sw.Close();
    Console.WriteLine("Saved dates...");
}

private static void RestoreDatesAsInvariantStrings()
{
    TimeZoneInfo.ClearCachedData();
    Console.WriteLine("Current Time Zone: {0}",
                      TimeZoneInfo.Local.DisplayName);
    Thread.CurrentThread.CurrentCulture =

```

```

CultureInfo.CreateSpecificCulture("en-GB");
StreamReader sr = new StreamReader(filenameTxt);
string[] inputValues = sr.ReadToEnd().Split(new char[] { '|' });

StringSplitOptions.RemoveEmptyEntries);
sr.Close();
Console.WriteLine("The dates on an {0} system:",
                  Thread.CurrentThread.CurrentCulture.Name);
foreach (var inputValue in inputValues)
{
    DateTime dateValue;
    if (DateTime.TryParseExact(inputValue, "0",
CultureInfo.InvariantCulture,
DateTimeStyles.RoundtripKind, out dateValue))
    {
        Console.WriteLine($"'{inputValue}' -->
{dateValue.ToLocalTime():f}");
    }
    else
    {
        Console.WriteLine("Cannot parse '{0}'", inputValue);
    }
}
Console.WriteLine("Restored dates...");
}

// When saved on an en-US system, the example displays the following output:
//     Current Time Zone: (UTC-08:00) Pacific Time (US & Canada)
//     The dates on an en-US system:
//     Saturday, June 14, 2014 6:32 AM
//     Thursday, July 10, 2014 11:49 PM
//     Saturday, January 10, 2015 1:16 AM
//     Saturday, December 20, 2014 9:45 PM
//     Monday, June 02, 2014 3:14 PM
//     Saved dates...
//
// When restored on an en-GB system, the example displays the following
output:
//     Current Time Zone: (UTC) Dublin, Edinburgh, Lisbon, London
//     The dates on an en-GB system:
//     '2014-06-14T13:32:00.0000000Z' --> 14 June 2014 14:32
//     '2014-07-11T06:49:00.0000000Z' --> 11 July 2014 07:49
//     '2015-01-10T09:16:00.0000000Z' --> 10 January 2015 09:16
//     '2014-12-21T05:45:00.0000000Z' --> 21 December 2014 05:45
//     '2014-06-02T22:14:00.0000000Z' --> 02 June 2014 23:14
//     Restored dates...

```

## Persist values as integers

You can persist a date and time as an [Int64](#) value that represents a number of ticks. In this case, you don't have to consider the culture of the systems the [DateTime](#) values are persisted and restored on.

To persist a [DateTime](#) value as an integer:

1. If the [DateTime](#) values represent single moments in time, convert them to UTC by calling the [ToUniversalTime](#) method.
2. Retrieve the number of ticks represented by the [DateTime](#) value from its [Ticks](#) property.

To restore a [DateTime](#) value that has been persisted as an integer:

1. Instantiate a new [DateTime](#) object by passing the [Int64](#) value to the [DateTime\(Int64\)](#) constructor.
2. If the [DateTime](#) value represents a single moment in time, convert it from UTC to the local time by calling the [ToLocalTime](#) method.

The following example persists an array of [DateTime](#) values as integers on a system in the U.S. Pacific Time zone. It restores it on a system in the UTC zone. The file that contains the integers includes an [Int32](#) value that indicates the total number of [Int64](#) values that immediately follow it.

C#

```
public static void PersistAsIntegers()
{
    SaveDatesAsInts();
    RestoreDatesAsInts();
}

private static void SaveDatesAsInts()
{
    DateTime[] dates = { new DateTime(2014, 6, 14, 6, 32, 0),
                        new DateTime(2014, 7, 10, 23, 49, 0),
                        new DateTime(2015, 1, 10, 1, 16, 0),
                        new DateTime(2014, 12, 20, 21, 45, 0),
                        new DateTime(2014, 6, 2, 15, 14, 0) };

    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    var ticks = new long[dates.Length];
    for (int ctr = 0; ctr < dates.Length; ctr++)
    {
        Console.WriteLine(dates[ctr].ToString("f"));
        ticks[ctr] = dates[ctr].ToUniversalTime().Ticks;
    }
    var fs = new FileStream(filenameInts, FileMode.Create);
    var bw = new BinaryWriter(fs);
    bw.Write(ticks.Length);
    foreach (var tick in ticks)
        bw.Write(tick);
```

```
        bw.Close();
        Console.WriteLine("Saved dates...");
    }

private static void RestoreDatesAsInts()
{
    TimeZoneInfo.ClearCachedData();
    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
    FileStream fs = new FileStream(filenameInts, FileMode.Open);
    BinaryReader br = new BinaryReader(fs);
    int items;
    DateTime[] dates;

    try
    {
        items = br.ReadInt32();
        dates = new DateTime[items];

        for (int ctr = 0; ctr < items; ctr++)
        {
            long ticks = br.ReadInt64();
            dates[ctr] = new DateTime(ticks).ToLocalTime();
        }
    }
    catch (EndOfStreamException)
    {
        Console.WriteLine("File corruption detected. Unable to restore
data...");
        return;
    }
    catch (IOException)
    {
        Console.WriteLine("Unspecified I/O error. Unable to restore
data...");
        return;
    }
    // Thrown during array initialization.
    catch (OutOfMemoryException)
    {
        Console.WriteLine("File corruption detected. Unable to restore
data...");
        return;
    }
    finally
    {
        br.Close();
    }

    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    foreach (var value in dates)
```

```

        Console.WriteLine(value.ToString("f"));

        Console.WriteLine("Restored dates...");
    }
// When saved on an en-US system, the example displays the following output:
//     Current Time Zone: (UTC-08:00) Pacific Time (US & Canada)
//     The dates on an en-US system:
//     Saturday, June 14, 2014 6:32 AM
//     Thursday, July 10, 2014 11:49 PM
//     Saturday, January 10, 2015 1:16 AM
//     Saturday, December 20, 2014 9:45 PM
//     Monday, June 02, 2014 3:14 PM
//     Saved dates...
//
// When restored on an en-GB system, the example displays the following
output:
//     Current Time Zone: (UTC) Dublin, Edinburgh, Lisbon, London
//     The dates on an en-GB system:
//     14 June 2014 14:32
//     11 July 2014 07:49
//     10 January 2015 09:16
//     21 December 2014 05:45
//     02 June 2014 23:14
//     Restored dates...

```

## Serialize DateTime values

You can persist [DateTime](#) values through serialization to a stream or file, and then restore them through deserialization. [DateTime](#) data is serialized in some specified object format. The objects are restored when they are deserialized. A formatter or serializer, such as [JsonSerializer](#) or [XmlSerializer](#), handles the process of serialization and deserialization. For more information about serialization and the types of serialization supported by .NET, see [Serialization](#).

The following example uses the [XmlSerializer](#) class to serialize and deserialize [DateTime](#) values. The values represent all leap year days in the twenty-first century. The output represents the result if the example is run on a system whose current culture is English (United Kingdom). Because you've deserialized the [DateTime](#) object itself, the code doesn't have to handle cultural differences in date and time formats.

C#

```

public static void PersistAsXML()
{
    // Serialize the data.
    var leapYears = new List<DateTime>();
    for (int year = 2000; year <= 2100; year += 4)
    {
        if (DateTime.IsLeapYear(year))

```

```

        leapYears.Add(new DateTime(year, 2, 29));
    }
    DateTime[] dateArray = leapYears.ToArray();

    var serializer = new XmlSerializer(dateArray.GetType());
    TextWriter sw = new StreamWriter(filenameXml);

    try
    {
        serializer.Serialize(sw, dateArray);
    }
    catch (InvalidOperationException e)
    {
        Console.WriteLine(e.InnerException?.Message);
    }
    finally
    {
        if (sw != null) sw.Close();
    }

    // Deserialize the data.
    DateTime[]? deserializedDates;
    using (var fs = new FileStream(filenameXml, FileMode.Open))
    {
        deserializedDates = (DateTime[]?)serializer.Deserialize(fs);
    }

    // Display the dates.
    Console.WriteLine($"Leap year days from 2000-2100 on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    int nItems = 0;
    if (deserializedDates is not null)
    {
        foreach (var dat in deserializedDates)
        {
            Console.Write($" {dat:d} ");
            nItems++;
            if (nItems % 5 == 0)
                Console.WriteLine();
        }
    }
}

// The example displays the following output:
//    Leap year days from 2000-2100 on an en-GB system:
//    29/02/2000    29/02/2004    29/02/2008    29/02/2012
29/02/2016
//    29/02/2020    29/02/2024    29/02/2028    29/02/2032
29/02/2036
//    29/02/2040    29/02/2044    29/02/2048    29/02/2052
29/02/2056
//    29/02/2060    29/02/2064    29/02/2068    29/02/2072
29/02/2076
//    29/02/2080    29/02/2084    29/02/2088    29/02/2092
29/02/2096

```

The previous example doesn't include time information. If a `DateTime` value represents a moment in time and is expressed as a local time, convert it from local time to UTC before serializing it by calling the `ToUniversalTime` method. After you deserialize it, convert it from UTC to local time by calling the `ToLocalTime` method.

## DateTime vs. TimeSpan

The `DateTime` and `TimeSpan` value types differ in that a `DateTime` represents an instant in time whereas a `TimeSpan` represents a time interval. You can subtract one instance of `DateTime` from another to obtain a `TimeSpan` object that represents the time interval between them. Or you could add a positive `TimeSpan` to the current `DateTime` to obtain a `DateTime` value that represents a future date.

You can add or subtract a time interval from a `DateTime` object. Time intervals can be negative or positive, and they can be expressed in units such as ticks, seconds, or as a `TimeSpan` object.

## Compare for equality within tolerance

Equality comparisons for `DateTime` values are exact. To be considered equal, two values must be expressed as the same number of ticks. That precision is often unnecessary or even incorrect for many applications. Often, you want to test if `DateTime` objects are **roughly equal**.

The following example demonstrates how to compare roughly equivalent `DateTime` values. It accepts a small margin of difference when declaring them equal.

C#

```
public static bool RoughlyEquals(DateTime time, DateTime timeWithWindow, int
windowInSeconds, int frequencyInSeconds)
{
    long delta = (long)((TimeSpan)(timeWithWindow - time)).TotalSeconds %
frequencyInSeconds;
    delta = delta > windowInSeconds ? frequencyInSeconds - delta : delta;
    return Math.Abs(delta) < windowInSeconds;
}

public static void TestRoughlyEquals()
{
    int window = 10;
    int freq = 60 * 60 * 2; // 2 hours;

    DateTime d1 = DateTime.Now;
```

```

DateTime d2 = d1.AddSeconds(2 * window);
DateTime d3 = d1.AddSeconds(-2 * window);
DateTime d4 = d1.AddSeconds(window / 2);
DateTime d5 = d1.AddSeconds(-window / 2);

DateTime d6 = (d1.AddHours(2)).AddSeconds(2 * window);
DateTime d7 = (d1.AddHours(2)).AddSeconds(-2 * window);
DateTime d8 = (d1.AddHours(2)).AddSeconds(window / 2);
DateTime d9 = (d1.AddHours(2)).AddSeconds(-window / 2);

Console.WriteLine($"d1 ({d1}) ~= d1 ({d1}): {RoughlyEquals(d1, d1, window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d2 ({d2}): {RoughlyEquals(d1, d2, window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d3 ({d3}): {RoughlyEquals(d1, d3, window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d4 ({d4}): {RoughlyEquals(d1, d4, window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d5 ({d5}): {RoughlyEquals(d1, d5, window, freq)}");

Console.WriteLine($"d1 ({d1}) ~= d6 ({d6}): {RoughlyEquals(d1, d6, window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d7 ({d7}): {RoughlyEquals(d1, d7, window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d8 ({d8}): {RoughlyEquals(d1, d8, window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d9 ({d9}): {RoughlyEquals(d1, d9, window, freq)}");
}

// The example displays output similar to the following:
//      d1 (1/28/2010 9:01:26 PM) ~= d1 (1/28/2010 9:01:26 PM): True
//      d1 (1/28/2010 9:01:26 PM) ~= d2 (1/28/2010 9:01:46 PM): False
//      d1 (1/28/2010 9:01:26 PM) ~= d3 (1/28/2010 9:01:06 PM): False
//      d1 (1/28/2010 9:01:26 PM) ~= d4 (1/28/2010 9:01:31 PM): True
//      d1 (1/28/2010 9:01:26 PM) ~= d5 (1/28/2010 9:01:21 PM): True
//      d1 (1/28/2010 9:01:26 PM) ~= d6 (1/28/2010 11:01:46 PM): False
//      d1 (1/28/2010 9:01:26 PM) ~= d7 (1/28/2010 11:01:06 PM): False
//      d1 (1/28/2010 9:01:26 PM) ~= d8 (1/28/2010 11:01:31 PM): True
//      d1 (1/28/2010 9:01:26 PM) ~= d9 (1/28/2010 11:01:21 PM): True

```

## COM interop considerations

A [DateTime](#) value that is transferred to a COM application, then is transferred back to a managed application, is said to round-trip. However, a [DateTime](#) value that specifies only a time does not round-trip as you might expect.

If you round-trip only a time, such as 3 P.M., the final date and time is December 30, 1899 C.E. at 3:00 P.M., instead of January, 1, 0001 C.E. at 3:00 P.M. .NET and COM assume a default date when only a time is specified. However, the COM system assumes

a base date of December 30, 1899 C.E., while .NET assumes a base date of January, 1, 0001 C.E.

When only a time is passed from .NET to COM, special processing is performed that converts the time to the format used by COM. When only a time is passed from COM to .NET, no special processing is performed because that would corrupt legitimate dates and times on or before December 30, 1899. If a date starts its round-trip from COM, .NET and COM preserve the date.

The behavior of .NET and COM means that if your application round-trips a [DateTime](#) that only specifies a time, your application must remember to modify or ignore the erroneous date from the final [DateTime](#) object.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

## [.NET feedback](#)

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.DateTime.ToBinary and FromBinary methods

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

Use the [ToBinary](#) method to convert the value of the current [DateTime](#) object to a binary value. Subsequently, use the binary value and the [FromBinary](#) method to recreate the original [DateTime](#) object.

## ⓘ Important

In some cases, the [DateTime](#) value returned by the [FromBinary](#) method is not identical to the original [DateTime](#) value supplied to the [ToBinary](#) method. For more information, see the next section, "Local Time Adjustment".

A [DateTime](#) structure consists of a private [Kind](#) field, which indicates whether the specified time value is based on local time, Coordinated Universal Time (UTC), or neither, concatenated to a private [Ticks](#) field, which contains the number of 100-nanosecond ticks that specify a date and time.

## Local time adjustment

A local time, which is a Coordinated Universal Time adjusted to the local time zone, is represented by a [DateTime](#) structure whose [Kind](#) property has the value [Local](#). When restoring a local [DateTime](#) value from the binary representation that is produced by the [ToBinary](#) method, the [FromBinary](#) method may adjust the recreated value so that it is not equal to the original value. This can occur under the following conditions:

- If a local [DateTime](#) object is serialized in one time zone by the [ToBinary](#) method, and then deserialized in a different time zone by the [FromBinary](#) method, the local time represented by the resulting [DateTime](#) object is automatically adjusted to the second time zone.

For example, consider a [DateTime](#) object that represents a local time of 3 P.M. An application that is executing in the U.S. Pacific Time zone uses the [ToBinary](#) method to convert that [DateTime](#) object to a binary value. Another application that is executing in the U.S. Eastern Time zone then uses the [FromBinary](#) method to

convert the binary value to a new [DateTime](#) object. The value of the new [DateTime](#) object is 6 P.M., which represents the same point in time as the original 3 P.M. value, but is adjusted to local time in the Eastern Time zone.

- If the binary representation of a local [DateTime](#) value represents an invalid time in the local time zone of the system on which [FromBinary](#) is called, the time is adjusted so that it is valid.

For example, the transition from standard time to daylight saving time occurs in the Pacific Time zone of the United States on March 14, 2010, at 2:00 A.M., when the time advances by one hour, to 3:00 A.M. This hour interval is an invalid time, that is, a time interval that does not exist in this time zone. The following example shows that when a time that falls within this range is converted to a binary value by the [ToBinary](#) method and is then restored by the [FromBinary](#) method, the original value is adjusted to become a valid time. You can determine whether a particular date and time value may be subject to modification by passing it to the [TimeZoneInfo.IsInvalidTime](#) method, as the example illustrates.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        DateTime localDate = new DateTime(2010, 3, 14, 2, 30, 0,
DateTimeKind.Local);
        long binLocal = localDate.ToBinary();
        if (TimeZoneInfo.Local.IsInvalidTime(localDate))
            Console.WriteLine("{0} is an invalid time in the {1} zone.",
                            localDate,
                            TimeZoneInfo.Local.StandardName);

        DateTime localDate2 = DateTime.FromBinary(binLocal);
        Console.WriteLine("{0} = {1}: {2}",
                            localDate, localDate2,
                            localDate.Equals(localDate2));
    }
}
// The example displays the following output:
// 3/14/2010 2:30:00 AM is an invalid time in the Pacific Standard
Time zone.
// 3/14/2010 2:30:00 AM = 3/14/2010 3:30:00 AM: False
```

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.DateTime.TryParse method

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

The [DateTime.TryParse\(String, IFormatProvider, DateTimeStyles, DateTime\)](#) method parses a string that can contain date, time, and time zone information. It is similar to the [DateTime.Parse\(String, IFormatProvider, DateTimeStyles\)](#) method, except that the [DateTime.TryParse\(String, DateTime\)](#) method does not throw an exception if the conversion fails.

This method attempts to ignore unrecognized data and parse the input string (`s`) completely. If `s` contains a time but no date, the method by default substitutes the current date or, if `styles` includes the [NoCurrentDateDefault](#) flag, it substitutes [DateTime.Date.MinValue](#). If `s` contains a date but no time, 12:00 midnight is used as the default time. If a date is present but its year component consists of only two digits, it is converted to a year in the `provider` parameter's current calendar based on the value of the [Calendar.TwoDigitYearMax](#) property. Any leading, inner, or trailing white space characters in `s` are ignored. The date and time can be bracketed with a pair of leading and trailing NUMBER SIGN characters ('#', U+0023), and can be trailed with one or more NULL characters (U+0000).

Specific valid formats for date and time elements, as well as the names and symbols used in dates and times, are defined by the `provider` parameter, which can be any of the following:

- A [CultureInfo](#) object that represents the culture whose formatting is used in the `s` parameter. The [DateTimeFormatInfo](#) object returned by the `CultureInfo.DateTimeFormat` property defines the formatting used in `s`.
- A [DateTimeFormatInfo](#) object that defines the formatting used in `s`.
- A custom [IFormatProvider](#) implementation. Its [IFormatProvider.GetFormat](#) method returns a [DateTimeFormatInfo](#) object that defines the formatting used in `s`.

If `provider` is `null`, the current culture is used.

If `s` is the string representation of a leap day in a leap year in the current calendar, the method parses `s` successfully. If `s` is the string representation of a leap day in a non-leap year in the current calendar of `provider`, the parse operation fails and the method returns `false`.

The `styles` parameter defines the precise interpretation of the parsed string and how the parse operation should handle it. It can be one or more members of the [DateTimeStyles](#) enumeration, as described in the following table.

[+] [Expand table](#)

<b>DateTimeStyles</b>	<b>Description</b>
<b>member</b>	
<a href="#">AdjustToUniversal</a>	Parses <code>s</code> and, if necessary, converts it to UTC. If <code>s</code> includes a time zone offset, or if <code>s</code> contains no time zone information but <code>styles</code> includes the <a href="#">DateTimeStyles.AssumeLocal</a> flag, the method parses the string, calls <code>ToUniversalTime</code> to convert the returned <a href="#">DateTime</a> value to UTC, and sets the <code>Kind</code> property to <a href="#">DateTimeKind.Utc</a> . If <code>s</code> indicates that it represents UTC, or if <code>s</code> does not contain time zone information but <code>styles</code> includes the <a href="#">DateTimeStyles.AssumeUniversal</a> flag, the method parses the string, performs no time zone conversion on the returned <a href="#">DateTime</a> value, and sets the <code>Kind</code> property to <a href="#">DateTimeKind.Utc</a> . In all other cases, the flag has no effect.
<a href="#">AllowInnerWhite</a>	Although valid, this value is ignored. Inner white space is permitted in the date and time elements of <code>s</code> .
<a href="#">AllowLeadingWhite</a>	Although valid, this value is ignored. Leading white space is permitted in the date and time elements of <code>s</code> .
<a href="#">AllowTrailingWhite</a>	Although valid, this value is ignored. Trailing white space is permitted in the date and time elements of <code>s</code> .
<a href="#">AllowWhiteSpaces</a>	Specifies that <code>s</code> may contain leading, inner, and trailing white spaces. This is the default behavior. It cannot be overridden by supplying a more restrictive <a href="#">DateTimeStyles</a> enumeration value such as <a href="#">DateTimeStyles.None</a> .
<a href="#">AssumeLocal</a>	Specifies that if <code>s</code> lacks any time zone information, it is assumed to represent a local time. Unless the <a href="#">DateTimeStyles.AdjustToUniversal</a> flag is present, the <code>Kind</code> property of the returned <a href="#">DateTime</a> value is set to <a href="#">DateTimeKind.Local</a> .
<a href="#">AssumeUniversal</a>	Specifies that if <code>s</code> lacks any time zone information, it is assumed to represent UTC. Unless the <a href="#">DateTimeStyles.AdjustToUniversal</a> flag is present, the method converts the returned <a href="#">DateTime</a> value from UTC to local time and sets its <code>Kind</code> property to <a href="#">DateTimeKind.Local</a> .
<a href="#">None</a>	Although valid, this value is ignored.
<a href="#">RoundtripKind</a>	For strings that contain time zone information, tries to prevent the conversion of a date and time string to a <a href="#">DateTime</a> value with its <code>Kind</code> property set to <a href="#">DateTimeKind.Local</a> . Typically, such a string is created by

DateTimeStyles member	Description
	calling the <a href="#">DateTime.ToString(String)</a> method using either the "o", "r", or "u" standard format specifiers.

If `s` contains no time zone information, the [DateTime.TryParse\(String, IFormatProvider, DateTimeStyles, DateTime\)](#) method returns a [DateTime](#) value whose [Kind](#) property is [DateTimeKind.Unspecified](#) unless a `styles` flag indicates otherwise. If `s` includes time zone or time zone offset information, the [DateTime.TryParse\(String, IFormatProvider, DateTimeStyles, DateTime\)](#) method performs any necessary time conversion and returns one of the following:

- A [DateTime](#) value whose date and time reflect the local time and whose [Kind](#) property is [DateTimeKind.Local](#).
- Or, if `styles` includes the [AdjustToUniversal](#) flag, a [DateTime](#) value whose date and time reflect UTC and whose [Kind](#) property is [DateTimeKind.Utc](#).

This behavior can be overridden by using the [DateTimeStyles.RoundtripKind](#) flag.

## Parse custom cultures

If you parse a date and time string generated for a custom culture, use the [TryParseExact](#) method instead of the [TryParse](#) method to improve the probability that the parse operation will succeed. A custom culture date and time string can be complicated and difficult to parse. The [TryParse](#) method attempts to parse a string with several implicit parse patterns, all of which might fail. In contrast, the [TryParseExact](#) method requires you to explicitly designate one or more exact parse patterns that are likely to succeed.

For more information about custom cultures, see the [System.Globalization.CultureAndRegionInfoBuilder](#) class.

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.TimeSpan struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

A [TimeSpan](#) object represents a time interval (duration of time or elapsed time) that is measured as a positive or negative number of days, hours, minutes, seconds, and fractions of a second. The [TimeSpan](#) structure can also be used to represent the time of day, but only if the time is unrelated to a particular date. Otherwise, the [DateTime](#) or [DateTimeOffset](#) structure should be used instead. (For more information about using the [TimeSpan](#) structure to reflect the time of day, see [Choosing Between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo](#).)

## ⓘ Note

A [TimeSpan](#) value represents a time interval and can be expressed as a particular number of days, hours, minutes, seconds, and milliseconds. Because it represents a general interval without reference to a particular start or end point, it cannot be expressed in terms of years and months, both of which have a variable number of days. It differs from a [DateTime](#) value, which represents a date and time without reference to a particular time zone, or a [DateTimeOffset](#) value, which represents a specific moment of time.

The largest unit of time that the [TimeSpan](#) structure uses to measure duration is a day. Time intervals are measured in days for consistency, because the number of days in larger units of time, such as months and years, varies.

The value of a [TimeSpan](#) object is the number of ticks that equal the represented time interval. A tick is equal to 100 nanoseconds, or one ten-millionth of a second. The value of a [TimeSpan](#) object can range from [TimeSpan.MinValue](#) to [TimeSpan.MaxValue](#).

## Instantiate a TimeSpan value

You can instantiate a [TimeSpan](#) value in a number of ways:

- By calling its implicit parameterless constructor. This creates an object whose value is [TimeSpan.Zero](#), as the following example shows.

C#

```
TimeSpan interval = new TimeSpan();
Console.WriteLine(interval.Equals(TimeSpan.Zero));      // Displays
"True".
```

- By calling one of its explicit constructors. The following example initializes a [TimeSpan](#) value to a specified number of hours, minutes, and seconds.

C#

```
TimeSpan interval = new TimeSpan(2, 14, 18);
Console.WriteLine(interval.ToString());
// Displays "02:14:18".
```

- By calling a method or performing an operation that returns a [TimeSpan](#) value. For example, you can instantiate a [TimeSpan](#) value that represents the interval between two date and time values, as the following example shows.

C#

```
DateTime departure = new DateTime(2010, 6, 12, 18, 32, 0);
DateTime arrival = new DateTime(2010, 6, 13, 22, 47, 0);
TimeSpan travelTime = arrival - departure;
Console.WriteLine("{0} - {1} = {2}", arrival, departure, travelTime);

// The example displays the following output:
//       6/13/2010 10:47:00 PM - 6/12/2010 6:32:00 PM = 1.04:15:00
```

You can also initialize a [TimeSpan](#) object to a zero time value in this way, as the following example shows.

C#

```
Random rnd = new Random();

TimeSpan timeSpent = TimeSpan.Zero;

timeSpent += GetTimeBeforeLunch();
timeSpent += GetTimeAfterLunch();

Console.WriteLine("Total time: {0}", timeSpent);

TimeSpan GetTimeBeforeLunch()
{
    return new TimeSpan(rnd.Next(3, 6), 0, 0);
}

TimeSpan GetTimeAfterLunch()
```

```

{
    return new TimeSpan(rnd.Next(3, 6), 0, 0);
}

// The example displays output like the following:
//      Total time: 08:00:00

```

`TimeSpan` values are returned by arithmetic operators and methods of the `DateTime`, `DateTimeOffset`, and `TimeSpan` structures.

- By parsing the string representation of a `TimeSpan` value. You can use the `Parse` and `TryParse` methods to convert strings that contain time intervals to `TimeSpan` values. The following example uses the `Parse` method to convert an array of strings to `TimeSpan` values.

C#

```

string[] values = { "12", "31.", "5.8:32:16", "12:12:15.95", ".12" };
foreach (string value in values)
{
    try {
        TimeSpan ts = TimeSpan.Parse(value);
        Console.WriteLine("{0} --> {1}", value, ts);
    }
    catch (FormatException) {
        Console.WriteLine("Unable to parse '{0}'", value);
    }
    catch (OverflowException) {
        Console.WriteLine("{0} is outside the range of a TimeSpan.", value);
    }
}

// The example displays the following output:
//      '12' --> 12.00:00:00
//      Unable to parse '31.'
//      '5.8:32:16' --> 5.08:32:16
//      '12:12:15.95' --> 12:12:15.9500000
//      Unable to parse '.12'

```

In addition, you can define the precise format of the input string to be parsed and converted to a `TimeSpan` value by calling the `ParseExact` or `TryParseExact` method.

## Perform operations on `TimeSpan` values

You can add and subtract time durations either by using the `Addition` and `Subtraction` operators, or by calling the `Add` and `Subtract` methods. You can also compare two time durations by calling the `Compare`, `CompareTo`, and `Equals` methods. The `TimeSpan`

structure also includes the [Duration](#) and [Negate](#) methods, which convert time intervals to positive and negative values,

The range of [TimeSpan](#) values is [MinValue](#) to [MaxValue](#).

## Format a TimeSpan value

A [TimeSpan](#) value can be represented as  $[-]d.hh:mm:ss.fff$ , where the optional minus sign indicates a negative time interval, the  $d$  component is days,  $hh$  is hours as measured on a 24-hour clock,  $mm$  is minutes,  $ss$  is seconds, and  $fff$  is fractions of a second. That is, a time interval consists of a positive or negative number of days without a time of day, or a number of days with a time of day, or only a time of day.

Beginning with .NET Framework 4, the [TimeSpan](#) structure supports culture-sensitive formatting through the overloads of its [ToString](#) method, which converts a [TimeSpan](#) value to its string representation. The default [TimeSpan.ToString\(\)](#) method returns a time interval by using an invariant format that is identical to its return value in previous versions of .NET Framework. The [TimeSpan.ToString\(String\)](#) overload lets you specify a format string that defines the string representation of the time interval. The [TimeSpan.ToString\(String, IFormatProvider\)](#) overload lets you specify a format string and the culture whose formatting conventions are used to create the string representation of the time interval. [TimeSpan](#) supports both standard and custom format strings. (For more information, see [Standard TimeSpan Format Strings](#) and [Custom TimeSpan Format Strings](#).) However, only standard format strings are culture-sensitive.

## Restore legacy TimeSpan formatting

In some cases, code that successfully formats [TimeSpan](#) values in .NET Framework 3.5 and earlier versions fails in .NET Framework 4. This is most common in code that calls a [<TimeSpan\\_LegacyFormatMode>](#) element method to format a [TimeSpan](#) value with a format string. The following example successfully formats a [TimeSpan](#) value in .NET Framework 3.5 and earlier versions, but throws an exception in .NET Framework 4 and later versions. Note that it attempts to format a [TimeSpan](#) value by using an unsupported format specifier, which is ignored in .NET Framework 3.5 and earlier versions.

C#

```
ShowFormattingCode();
// Output from .NET Framework 3.5 and earlier versions:
//      12:30:45
// Output from .NET Framework 4:
```

```

//      Invalid Format

Console.WriteLine("---");

ShowParsingCode();
// Output:
//      00000006 --> 6.00:00:00

void ShowFormattingCode()
{
    TimeSpan interval = new TimeSpan(12, 30, 45);
    string output;
    try
    {
        output = String.Format("{0:r}", interval);
    }
    catch (FormatException)
    {
        output = "Invalid Format";
    }
    Console.WriteLine(output);
}

void ShowParsingCode()
{
    string value = "000000006";
    try
    {
        TimeSpan interval = TimeSpan.Parse(value);
        Console.WriteLine("{0} --> {1}", value, interval);
    }
    catch (FormatException)
    {
        Console.WriteLine("{0}: Bad Format", value);
    }
    catch (OverflowException)
    {
        Console.WriteLine("{0}: Overflow", value);
    }
}

```

If you cannot modify the code, you can restore the legacy formatting of [TimeSpan](#) values in one of the following ways:

- By creating a configuration file that contains the [<TimeSpan\\_LegacyFormatMode> element](#). Setting this element's `enabled` attribute to `true` restores legacy [TimeSpan](#) formatting on a per-application basis.
- By setting the "NetFx40\_TimeSpanLegacyFormatMode" compatibility switch when you create an application domain. This enables legacy [TimeSpan](#) formatting on a

per-application-domain basis. The following example creates an application domain that uses legacy [TimeSpan](#) formatting.

```
C#  
  
using System;  
  
public class Example2  
{  
    public static void Main()  
    {  
        AppDomainSetup appSetup = new AppDomainSetup();  
        appSetup.SetCompatibilitySwitches(new string[] {  
    "NetFx40_TimeSpanLegacyFormatMode" });  
        AppDomain legacyDomain = AppDomain.CreateDomain("legacyDomain",  
                                            null,  
                                            appSetup);  
        legacyDomain.ExecuteAssembly("ShowTimeSpan.exe");  
    }  
}
```

When the following code executes in the new application domain, it reverts to legacy [TimeSpan](#) formatting behavior.

```
C#  
  
using System;  
  
public class Example3  
{  
    public static void Main()  
    {  
        TimeSpan interval = DateTime.Now - DateTime.Now.Date;  
        string msg = String.Format("Elapsed Time Today: {0:d} hours.",  
                                    interval);  
        Console.WriteLine(msg);  
    }  
}  
// The example displays the following output:  
//     Elapsed Time Today: 01:40:52.2524662 hours.
```

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review

 .NET

[.NET feedback](#)

.NET is an open source project.  
Select a link to provide feedback:

 [Open a documentation issue](#)

issues and pull requests. For more information, see [our contributor guide](#).

 [Provide product feedback](#)

# System.TimeSpan.Parse method

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

The input string to the [Parse](#) methods contains a time interval specification in the form:

```
[ws][-]{ d | [d.]hh:mm[:ss[.ff]] }[ws]
```

Elements in square brackets ([ and ]) are optional. One selection from the list of alternatives enclosed in braces ({ and }) and separated by vertical bars (|) is required. The following table describes each element.

[Expand table](#)

Element	Description
ws	Optional white space.
-	An optional minus sign, which indicates a negative <a href="#">TimeSpan</a> .
d	Days, ranging from 0 to 10675199.
.	A culture-sensitive symbol that separates days from hours. The invariant format uses a period (".") character.
hh	Hours, ranging from 0 to 23.
:	The culture-sensitive time separator symbol. The invariant format uses a colon (":") character.
mm	Minutes, ranging from 0 to 59.
ss	Optional seconds, ranging from 0 to 59.
.	A culture-sensitive symbol that separates seconds from fractions of a second. The invariant format uses a period (".") character.
ff	Optional fractional seconds, consisting of one to seven decimal digits.

If the input string is not a day value only, it must include an hours and a minutes component; other components are optional. If they are present, the values of each time component must fall within a specified range. For example, the value of *hh*, the hours component, must be between 0 and 23. Because of this, passing "23:00:00" to the [Parse](#) method returns a time interval of 23 hours. On the other hand, passing "24:00:00"

returns a time interval of 24 days. Because "24" is outside the range of the hours component, it is interpreted as the days component.

The components of the input string must collectively specify a time interval that is greater than or equal to [TimeSpan.MinValue](#) and less than or equal to [TimeSpan.MaxValue](#).

The [Parse\(String\)](#) method tries to parse the input string by using each of the culture-specific formats for the current culture.

## Notes to callers

When a time interval component in the string to be parsed contains more than seven digits, parsing operations in .NET Framework 3.5 and earlier versions may behave differently from parsing operations in .NET Framework 4 and later versions. In some cases, parsing operations that succeed in .NET Framework 3.5 and earlier versions may fail and throw an [OverflowException](#) in .NET Framework 4 and later. In other cases, parsing operations that throw a [FormatException](#) in .NET Framework 3.5 and earlier versions may fail and throw an [OverflowException](#) in .NET Framework 4 and later. The following example illustrates both scenarios.

C#

```
string[] values = { "00000006", "12.12:12:12.12345678" };
foreach (string value in values)
{
    try {
        TimeSpan interval = TimeSpan.Parse(value);
        Console.WriteLine("{0} --> {1}", value, interval);
    }
    catch (FormatException) {
        Console.WriteLine("{0}: Bad Format", value);
    }
    catch (OverflowException) {
        Console.WriteLine("{0}: Overflow", value);
    }
}

// Output from .NET Framework 3.5 and earlier versions:
//      00000006 --> 6.00:00:00
//      12.12:12:12.12345678: Bad Format
// Output from .NET Framework 4 and later versions or .NET Core:
//      00000006: Overflow
//      12.12:12:12.12345678: Overflow
```

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# System.TimeSpan.TryParse methods

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

## TryParse(System.String, System.TimeSpan@) method

The [TimeSpan.TryParse\(String, TimeSpan\)](#) method is like the [TimeSpan.Parse\(String\)](#) method, except that it doesn't throw an exception if the conversion fails.

The `s` parameter contains a time interval specification in the form:

```
[ws][-]{ d | d.hh:mm[:ss[.ff]] | hh:mm[:ss[.ff]] }[ws]
```

Elements in square brackets ([ and ]) are optional. One selection from the list of alternatives enclosed in braces ({ and }) and separated by vertical bars (|) is required. The following table describes each element.

[ ] [Expand table](#)

Element	Description
<code>ws</code>	Optional white space.
<code>-</code>	An optional minus sign, which indicates a negative <a href="#">TimeSpan</a> .
<code>d</code>	Days, ranging from 0 to 10675199.
<code>.</code>	A culture-sensitive symbol that separates days from hours. The invariant format uses a period (".") character.
<code>hh</code>	Hours, ranging from 0 to 23.
<code>:</code>	The culture-sensitive time separator symbol. The invariant format uses a colon (":") character.
<code>mm</code>	Minutes, ranging from 0 to 59.
<code>ss</code>	Optional seconds, ranging from 0 to 59.
<code>.</code>	A culture-sensitive symbol that separates seconds from fractions of a second. The invariant format uses a period (".") character.

Element	Description
<i>ff</i>	Optional fractional seconds, consisting of one to seven decimal digits.

The components of *s* must collectively specify a time interval that's greater than or equal to [TimeSpan.MinValue](#) and less than or equal to [TimeSpan.MaxValue](#).

The [Parse\(String\)](#) method tries to parse *s* by using each of the culture-specific formats for the current culture.

## TryParse(String, IFormatProvider, TimeSpan) method

The [TryParse\(String, IFormatProvider, TimeSpan\)](#) method is like the [Parse\(String, IFormatProvider\)](#) method, except that it does not throw an exception if the conversion fails.

The *input* parameter contains a time interval specification in the form:

```
[ws][-]{ d | d.hh:mm[:ss[.ff]] | hh:mm[:ss[.ff]] }[ws]
```

Elements in square brackets ([ and ]) are optional. One selection from the list of alternatives enclosed in braces ({ and }) and separated by vertical bars (|) is required. The following table describes each element.

[Expand table](#)

Element	Description
<i>ws</i>	Optional white space.
-	An optional minus sign, which indicates a negative <a href="#">TimeSpan</a> .
<i>d</i>	Days, ranging from 0 to 10675199.
.	A culture-sensitive symbol that separates days from hours. The invariant format uses a period (".") character.
<i>hh</i>	Hours, ranging from 0 to 23.
:	The culture-sensitive time separator symbol. The invariant format uses a colon (":") character.
<i>mm</i>	Minutes, ranging from 0 to 59.
<i>ss</i>	Optional seconds, ranging from 0 to 59.

Element	Description
.	A culture-sensitive symbol that separates seconds from fractions of a second. The invariant format uses a period (".") character.
ff	Optional fractional seconds, consisting of one to seven decimal digits.

The components of `input` must collectively specify a time interval that is greater than or equal to `TimeSpan.MinValue` and less than or equal to `TimeSpan.MaxValue`.

The `TryParse(String, IFormatProvider, TimeSpan)` method tries to parse `input` by using each of the culture-specific formats for the culture specified by `formatProvider`.

The `formatProvider` parameter is an `IFormatProvider` implementation that provides culture-specific information about the format of the returned string. The `formatProvider` parameter can be any of the following:

- A `CultureInfo` object that represents the culture whose formatting conventions are to be reflected in the returned string. The `DateTimeFormatInfo` object returned by the `CultureInfo.DateTimeFormat` property defines the formatting of the returned string.
- A `DateTimeFormatInfo` object that defines the formatting of the returned string.
- A custom object that implements the `IFormatProvider` interface. Its `IFormatProvider.GetFormat` method returns a `DateTimeFormatInfo` object that provides formatting information.

If `formatProvider` is `null`, the `DateTimeFormatInfo` object that is associated with the current culture is used.

## Notes to callers

In some cases, when a time interval component in the string to be parsed contains more than seven digits, parsing operations that succeeded and returned `true` in .NET Framework 3.5 and earlier versions may fail and return `false` in .NET Framework 4 and later versions. The following example illustrates this scenario:

C#

```
string value = "00000006";
TimeSpan interval;
if (TimeSpan.TryParse(value, out interval))
    Console.WriteLine("{0} --> {1}", value, interval);
else
    Console.WriteLine("Unable to parse '{0}'", value);
```

```
// Output from .NET Framework 3.5 and earlier versions:  
//      00000006 --> 6.00:00:00  
// Output from .NET Framework 4:  
//      Unable to parse //00000006//
```

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Extend metadata using attributes

Article • 09/15/2021

The common language runtime allows you to add keyword-like descriptive declarations, called attributes, to annotate programming elements such as types, fields, methods, and properties. When you compile your code for the runtime, it is converted into Microsoft intermediate language (MSIL) and placed inside a portable executable (PE) file along with metadata generated by the compiler. Attributes allow you to place extra descriptive information into metadata that can be extracted using runtime reflection services. The compiler creates attributes when you declare instances of special classes that derive from [System.Attribute](#).

.NET uses attributes for a variety of reasons and to address a number of issues. Attributes describe how to serialize data, specify characteristics that are used to enforce security, and limit optimizations by the just-in-time (JIT) compiler so the code remains easy to debug. Attributes can also record the name of a file or the author of code, or control the visibility of controls and members during forms development.

## Related articles

Title	Description
<a href="#">Applying Attributes</a>	Describes how to apply an attribute to an element of your code.
<a href="#">Writing Custom Attributes</a>	Describes how to design custom attribute classes.
<a href="#">Retrieving Information Stored in Attributes</a>	Describes how to retrieve custom attributes for code that is loaded into the execution context.
<a href="#">Metadata and Self-Describing Components</a>	Provides an overview of metadata and describes how it is implemented in a .NET portable executable (PE) file.
<a href="#">How to: Load Assemblies into the Reflection-Only Context</a>	Explains how to retrieve custom attribute information in the reflection-only context.

## Reference

- [System.Attribute](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Apply attributes

Article • 03/15/2023

Use the following process to apply an attribute to an element of your code.

1. Define a new attribute or use an existing .NET attribute.
2. Apply the attribute to the code element by placing it immediately before the element.

Each language has its own attribute syntax. In C++ and C#, the attribute is surrounded by square brackets and separated from the element by white space, which can include a line break. In Visual Basic, the attribute is surrounded by angle brackets and must be on the same logical line; the line continuation character can be used if a line break is desired.

3. Specify positional parameters and named parameters for the attribute.

*Positional* parameters are required and must come before any named parameters; they correspond to the parameters of one of the attribute's constructors. *Named* parameters are optional and correspond to read/write properties of the attribute. In C++, and C#, specify `name=value` for each optional parameter, where `name` is the name of the property. In Visual Basic, specify `name:=value`.

The attribute is emitted into metadata when you compile your code and is available to the common language runtime and any custom tool or application through the runtime reflection services.

By convention, all attribute names end with "Attribute". However, several languages that target the runtime, such as Visual Basic and C#, do not require you to specify the full name of an attribute. For example, if you want to initialize [System.ObsoleteAttribute](#), you only need to reference it as **Obsolete**.

## Apply an attribute to a method

The following code example shows how to use [System.ObsoleteAttribute](#), which marks code as obsolete. The string `"Will be removed in next version"` is passed to the attribute. This attribute causes a compiler warning that displays the passed string when code that the attribute describes is called.

C#

```
public class Example
{
    // Specify attributes between square brackets in C#.
    // This attribute is applied only to the Add method.
    [Obsolete("Will be removed in next version.")]
    public static int Add(int a, int b)
    {
        return (a + b);
    }
}

class Test
{
    public static void Main()
    {
        // This generates a compile-time warning.
        int i = Example.Add(2, 2);
    }
}
```

## Apply attributes at the assembly level

If you want to apply an attribute at the assembly level, use the `assembly` (`Assembly` in Visual Basic) keyword. The following code shows the `AssemblyTitleAttribute` applied at the assembly level.

C#

```
using System.Reflection;
[assembly: AssemblyTitle("My Assembly")]
```

When this attribute is applied, the string `"My Assembly"` is placed in the assembly manifest in the metadata portion of the file. You can view the attribute either by using the [MSIL Disassembler \(Ildasm.exe\)](#) or by creating a custom program to retrieve the attribute.

## See also

- [Attributes](#)
- [Retrieving Information Stored in Attributes](#)
- [Concepts](#)
- [Attributes \(C#\)](#)
- [Attributes overview \(Visual Basic\)](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Write custom attributes

Article • 04/12/2023

To design custom attributes, you don't need to learn many new concepts. If you're familiar with object-oriented programming and know how to design classes, you already have most of the knowledge needed. Custom attributes are traditional classes that derive directly or indirectly from the [System.Attribute](#) class. Just like traditional classes, custom attributes contain methods that store and retrieve data.

The primary steps to properly design custom attribute classes are as follows:

- [Applying the AttributeUsageAttribute](#)
- [Declaring the attribute class](#)
- [Declaring constructors](#)
- [Declaring properties](#)

This section describes each of these steps and concludes with a [custom attribute example](#).

## Applying the AttributeUsageAttribute

A custom attribute declaration begins with the [System.AttributeUsageAttribute](#) attribute, which defines some of the key characteristics of your attribute class. For example, you can specify whether your attribute can be inherited by other classes or which elements the attribute can be applied to. The following code fragment demonstrates how to use the [AttributeUsageAttribute](#):

C#

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]
```

The [AttributeUsageAttribute](#) has three members that are important for the creation of custom attributes: [AttributeTargets](#), [Inherited](#), and [AllowMultiple](#).

## AttributeTargets Member

In the preceding example, [AttributeTargets.All](#) is specified, indicating that this attribute can be applied to all program elements. Alternatively, you can specify

`AttributeTargets.Class`, indicating that your attribute can be applied only to a class, or `AttributeTargets.Method`, indicating that your attribute can be applied only to a method. All program elements can be marked for description by a custom attribute in this manner.

You can also pass multiple `AttributeTargets` values. The following code fragment specifies that a custom attribute can be applied to any class or method:

```
C#
```

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
```

## Inherited Property

The `AttributeUsageAttribute.Inherited` property indicates whether your attribute can be inherited by classes that are derived from the classes to which your attribute is applied. This property takes either a `true` (the default) or `false` flag. In the following example, `MyAttribute` has a default `Inherited` value of `true`, while `YourAttribute` has an `Inherited` value of `false`:

```
C#
```

```
// This defaults to Inherited = true.
public class MyAttribute : Attribute
{
    //...
}

[AttributeUsage(AttributeTargets.Method, Inherited = false)]
public class YourAttribute : Attribute
{
    //...
}
```

The two attributes are then applied to a method in the base class `MyClass`:

```
C#
```

```
public class MyClass
{
    [MyAttribute]
    [YourAttribute]
    public virtual void MyMethod()
    {
        //...
    }
}
```

```
    }  
}
```

Finally, the class `YourClass` is inherited from the base class `MyClass`. The method `MyMethod` shows `MyAttribute` but not `YourAttribute`:

C#

```
public class YourClass : MyClass  
{  
    // MyMethod will have MyAttribute but not YourAttribute.  
    public override void MyMethod()  
    {  
        //...  
    }  
}
```

## AllowMultiple Property

The `AttributeUsageAttribute.AllowMultiple` property indicates whether multiple instances of your attribute can exist on an element. If set to `true`, multiple instances are allowed. If set to `false` (the default), only one instance is allowed.

In the following example, `MyAttribute` has a default `AllowMultiple` value of `false`, while `YourAttribute` has a value of `true`:

C#

```
//This defaults to AllowMultiple = false.  
public class MyAttribute : Attribute  
{  
}  
  
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]  
public class YourAttribute : Attribute  
{  
}
```

When multiple instances of these attributes are applied, `MyAttribute` produces a compiler error. The following code example shows the valid use of `YourAttribute` and the invalid use of `MyAttribute`:

C#

```
public class MyClass  
{
```

```

// This produces an error.
// Duplicates are not allowed.
[MyAttribute]
[MyAttribute]
public void MyMethod()
{
    //...
}

// This is valid.
[YourAttribute]
[YourAttribute]
public void YourMethod()
{
    //...
}

```

If both the `AllowMultiple` property and the `Inherited` property are set to `true`, a class that's inherited from another class can inherit an attribute and have another instance of the same attribute applied in the same child class. If `AllowMultiple` is set to `false`, the values of any attributes in the parent class will be overwritten by new instances of the same attribute in the child class.

## Declaring the Attribute Class

After you apply the `AttributeUsageAttribute`, start defining the specifics of your attribute. The declaration of an attribute class looks similar to the declaration of a traditional class, as demonstrated by the following code:

C#

```

[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute : Attribute
{
    // . . .
}

```

This attribute definition demonstrates the following points:

- Attribute classes must be declared as public classes.
- By convention, the name of the attribute class ends with the word **Attribute**. While not required, this convention is recommended for readability. When the attribute is applied, the inclusion of the word **Attribute** is optional.

- All attribute classes must inherit directly or indirectly from the [System.Attribute](#) class.
- In Microsoft Visual Basic, all custom attribute classes must have the [System.AttributeUsageAttribute](#) attribute.

## Declaring Constructors

Just like traditional classes, attributes are initialized with constructors. The following code fragment illustrates a typical attribute constructor. This public constructor takes a parameter and sets a member variable equal to its value.

C#

```
public MyAttribute(bool myvalue)
{
    this.myvalue = myvalue;
}
```

You can overload the constructor to accommodate different combinations of values. If you also define a [property](#) for your custom attribute class, you can use a combination of named and positional parameters when initializing the attribute. Typically, you define all required parameters as positional and all optional parameters as named. In this case, the attribute can't be initialized without the required parameter. All other parameters are optional.

 **Note**

In Visual Basic, constructors for an attribute class shouldn't use a `ParamArray` argument.

The following code example shows how an attribute that uses the previous constructor can be applied using optional and required parameters. It assumes that the attribute has one required Boolean value and one optional string property.

C#

```
// One required (positional) and one optional (named) parameter are applied.
[MyAttribute(false, OptionalParameter = "optional data")]
public class SomeClass
{
    //...
}
// One required (positional) parameter is applied.
```

```
[MyAttribute(false)]
public class SomeOtherClass
{
    //...
}
```

## Declaring Properties

If you want to define a named parameter or provide an easy way to return the values stored by your attribute, declare a [property](#). Attribute properties should be declared as public entities with a description of the data type that will be returned. Define the variable that will hold the value of your property and associate it with the `get` and `set` methods. The following code example demonstrates how to implement a property in your attribute:

C#

```
public bool MyProperty
{
    get {return this.myvalue;}
    set {this.myvalue = value;}
}
```

## Custom Attribute Example

This section incorporates the previous information and shows how to design an attribute that documents information about the author of a section of code. The attribute in this example stores the name and level of the programmer, and whether the code has been reviewed. It uses three private variables to store the actual values to save. Each variable is represented by a public property that gets and sets the values. Finally, the constructor is defined with two required parameters:

C#

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperAttribute : Attribute
{
    // Private fields.
    private string name;
    private string level;
    private bool reviewed;

    // This constructor defines two required parameters: name and level.

    public DeveloperAttribute(string name, string level)
```

```
{

    this.name = name;
    this.level = level;
    this.reviewed = false;
}

// Define Name property.
// This is a read-only attribute.

public virtual string Name
{
    get {return name;}
}

// Define Level property.
// This is a read-only attribute.

public virtual string Level
{
    get {return level;}
}

// Define Reviewed property.
// This is a read/write attribute.

public virtual bool Reviewed
{
    get {return reviewed;}
    set {reviewed = value;}
}
}
```

You can apply this attribute using the full name, `DeveloperAttribute`, or using the abbreviated name, `Developer`, in one of the following ways:

C#

```
[Developer("Joan Smith", "1")]

-or-

[Developer("Joan Smith", "1", Reviewed = true)]
```

The first example shows the attribute applied with only the required named parameters. The second example shows the attribute applied with both the required and optional parameters.

## See also

- [System.Attribute](#)
- [System.AttributeUsageAttribute](#)
- [Attributes](#)
- [Attribute parameter types](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Retrieving Information Stored in Attributes

Article • 10/04/2022

Retrieving a custom attribute is a simple process. First, declare an instance of the attribute you want to retrieve. Then, use the [Attribute.GetCustomAttribute](#) method to initialize the new attribute to the value of the attribute you want to retrieve. Once the new attribute is initialized, you can use its properties to get the values.

## Important

This article describes how to retrieve attributes for code loaded into the execution context. To retrieve attributes for code loaded into the reflection-only context, you must use the [CustomAttributeData](#) class, as shown in [How to: Load Assemblies into the Reflection-Only Context](#).

This section describes the following ways to retrieve attributes:

- [Retrieving a single instance of an attribute](#)
- [Retrieving multiple instances of an attribute applied to the same scope](#)
- [Retrieving multiple instances of an attribute applied to different scopes](#)

## Retrieving a Single Instance of an Attribute

In the following example, the `DeveloperAttribute` (described in the previous section) is applied to the `MainApp` class on the class level. The `GetAttribute` method uses `GetCustomAttribute` to retrieve the values stored in `DeveloperAttribute` on the class level before displaying them to the console.

C#

```
using System;
using System.Reflection;
using CustomCodeAttributes;

[Developer("Joan Smith", "42", Reviewed = true)]
class MainApp
{
    public static void Main()
    {
```

```

        // Call function to get and display the attribute.
        GetAttribute(typeof(MainApp));
    }

    public static void GetAttribute(Type t)
    {
        // Get instance of the attribute.
        DeveloperAttribute MyAttribute =
            (DeveloperAttribute) Attribute.GetCustomAttribute(t, typeof
(DeveloperAttribute));

        if (MyAttribute == null)
        {
            Console.WriteLine("The attribute was not found.");
        }
        else
        {
            // Get the Name value.
            Console.WriteLine("The Name Attribute is: {0}." ,
MyAttribute.Name);
            // Get the Level value.
            Console.WriteLine("The Level Attribute is: {0}." ,
MyAttribute.Level);
            // Get the Reviewed value.
            Console.WriteLine("The Reviewed Attribute is: {0}." ,
MyAttribute.Reviewed);
        }
    }
}

```

The execution of the preceding program displays the following text:

```

Console

The Name Attribute is: Joan Smith.
The Level Attribute is: 42.
The Reviewed Attribute is: True.

```

If the attribute isn't found, the `GetCustomAttribute` method initializes `MyAttribute` to a null value. This example checks `MyAttribute` for such an instance and notifies the user if the attribute isn't found. If `DeveloperAttribute` isn't found in the class scope, the console displays the following message:

```

Console

The attribute was not found.

```

The preceding example assumes that the attribute definition is in the current namespace. Remember to import the namespace in which the attribute definition

resides if it isn't in the current namespace.

## Retrieving Multiple Instances of an Attribute Applied to the Same Scope

In the preceding example, the class to inspect and the specific attribute to find are passed to the `GetCustomAttribute` method. That code works well if only one instance of an attribute is applied on the class level. However, if multiple instances of an attribute are applied on the same class level, the `GetCustomAttribute` method doesn't retrieve all the information. In cases where multiple instances of the same attribute are applied to the same scope, you can use `Attribute.GetCustomAttributes` method to place all instances of an attribute into an array. For example, if two instances of `DeveloperAttribute` are applied on the class level of the same class, the `GetAttribute` method can be modified to display the information found in both attributes. Remember, to apply multiple attributes on the same level, the attribute must be defined with the `AllowMultiple` property set to `true` in the `AttributeUsageAttribute` class.

The following code example shows how to use the `GetCustomAttributes` method to create an array that references all instances of `DeveloperAttribute` in any given class. The code then outputs the values of all the attributes to the console.

C#

```
public static void GetAttribute(Type t)
{
    DeveloperAttribute[] MyAttributes =
        (DeveloperAttribute[]) Attribute.GetCustomAttributes(t, typeof
(DeveloperAttribute));

    if (MyAttributes.Length == 0)
    {
        Console.WriteLine("The attribute was not found.");
    }
    else
    {
        for (int i = 0 ; i < MyAttributes.Length ; i++)
        {
            // Get the Name value.
            Console.WriteLine("The Name Attribute is: {0}." ,
MyAttributes[i].Name);
            // Get the Level value.
            Console.WriteLine("The Level Attribute is: {0}." ,
MyAttributes[i].Level);
            // Get the Reviewed value.
            Console.WriteLine("The Reviewed Attribute is: {0}." ,
MyAttributes[i].Reviewed);
```

```
        }
    }
}
```

If no attributes are found, this code alerts the user. Otherwise, the information contained in both instances of `DeveloperAttribute` is displayed.

## Retrieving Multiple Instances of an Attribute Applied to Different Scopes

The `GetCustomAttributes` and `GetCustomAttribute` methods don't search an entire class and return all instances of an attribute in that class. Rather, they search only one specified method or member at a time. If you have a class with the same attribute applied to every member and you want to retrieve the values in all the attributes applied to those members, you must supply every method or member individually to `GetCustomAttributes` and `GetCustomAttribute`.

The following code example takes a class as a parameter and searches for the `DeveloperAttribute` (defined previously) on the class level and on every individual method of that class:

C#

```
public static void GetAttribute(Type t)
{
    DeveloperAttribute att;

    // Get the class-level attributes.

    // Put the instance of the attribute on the class level in the att
    object.
    att = (DeveloperAttribute) Attribute.GetCustomAttribute (t, typeof
    (DeveloperAttribute));

    if (att == null)
    {
        Console.WriteLine("No attribute in class {0}.\n", t.ToString());
    }
    else
    {
        Console.WriteLine("The Name Attribute on the class level is: {0}.", att.Name);
        Console.WriteLine("The Level Attribute on the class level is: {0}.", att.Level);
        Console.WriteLine("The Reviewed Attribute on the class level is:
{0}.\n", att.Reviewed);
    }
}
```

```

// Get the method-level attributes.

// Get all methods in this class, and put them
// in an array of System.Reflection.MemberInfo objects.
MethodInfo[] MyMethodInfo = t.GetMethods();

// Loop through all methods in this class that are in the
// MyMethodInfo array.
for (int i = 0; i < MyMethodInfo.Length; i++)
{
    att = (DeveloperAttribute)
Attribute.GetCustomAttribute(MyMethodInfo[i], typeof(DeveloperAttribute));
    if (att == null)
    {
        Console.WriteLine("No attribute in member function {0}.\n" ,
MyMethodInfo[i].ToString());
    }
    else
    {
        Console.WriteLine("The Name Attribute for the {0} member is:
{1}.",
                           MyMethodInfo[i].ToString(), att.Name);
        Console.WriteLine("The Level Attribute for the {0} member is:
{1}.",
                           MyMethodInfo[i].ToString(), att.Level);
        Console.WriteLine("The Reviewed Attribute for the {0} member is:
{1}.\n",
                           MyMethodInfo[i].ToString(), att.Reviewed);
    }
}
}

```

If no instances of the `DeveloperAttribute` are found on the method level or class level, the `GetAttribute` method notifies the user that no attributes were found and displays the name of the method or class that doesn't contain the attribute. If an attribute is found, the console displays the `Name`, `Level`, and `Reviewed` fields.

You can use the members of the `Type` class to get the individual methods and members in the passed class. This example first queries the `Type` object to get attribute information for the class level. Next, it uses `Type.GetMethods` to place instances of all methods into an array of `System.Reflection.MemberInfo` objects to retrieve attribute information for the method level. You can also use the `Type.GetProperties` method to check for attributes on the property level or `Type.GetConstructors` to check for attributes on the constructor level.

## See also

- [System.Type](#)
- [Attribute.GetCustomAttribute](#)
- [Attribute.GetCustomAttributes](#)
- [Attributes](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Memory- and span-related types

Article • 09/21/2023

Starting with .NET Core 2.1, .NET includes a number of interrelated types that represent a contiguous, strongly typed region of arbitrary memory. These include:

- [System.Span<T>](#), a type that is used to access a contiguous region of memory. A `Span<T>` instance can be backed by an array of type `T`, a buffer allocated with `stackalloc`, or a pointer to unmanaged memory. Because it has to be allocated on the stack, it has a number of restrictions. For example, a field in a class cannot be of type `Span<T>`, nor can span be used in asynchronous operations.
- [System.ReadOnlySpan<T>](#), an immutable version of the `Span<T>` structure. Instances can be also backed by a `String`.
- [System.Memory<T>](#), a wrapper over a contiguous region of memory. A `Memory<T>` instance can be backed by an array of type `T` or a memory manager. As it can be stored on the managed heap, `Memory<T>` has none of the limitations of `Span<T>`.
- [System.ReadOnlyMemory<T>](#), an immutable version of the `Memory<T>` structure. Instances can also be backed by a `String`.
- [System.Buffers.MemoryPool<T>](#), which allocates strongly typed blocks of memory from a memory pool to an owner. `IMemoryOwner<T>` instances can be rented from the pool by calling `MemoryPool<T>.Rent` and released back to the pool by calling `MemoryPool<T>.Dispose()`.
- [System.Buffers.IMemoryOwner<T>](#), which represents the owner of a block of memory and controls its lifetime management.
- [MemoryManager<T>](#), an abstract base class that can be used to replace the implementation of `Memory<T>` so that `Memory<T>` can be backed by additional types, such as safe handles. `MemoryManager<T>` is intended for advanced scenarios.
- [ArraySegment<T>](#), a wrapper for a particular number of array elements starting at a particular index.
- [System.MemoryExtensions](#), a collection of extension methods for converting strings, arrays, and array segments to `Memory<T>` blocks.

`System.Span<T>`, `System.Memory<T>`, and their readonly counterparts are designed to allow the creation of algorithms that avoid copying memory or allocating on the managed heap more than necessary. Creating them (either via `slice` or their constructors) does not involve duplicating the underlying buffers: only the relevant references and offsets, which represent the "view" of the wrapped memory, are updated.

### ⓘ Note

For earlier frameworks, `Span<T>` and `Memory<T>` are available in the [System.Memory NuGet package](#).

For more information, see the [System.Buffers](#) namespace.

## Working with memory and span

Because the memory- and span-related types are typically used to store data in a processing pipeline, it is important that developers follow a set of best practices when using `Span<T>`, `Memory<T>`, and related types. These best practices are documented in [Memory<T> and Span<T> usage guidelines](#).

## See also

- [System.Memory<T>](#)
- [System.ReadOnlyMemory<T>](#)
- [System.Span<T>](#)
- [System.ReadOnlySpan<T>](#)
- [System.Buffers](#)

### ⓘ Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

#### ⓘ Open a documentation issue

#### ⓘ Provide product feedback

# Memory<T> and Span<T> usage guidelines

Article • 04/19/2023

.NET includes a number of types that represent an arbitrary contiguous region of memory. [Span<T>](#) and [ReadOnlySpan<T>](#) are lightweight memory buffers that wrap references to managed or unmanaged memory. Because these types can only be stored on the stack, they're unsuitable for scenarios such as asynchronous method calls. To address this problem, .NET 2.1 added some additional types, including [Memory<T>](#), [ReadOnlyMemory<T>](#), [IMemoryOwner<T>](#), and [MemoryPool<T>](#). Like [Span<T>](#), [Memory<T>](#) and its related types can be backed by both managed and unmanaged memory. Unlike [Span<T>](#), [Memory<T>](#) can be stored on the managed heap.

Both [Span<T>](#) and [Memory<T>](#) are wrappers over buffers of structured data that can be used in pipelines. That is, they're designed so that some or all of the data can be efficiently passed to components in the pipeline, which can process them and optionally modify the buffer. Because [Memory<T>](#) and its related types can be accessed by multiple components or by multiple threads, it's important to follow some standard usage guidelines to produce robust code.

## Owners, consumers, and lifetime management

Buffers can be passed around between APIs and can sometimes be accessed from multiple threads, so be aware of how a buffer's lifetime is managed. There are three core concepts:

- **Ownership.** The owner of a buffer instance is responsible for lifetime management, including destroying the buffer when it's no longer in use. All buffers have a single owner. Generally the owner is the component that created the buffer or that received the buffer from a factory. Ownership can also be transferred; **Component-A** can relinquish control of the buffer to **Component-B**, at which point **Component-A** may no longer use the buffer, and **Component-B** becomes responsible for destroying the buffer when it's no longer in use.
- **Consumption.** The consumer of a buffer instance is allowed to use the buffer instance by reading from it and possibly writing to it. Buffers can have one consumer at a time unless some external synchronization mechanism is provided. The active consumer of a buffer isn't necessarily the buffer's owner.

- **Lease.** The lease is the length of time that a particular component is allowed to be the consumer of the buffer.

The following pseudo-code example illustrates these three concepts. `Buffer` in the pseudo-code represents a `Memory<T>` or `Span<T>` buffer of type `Char`. The `Main` method instantiates the buffer, calls the `WriteInt32ToBuffer` method to write the string representation of an integer to the buffer, and then calls the `DisplayBufferToConsole` method to display the value of the buffer.

C#

```
using System;

class Program
{
    // Write 'value' as a human-readable string to the output buffer.
    void WriteInt32ToBuffer(int value, Buffer buffer);

    // Display the contents of the buffer to the console.
    void DisplayBufferToConsole(Buffer buffer);

    // Application code
    static void Main()
    {
        var buffer = CreateBuffer();
        try
        {
            int value = Int32.Parse(Console.ReadLine());
            WriteInt32ToBuffer(value, buffer);
            DisplayBufferToConsole(buffer);
        }
        finally
        {
            buffer.Destroy();
        }
    }
}
```

The `Main` method creates the buffer and so is its owner. Therefore, `Main` is responsible for destroying the buffer when it's no longer in use. The pseudo-code illustrates this by calling a `Destroy` method on the buffer. (Neither `Memory<T>` nor `Span<T>` actually has a `Destroy` method. You'll see actual code examples later in this article.)

The buffer has two consumers, `WriteInt32ToBuffer` and `DisplayBufferToConsole`. There is only one consumer at a time (first `WriteInt32ToBuffer`, then `DisplayBufferToConsole`), and neither of the consumers owns the buffer. Note also that "consumer" in this context

doesn't imply a read-only view of the buffer; consumers can modify the buffer's contents, as `WriteInt32ToBuffer` does, if given a read/write view of the buffer.

The `WriteInt32ToBuffer` method has a lease on (can consume) the buffer between the start of the method call and the time the method returns. Similarly, `DisplayBufferToConsole` has a lease on the buffer while it's executing, and the lease is released when the method unwinds. (There is no API for lease management; a "lease" is a conceptual matter.)

## Memory<T> and the owner/consumer model

As the [Owners, consumers, and lifetime management](#) section notes, a buffer always has an owner. .NET supports two ownership models:

- A model that supports single ownership. A buffer has a single owner for its entire lifetime.
- A model that supports ownership transfer. Ownership of a buffer can be transferred from its original owner (its creator) to another component, which then becomes responsible for the buffer's lifetime management. That owner can in turn transfer ownership to another component, and so on.

You use the `System.Buffers.IMemoryOwner<T>` interface to explicitly manage the ownership of a buffer. `IMemoryOwner<T>` supports both ownership models. The component that has an `IMemoryOwner<T>` reference owns the buffer. The following example uses an `IMemoryOwner<T>` instance to reflect the ownership of a `Memory<T>` buffer.

C#

```
using System;
using System.Buffers;

class Example
{
    static void Main()
    {
        IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent();

        Console.Write("Enter a number: ");
        try
        {
            string? s = Console.ReadLine();

            if (s is null)
                return;
        }
    }
}
```

```

        var value = Int32.Parse(s);

        var memory = owner.Memory;

        WriteInt32ToBuffer(value, memory);

        DisplayBufferToConsole(owner.Memory.Slice(0,
value.ToString().Length));
    }
    catch (FormatException)
    {
        Console.WriteLine("You did not enter a valid number.");
    }
    catch (OverflowException)
    {
        Console.WriteLine($"You entered a number less than
{Int32.MinValue:N0} or greater than {Int32.MaxValue:N0}.");
    }
    finally
    {
        owner?.Dispose();
    }
}

static void WriteInt32ToBuffer(int value, Memory<char> buffer)
{
    var strValue = value.ToString();

    var span = buffer.Span;
    for (int ctr = 0; ctr < strValue.Length; ctr++)
        span[ctr] = strValue[ctr];
}

static void DisplayBufferToConsole(Memory<char> buffer) =>
    Console.WriteLine($"Contents of the buffer: '{buffer}'");
}

```

We can also write this example with the [using statement](#):

```

C#

using System;
using System.Buffers;

class Example
{
    static void Main()
    {
        using (IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent())
        {
            Console.Write("Enter a number: ");
            try

```

```

    {
        string? s = Console.ReadLine();

        if (s is null)
            return;

        var value = Int32.Parse(s);

        var memory = owner.Memory;
        WriteInt32ToBuffer(value, memory);
        DisplayBufferToConsole(memory.Slice(0,
value.ToString().Length));
    }
    catch (FormatException)
    {
        Console.WriteLine("You did not enter a valid number.");
    }
    catch (OverflowException)
    {
        Console.WriteLine($"You entered a number less than
{Int32.MinValue:N0} or greater than {Int32.MaxValue:N0}.");
    }
}

static void WriteInt32ToBuffer(int value, Memory<char> buffer)
{
    var strValue = value.ToString();

    var span = buffer.Slice(0, strValue.Length).Span;
    strValue.AsSpan().CopyTo(span);
}

static void DisplayBufferToConsole(Memory<char> buffer) =>
    Console.WriteLine($"Contents of the buffer: '{buffer}'");
}

```

In this code:

- The `Main` method holds the reference to the `IMemoryOwner<T>` instance, so the `Main` method is the owner of the buffer.
- The `WriteInt32ToBuffer` and `DisplayBufferToConsole` methods accept `Memory<T>` as a public API. Therefore, they are consumers of the buffer. These methods consume the buffer one at a time.

Although the `WriteInt32ToBuffer` method is intended to write a value to the buffer, the `DisplayBufferToConsole` method isn't intended to. To reflect this, it could have accepted an argument of type `ReadOnlyMemory<T>`. For more information on

`ReadOnlyMemory<T>`, see Rule #2: Use `ReadOnlySpan<T>` or `ReadOnlyMemory<T>` if the buffer should be read-only.

## "Ownerless" `Memory<T>` instances

You can create a `Memory<T>` instance without using `IMemoryOwner<T>`. In this case, ownership of the buffer is implicit rather than explicit, and only the single-owner model is supported. You can do this by:

- Calling one of the `Memory<T>` constructors directly, passing in a `T[]`, as the following example does.
- Calling the `String.AsMemory` extension method to produce a `ReadOnlyMemory<char>` instance.

C#

```
using System;

class Example
{
    static void Main()
    {
        Memory<char> memory = new char[64];

        Console.Write("Enter a number: ");
        string? s = Console.ReadLine();

        if (s is null)
            return;

        var value = Int32.Parse(s);

        WriteInt32ToBuffer(value, memory);
        DisplayBufferToConsole(memory);
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();
        strValue.AsSpan().CopyTo(buffer.Slice(0, strValue.Length).Span);
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");
}
```

The method that initially creates the `Memory<T>` instance is the implicit owner of the buffer. Ownership cannot be transferred to any other component because there is no

`IMemoryOwner<T>` instance to facilitate the transfer. (As an alternative, you can also imagine that the runtime's garbage collector owns the buffer, and all methods just consume the buffer.)

## Usage guidelines

Because a memory block is owned but is intended to be passed to multiple components, some of which may operate upon a particular memory block simultaneously, it's important to establish guidelines for using both `Memory<T>` and `Span<T>`. Guidelines are necessary because it's possible for a component to:

- Retain a reference to a memory block after its owner has released it.
- Operate on a buffer at the same time that another component is operating on it, in the process corrupting the data in the buffer.
- While the stack-allocated nature of `Span<T>` optimizes performance and makes `Span<T>` the preferred type for operating on a memory block, it also subjects `Span<T>` to some major restrictions. It's important to know when to use a `Span<T>` and when to use `Memory<T>`.

The following are our recommendations for successfully using `Memory<T>` and its related types. Guidance that applies to `Memory<T>` and `Span<T>` also applies to `ReadOnlyMemory<T>` and `ReadOnlySpan<T>` unless noted otherwise.

**Rule #1: For a synchronous API, use `Span<T>` instead of `Memory<T>` as a parameter if possible.**

`Span<T>` is more versatile than `Memory<T>` and can represent a wider variety of contiguous memory buffers. `Span<T>` also offers better performance than `Memory<T>`. Finally, you can use the `Memory<T>.Span` property to convert a `Memory<T>` instance to a `Span<T>`, although `Span<T>`-to-`Memory<T>` conversion isn't possible. So if your callers happen to have a `Memory<T>` instance, they'll be able to call your methods with `Span<T>` parameters anyway.

Using a parameter of type `Span<T>` instead of type `Memory<T>` also helps you write a correct consuming method implementation. You'll automatically get compile-time checks to ensure that you're not attempting to access the buffer beyond your method's lease (more on this later).

Sometimes, you'll have to use a `Memory<T>` parameter instead of a `Span<T>` parameter, even if you're fully synchronous. Perhaps an API that you depend on accepts

only `Memory<T>` arguments. This is fine, but be aware of the tradeoffs involved when using `Memory<T>` synchronously.

**Rule #2: Use `ReadOnlySpan<T>` or `ReadOnlyMemory<T>` if the buffer should be read-only.**

In the earlier examples, the `DisplayBufferToConsole` method only reads from the buffer; it doesn't modify the contents of the buffer. The method signature should be changed to the following.

C#

```
void DisplayBufferToConsole(ReadOnlyMemory<char> buffer);
```

In fact, if we combine this rule and Rule #1, we can do even better and rewrite the method signature as follows:

C#

```
void DisplayBufferToConsole(ReadOnlySpan<char> buffer);
```

The `DisplayBufferToConsole` method now works with virtually every buffer type imaginable: `T[]`, storage allocated with `stackalloc`, and so on. You can even pass a `String` directly into it! For more information, see GitHub issue [dotnet/docs #25551](#).

**Rule #3: If your method accepts `Memory<T>` and returns `void`, you must not use the `Memory<T>` instance after your method returns.**

This relates to the "lease" concept mentioned earlier. A void-returning method's lease on the `Memory<T>` instance begins when the method is entered, and it ends when the method exits. Consider the following example, which calls `Log` in a loop based on input from the console.

C#

```
using System;
using System.Buffers;

public class Example
{
    // implementation provided by third party
    static extern void Log(ReadOnlyMemory<char> message);

    // user code
    public static void Main()
    {
```

```

        using (var owner = MemoryPool<char>.Shared.Rent())
        {
            var memory = owner.Memory;
            var span = memory.Span;
            while (true)
            {
                string? s = Console.ReadLine();

                if (s is null)
                    return;

                int value = Int32.Parse(s);
                if (value < 0)
                    return;

                int numCharsWritten = ToBuffer(value, span);
                Log(memory.Slice(0, numCharsWritten));
            }
        }
    }

    private static int ToBuffer(int value, Span<char> span)
    {
        string strValue = value.ToString();
        int length = strValue.Length;
        strValue.AsSpan().CopyTo(span.Slice(0, length));
        return length;
    }
}

```

If `Log` is a fully synchronous method, this code will behave as expected because there is only one active consumer of the memory instance at any given time. But imagine instead that `Log` has this implementation.

C#

```

// !!! INCORRECT IMPLEMENTATION !!!
static void Log(ReadOnlyMemory<char> message)
{
    // Run in background so that we don't block the main thread while
    // performing IO.
    Task.Run(() =>
    {
        StreamWriter sw = File.AppendText(@"..\input-numbers.dat");
        sw.WriteLine(message);
    });
}

```

In this implementation, `Log` violates its lease because it still attempts to use the `Memory<T>` instance in the background after the original method has returned. The

`Main` method could mutate the buffer while `Log` attempts to read from it, which could result in data corruption.

There are several ways to resolve this:

- The `Log` method can return a `Task` instead of `void`, as the following implementation of the `Log` method does.

C#

```
// An acceptable implementation.
static Task Log(ReadOnlyMemory<char> message)
{
    // Run in the background so that we don't block the main thread
    // while performing IO.
    return Task.Run(() => {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(message);
        sw.Flush();
    });
}
```

- `Log` can instead be implemented as follows:

C#

```
// An acceptable implementation.
static void Log(ReadOnlyMemory<char> message)
{
    string defensiveCopy = message.ToString();
    // Run in the background so that we don't block the main thread
    // while performing IO.
    Task.Run(() =>
    {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(defensiveCopy);
        sw.Flush();
    });
}
```

**Rule #4: If your method accepts a `Memory<T>` and returns a `Task`, you must not use the `Memory<T>` instance after the `Task` transitions to a terminal state.**

This is just the `async` variant of Rule #3. The `Log` method from the earlier example can be written as follows to comply with this rule:

C#

```

// An acceptable implementation.
static Task Log(ReadOnlyMemory<char> message)
{
    // Run in the background so that we don't block the main thread while
    // performing IO.
    return Task.Run(() =>
    {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(message);
        sw.Flush();
    });
}

```

Here, "terminal state" means that the task transitions to a completed, faulted, or canceled state. In other words, "terminal state" means "anything that would cause await to throw or to continue execution."

This guidance applies to methods that return `Task`, `Task<TResult>`, `ValueTask<TResult>`, or any similar type.

**Rule #5: If your constructor accepts `Memory<T>` as a parameter, instance methods on the constructed object are assumed to be consumers of the `Memory<T>` instance.**

Consider the following example:

C#

```

class OddValueExtractor
{
    public OddValueExtractor(ReadOnlyMemory<int> input);
    public bool TryReadNextOddValue(out int value);
}

void PrintAllOddValues(ReadOnlyMemory<int> input)
{
    var extractor = new OddValueExtractor(input);
    while (extractor.TryReadNextOddValue(out int value))
    {
        Console.WriteLine(value);
    }
}

```

Here, the `OddValueExtractor` constructor accepts a `ReadOnlyMemory<int>` as a constructor parameter, so the constructor itself is a consumer of the `ReadOnlyMemory<int>` instance, and all instance methods on the returned value are also consumers of the original `ReadOnlyMemory<int>` instance. This means that

`TryReadNextOddValue` consumes the `ReadOnlyMemory<int>` instance, even though the instance isn't passed directly to the `TryReadNextOddValue` method.

**Rule #6: If you have a settable `Memory<T>`-typed property (or an equivalent instance method) on your type, instance methods on that object are assumed to be consumers of the `Memory<T>` instance.**

This is really just a variant of Rule #5. This rule exists because property setters or equivalent methods are assumed to capture and persist their inputs, so instance methods on the same object may utilize the captured state.

The following example triggers this rule:

C#

```
class Person
{
    // Settable property.
    public Memory<char> FirstName { get; set; }

    // alternatively, equivalent "setter" method
    public SetFirstName(Memory<char> value);

    // alternatively, a public settable field
    public Memory<char> FirstName;
}
```

**Rule #7: If you have an `IMemoryOwner<T>` reference, you must at some point dispose of it or transfer its ownership (but not both).**

Since a `Memory<T>` instance may be backed by either managed or unmanaged memory, the owner must call `Dispose` on `IMemoryOwner<T>` when work performed on the `Memory<T>` instance is complete. Alternatively, the owner may transfer ownership of the `IMemoryOwner<T>` instance to a different component, at which point the acquiring component becomes responsible for calling `Dispose` at the appropriate time (more on this later).

Failure to call the `Dispose` method on an `IMemoryOwner<T>` instance may lead to unmanaged memory leaks or other performance degradation.

This rule also applies to code that calls factory methods like `MemoryPool<T>.Rent`. The caller becomes the owner of the returned `IMemoryOwner<T>` and is responsible for disposing of the instance when finished.

**Rule #8: If you have an `IMemoryOwner<T>` parameter in your API surface, you are accepting ownership of that instance.**

Accepting an instance of this type signals that your component intends to take ownership of this instance. Your component becomes responsible for proper disposal according to Rule #7.

Any component that transfers ownership of the `IMemoryOwner<T>` instance to a different component should no longer use that instance after the method call completes.

### ⓘ Important

If your constructor accepts `IMemoryOwner<T>` as a parameter, its type should implement `IDisposable`, and your `Dispose` method should call `Dispose` on the `IMemoryOwner<T>` object.

**Rule #9: If you're wrapping a synchronous p/invoke method, your API should accept `Span<T>` as a parameter.**

According to Rule #1, `Span<T>` is generally the correct type to use for synchronous APIs. You can pin `Span<T>` instances via the `fixed` keyword, as in the following example.

C#

```
using System.Runtime.InteropServices;

[DllImport(...)]
private static extern unsafe int ExportedMethod(byte* pbData, int cbData);

public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        int retVal = ExportedMethod(pbData, data.Length);

        /* error checking retVal goes here */

        return retVal;
    }
}
```

In the previous example, `pbData` can be null if, for example, the input span is empty. If the exported method absolutely requires that `pbData` be non-null, even if `cbData` is 0, the method can be implemented as follows:

C#

```

public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        byte dummy = 0;
        int retVal = ExportedMethod((pbData != null) ? pbData : &dummy,
data.Length);

        /* error checking retVal goes here */

        return retVal;
    }
}

```

**Rule #10: If you're wrapping an asynchronous p/invoke method, your API should accept `Memory<T>` as a parameter.**

Since you cannot use the `fixed` keyword across asynchronous operations, you use the `Memory<T>.Pin` method to pin `Memory<T>` instances, regardless of the kind of contiguous memory the instance represents. The following example shows how to use this API to perform an asynchronous p/invoke call.

C#

```

using System.Runtime.InteropServices;

[UnmanagedFunctionPointer(...)]
private delegate void OnCompletedCallback(IntPtr state, int result);

[DllImport(...)]
private static extern unsafe int ExportedAsyncMethod(byte* pbData, int
cbData, IntPtr pState, IntPtr lpfnOnCompletedCallback);

private static readonly IntPtr _callbackPtr =
GetCompletionCallbackPointer();

public unsafe Task<int> ManagedWrapperAsync(Memory<byte> data)
{
    // setup
    var tcs = new TaskCompletionSource<int>();
    var state = new MyCompletedCallbackState
    {
        Tcs = tcs
    };
    var pState = (IntPtr)GCHandle.Alloc(state);

    var memoryHandle = data.Pin();
    state.MemoryHandle = memoryHandle;

    // make the call
    int result;

```

```

try
{
    result = ExportedAsyncMethod((byte*)memoryHandle.Pointer,
data.Length, pState, _callbackPtr);
}
catch
{
    ((GCHandle)pState).Free(); // cleanup since callback won't be
invoked
    memoryHandle.Dispose();
    throw;
}

if (result != PENDING)
{
    // Operation completed synchronously; invoke callback manually
    // for result processing and cleanup.
    MyCompletedCallbackImplementation(pState, result);
}

return tcs.Task;
}

private static void MyCompletedCallbackImplementation(IntPtr state, int
result)
{
    GCHandle handle = (GCHandle)state;
    var actualState = (MyCompletedCallbackState)(handle.Target);
    handle.Free();
    actualState.MemoryHandle.Dispose();

    /* error checking result goes here */

    if (error)
    {
        actualState.Tcs.SetException(...);
    }
    else
    {
        actualState.Tcs.SetResult(result);
    }
}

private static IntPtr GetCompletionCallbackPointer()
{
    OnCompletedCallback callback = MyCompletedCallbackImplementation;
    GCHandle.Alloc(callback); // keep alive for lifetime of application
    return Marshal.GetFunctionPointerForDelegate(callback);
}

private class MyCompletedCallbackState
{
    public TaskCompletionSource<int> Tcs;
    public MemoryHandle MemoryHandle;
}

```

## See also

- [System.Memory<T>](#)
- [System.Buffers.IMemoryOwner<T>](#)
- [System.Span<T>](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Use SIMD-accelerated numeric types

Article • 06/07/2022

SIMD (Single instruction, multiple data) provides hardware support for performing an operation on multiple pieces of data, in parallel, using a single instruction. In .NET, there's a set of SIMD-accelerated types under the [System.Numerics](#) namespace. SIMD operations can be parallelized at the hardware level. That increases the throughput of the vectorized computations, which are common in mathematical, scientific, and graphics apps.

## .NET SIMD-accelerated types

The .NET SIMD-accelerated types include the following types:

- The [Vector2](#), [Vector3](#), and [Vector4](#) types, which represent vectors with 2, 3, and 4 [Single](#) values.
- Two matrix types, [Matrix3x2](#), which represents a 3x2 matrix, and [Matrix4x4](#), which represents a 4x4 matrix of [Single](#) values.
- The [Plane](#) type, which represents a plane in three-dimensional space using [Single](#) values.
- The [Quaternion](#) type, which represents a vector that is used to encode three-dimensional physical rotations using [Single](#) values.
- The [Vector<T>](#) type, which represents a vector of a specified numeric type and provides a broad set of operators that benefit from SIMD support. The count of a [Vector<T>](#) instance is fixed for the lifetime of an application, but its value [Vector<T>.Count](#) depends on the CPU of the machine running the code.

### Note

The [Vector<T>](#) type is not included in the .NET Framework. You must install the [System.Numerics.Vectors](#)  NuGet package to get access to this type.

The SIMD-accelerated types are implemented in such a way that they can be used with non-SIMD-accelerated hardware or JIT compilers. To take advantage of SIMD instructions, your 64-bit apps must be run by the runtime that uses the [RyuJIT](#) compiler. A [RyuJIT](#) compiler is included in .NET Core and in .NET Framework 4.6 and later. SIMD support is only provided when targeting 64-bit processors.

# How to use SIMD?

Before executing custom SIMD algorithms, it's possible to check if the host machine supports SIMD by using [Vector.IsHardwareAccelerated](#), which returns a [Boolean](#). This doesn't guarantee that SIMD-acceleration is enabled for a specific type, but is an indicator that it's supported by some types.

## Simple Vectors

The most primitive SIMD-accelerated types in .NET are [Vector2](#), [Vector3](#), and [Vector4](#) types, which represent vectors with 2, 3, and 4 [Single](#) values. The example below uses [Vector2](#) to add two vectors.

C#

```
var v1 = new Vector2(0.1f, 0.2f);
var v2 = new Vector2(1.1f, 2.2f);
var vResult = v1 + v2;
```

It's also possible to use .NET vectors to calculate other mathematical properties of vectors such as [Dot product](#), [Transform](#), [Clamp](#) and so on.

C#

```
var v1 = new Vector2(0.1f, 0.2f);
var v2 = new Vector2(1.1f, 2.2f);
var vResult1 = Vector2.Dot(v1, v2);
var vResult2 = Vector2.Distance(v1, v2);
var vResult3 = Vector2.Clamp(v1, Vector2.Zero, Vector2.One);
```

## Matrix

[Matrix3x2](#), which represents a 3x2 matrix, and [Matrix4x4](#), which represents a 4x4 matrix. Can be used for matrix-related calculations. The example below demonstrates multiplication of a matrix to its correspondent transpose matrix using SIMD.

C#

```
var m1 = new Matrix4x4(
    1.1f, 1.2f, 1.3f, 1.4f,
    2.1f, 2.2f, 3.3f, 4.4f,
    3.1f, 3.2f, 3.3f, 3.4f,
    4.1f, 4.2f, 4.3f, 4.4f);
```

```
var m2 = Matrix4x4.Transpose(m1);
var mResult = Matrix4x4.Multiply(m1, m2);
```

## Vector<T>

The `Vector<T>` gives the ability to use longer vectors. The count of a `Vector<T>` instance is fixed, but its value `Vector<T>.Count` depends on the CPU of the machine running the code.

The following example demonstrates how to calculate the element-wise sum of two arrays using `Vector<T>`.

C#

```
double[] Sum(double[] left, double[] right)
{
    if (left is null)
    {
        throw new ArgumentNullException(nameof(left));
    }

    if (right is null)
    {
        throw new ArgumentNullException(nameof(right));
    }

    if (left.Length != right.Length)
    {
        throw new ArgumentException($"{nameof(left)} and {nameof(right)} are
not the same length");
    }

    int length = left.Length;
    double[] result = new double[length];

    // Get the number of elements that can't be processed in the vector
    // NOTE: Vector<T>.Count is a JIT time constant and will get optimized
    // accordingly
    int remaining = length % Vector<double>.Count;

    for (int i = 0; i < length - remaining; i += Vector<double>.Count)
    {
        var v1 = new Vector<double>(left, i);
        var v2 = new Vector<double>(right, i);
        (v1 + v2).CopyTo(result, i);
    }

    for (int i = length - remaining; i < length; i++)
    {
        result[i] = left[i] + right[i];
    }
}
```

```
    }

    return result;
}
```

## Remarks

SIMD is more likely to remove one bottleneck and expose the next, for example memory throughput. In general the performance benefit of using SIMD varies depending on the specific scenario, and in some cases it can even perform worse than simpler non-SIMD equivalent code.

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Value tuples

Article • 01/08/2024

A value tuple is a data structure that has a specific number and sequence of values. .NET provides the following built-in value tuple types:

- The `ValueTuple<T1>` structure represents a value tuple that has one element.
- The `ValueTuple<T1,T2>` structure represents a value tuple that has two elements.-
- The `ValueTuple<T1,T2,T3>` structure represents a value tuple that has three elements.
- The `ValueTuple<T1,T2,T3,T4>` structure represents a value tuple that has four elements.
- The `ValueTuple<T1,T2,T3,T4,T5>` structure represents a value tuple that has five elements.
- The `ValueTuple<T1,T2,T3,T4,T5,T6>` structure represents a value tuple that has six elements.
- The `ValueTuple<T1,T2,T3,T4,T5,T6,T7>` structure represents a value tuple that has seven elements.
- The `ValueTuple<T1,T2,T3,T4,T5,T6,T7,TRest>` structure represents a value tuple that has eight or more elements.

The value tuple types differ from the tuple types (such as `Tuple<T1,T2>`) as follows:

- They are structures (value types) rather than classes (reference types).
- Members such as `Item1` and `Item2` are fields rather than properties.
- Their fields are mutable rather than read-only.

The value tuple types provide the runtime implementation that supports [tuples in C#](#) and struct tuples in F#. In addition to creating a `ValueTuple<T1,T2>` instance by using language syntax, you can call the `Create` factory method.

## See also

- [Tuple types \(C# reference\)](#)

 Collaborate with us on  
GitHub

The source for this content can  
be found on GitHub, where you

.NET

## .NET feedback

.NET is an open source project.  
Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)