C# types and members

Article • 05/26/2023

As an object-oriented language, C# supports the concepts of encapsulation, inheritance, and polymorphism. A class may inherit directly from one parent class, and it may implement any number of interfaces. Methods that override virtual methods in a parent class require the `override` keyword as a way to avoid accidental redefinition. In C#, a struct is like a lightweight class; it's a stack-allocated type that can implement interfaces but doesn't support inheritance. C# provides `record class` and `record struct` types, which are types whose purpose is primarily storing data values.

All types are initialized through a *constructor*, a method responsible for initializing an instance. Two constructor declarations have unique behavior:

- A *parameterless constructor*, which initializes all fields to their default value.
- A *primary constructor*, which declares the required parameters for an instance of that type.

Classes and objects

Classes are the most fundamental of C#'s types. A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit. A class provides a definition for *instances* of the class, also known as *objects*. Classes support *inheritance* and *polymorphism*, mechanisms whereby *derived classes* can extend and specialize *base classes*.

New classes are created using class declarations. A class declaration starts with a header. The header specifies:

- The attributes and modifiers of the class
- The name of the class
- The base class (when inheriting from a base class)
- The interfaces implemented by the class.

The header is followed by the class body, which consists of a list of member declarations written between the delimiters `{` and `}`.

The following code shows a declaration of a simple class named `Point`:

C#

```
public class Point
{
    public int X { get; }
    public int Y { get; }
    public Point(int x, int y) => (X, Y) = (x, y);
}
```

Instances of classes are created using the `new` operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance. The following statements create two `Point` objects and store references to those objects in two variables:

C#

```
var p1 = new Point(0, 0);
var p2 = new Point(10, 20);
```

The memory occupied by an object is automatically reclaimed when the object is no longer reachable. It's not necessary or possible to explicitly deallocate objects in C#.

C#

```
var p1 = new Point(0, 0);
var p2 = new Point(10, 20);
```

Applications or tests for algorithms might need to create multiple `Point` objects. The following class generates a sequence of random points. The number of points is set by the *primary constructor* parameter. The primary constructor parameter `numberOfPoints` is in scope for all members of the class:

C#

```
public class PointFactory(int numberOfPoints)
{
    public IEnumerable<Point> CreatePoints()
    {
        var generator = new Random();
        for (int i = 0; i < numberOfPoints; i++)
        {
            yield return new Point(generator.Next(), generator.Next());
        }
    }
}
```

You can use the class as shown in the following code:

C#

```
var factory = new PointFactory(10);
foreach (var point in factory.CreatePoints())
{
    Console.WriteLine($"({point.X}, {point.Y})");
}
```

Type parameters

Generic classes define ***type parameters***. Type parameters are a list of type parameter names enclosed in angle brackets. Type parameters follow the class name. The type parameters can then be used in the body of the class declarations to define the members of the class. In the following example, the type parameters of `Pair` are `TFirst` and `TSecond`:

C#

```
public class Pair<TFirst, TSecond>
{
    public TFirst First { get; }
    public TSecond Second { get; }

    public Pair(TFirst first, TSecond second) =>
        (First, Second) = (first, second);
}
```

A class type that is declared to take type parameters is called a *generic class type*. Struct, interface, and delegate types can also be generic. When the generic class is used, type arguments must be provided for each of the type parameters:

C#

```
var pair = new Pair<int, string>(1, "two");
int i = pair.First;     //TFirst int
string s = pair.Second; //TSecond string
```

A generic type with type arguments provided, like `Pair<int,string>` above, is called a *constructed type*.

Base classes

A class declaration may specify a base class. Follow the class name and type parameters with a colon and the name of the base class. Omitting a base class specification is the same as deriving from type `object`. In the following example, the base class of `Point3D` is `Point`. From the first example, the base class of `Point` is `object`:

C#

```
public class Point3D : Point
{
    public int Z { get; set; }

    public Point3D(int x, int y, int z) : base(x, y)
    {
        Z = z;
    }
}
```

A class inherits the members of its base class. Inheritance means that a class implicitly contains almost all members of its base class. A class doesn't inherit the instance and static constructors, and the finalizer. A derived class can add new members to those members it inherits, but it can't remove the definition of an inherited member. In the previous example, `Point3D` inherits the `X` and `Y` members from `Point`, and every `Point3D` instance contains three properties, `X`, `Y`, and `Z`.

An implicit conversion exists from a class type to any of its base class types. A variable of a class type can reference an instance of that class or an instance of any derived class. For example, given the previous class declarations, a variable of type `Point` can reference either a `Point` or a `Point3D`:

C#

```
Point a = new(10, 20);
Point b = new Point3D(10, 20, 30);
```

Structs

Classes define types that support inheritance and polymorphism. They enable you to create sophisticated behaviors based on hierarchies of derived classes. By contrast, ***struct*** types are simpler types whose primary purpose is to store data values. Structs can't declare a base type; they implicitly derive from System.ValueType. You can't derive other `struct` types from a `struct` type. They're implicitly sealed.

C#

```
public struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y) => (X, Y) = (x, y);
}
```

Interfaces

An ***interface*** defines a contract that can be implemented by classes and structs. You define an *interface* to declare capabilities that are shared among distinct types. For example, the System.Collections.Generic.IEnumerable<T> interface defines a consistent way to traverse all the items in a collection, such as an array. An interface can contain methods, properties, events, and indexers. An interface typically doesn't provide implementations of the members it defines—it merely specifies the members that must be supplied by classes or structs that implement the interface.

Interfaces may employ ***multiple inheritance***. In the following example, the interface `IComboBox` inherits from both `ITextBox` and `IListBox`.

C#

```csharp
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox { }
```

Classes and structs can implement multiple interfaces. In the following example, the class `EditBox` implements both `IControl` and `IDataBound`.

C#

```csharp
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox : IControl, IDataBound
{
    public void Paint() { }
    public void Bind(Binder b) { }
}
```

When a class or struct implements a particular interface, instances of that class or struct can be implicitly converted to that interface type. For example

C#

```csharp
EditBox editBox = new();
IControl control = editBox;
IDataBound dataBound = editBox;
```

Enums

An *Enum* type defines a set of constant values. The following `enum` declares constants that define different root vegetables:

C#

```
public enum SomeRootVegetable
{
    HorseRadish,
    Radish,
    Turnip
}
```

You can also define an `enum` to be used in combination as flags. The following declaration declares a set of flags for the four seasons. Any combination of the seasons may be applied, including an `All` value that includes all seasons:

C#

```
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}
```

The following example shows declarations of both the preceding enums:

C#

```
var turnip = SomeRootVegetable.Turnip;

var spring = Seasons.Spring;
var startingOnEquinox = Seasons.Spring | Seasons.Autumn;
var theYear = Seasons.All;
```

Nullable types

Variables of any type may be declared as ***non-nullable*** or ***nullable***. A nullable variable can hold an additional `null` value, indicating no value. Nullable Value types (structs or enums) are represented by System.Nullable<T>. Non-nullable and Nullable Reference types are both represented by the underlying reference type. The distinction is represented by metadata read by the compiler and some libraries. The compiler provides warnings when nullable references are dereferenced without first checking their value against `null`. The compiler also provides warnings when non-nullable references are assigned a value that may be `null`. The following example declares a ***nullable int***, initializing it to `null`. Then, it sets the value to `5`. It demonstrates the same concept with a ***nullable string***. For more information, see nullable value types and nullable reference types.

C#

```csharp
int? optionalInt = default;
optionalInt = 5;
string? optionalText = default;
optionalText = "Hello World.";
```

Tuples

C# supports ***tuples***, which provides concise syntax to group multiple data elements in a lightweight data structure. You instantiate a tuple by declaring the types and names of the members between `(` and `)`, as shown in the following example:

C#

```csharp
(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
//Output:
//Sum of 3 elements is 4.5.
```

Tuples provide an alternative for data structure with multiple members, without using the building blocks described in the next article.

Previous Next

DateTime Struct

Reference

Definition

Namespace: System

Assembly: mscorlib.dll

Represents an instant in time, typically expressed as a date and time of day.

C#

```
[System.Serializable]
public struct DateTime : IComparable, IComparable<DateTime>, IConvertible,
IEquatable<DateTime>, IFormattable, System.Runtime.Serialization.ISerializable
```

Inheritance Object → ValueType → DateTime

Attributes SerializableAttribute

Implements IComparable , IComparable<DateTime> , IConvertible , IEquatable<DateTime> , IFormattable , ISerializable

Remarks

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**.

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

Formatting `DateTime` objects as strings

- 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

Parsing strings as `DateTime` objects

- 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

- Explore the resolution of date and time values
- Comparing for equality within a tolerance

Culture and calendars

- 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

- 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

Quick links to Remarks topics

This section contains topics for many common uses of the `DateTime` struct:

- Initialize a DateTime object
- DateTime values and their string representations
- Parse DateTime values from strings
- DateTime values
- DateTime operations
- DateTime resolution
- DateTime values and calendars
- Persist DateTime values
- DateTime vs. TimeSpan
- Compare for equality within tolerance
- COM interop considerations

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 are 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# from the docs repository on GitHub.

> ⓘ **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**.

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#

```csharp
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

```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#

```csharp
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#

```csharp
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString(System.Globalization.CultureInfo.CreateSpecificCultu
re("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.Now.Milliseconds` property changes only after the call to Thread.Sleep.

C#

```csharp
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, "O", 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:

- If the DateTime values represent single moments in time, convert them to UTC by calling the ToUniversalTime method.
- 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. That means two values must be expressed as the same number of ticks to be considered equal. 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#

```csharp
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. The .NET Framework 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 the .NET Framework assumes a base date of January, 1, 0001 C.E.

When only a time is passed from the .NET Framework 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 the .NET Framework, 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, the .NET Framework and COM preserve the date.

The behavior of the .NET Framework 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.

Constructors

DateTime(Int32, Int32, Int32)	Initializes a new instance of the DateTime structure to the specified year, month, and day.
DateTime(Int32, Int32, Int32, Calendar)	Initializes a new instance of the DateTime structure to the specified year, month, and day for the specified calendar.
DateTime(Int32, Int32, Int32, Int32, Int32, Int32)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, and second.

DateTime(Int32, Int32, Int32, Int32, Int32, Int32, Calendar)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, and second for the specified calendar.
DateTime(Int32, Int32, Int32, Int32, Int32, Int32, DateTimeKind)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, second, and Coordinated Universal Time (UTC) or local time.
DateTime(Int32, Int32, Int32, Int32, Int32, Int32, Int32)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, second, and millisecond.
DateTime(Int32, Int32, Int32, Int32, Int32, Int32, Int32, Calendar)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, second, and millisecond for the specified calendar.
DateTime(Int32, Int32, Int32, Int32, Int32, Int32, Int32, Calendar, DateTimeKind)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, second, millisecond, and Coordinated Universal Time (UTC) or local time for the specified calendar.
DateTime(Int32, Int32, Int32, Int32, Int32, Int32, Int32, DateTimeKind)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, second, millisecond, and Coordinated Universal Time (UTC) or local time.
DateTime(Int64)	Initializes a new instance of the DateTime structure to a specified number of ticks.
DateTime(Int64, DateTimeKind)	Initializes a new instance of the DateTime structure to a specified number of ticks and to Coordinated Universal Time (UTC) or local time.

Fields

MaxValue	Represents the largest possible value of DateTime. This field is read-only.
MinValue	Represents the smallest possible value of DateTime. This field is read-only.

Properties

Date	Gets the date component of this instance.
Day	Gets the day of the month represented by this instance.
DayOfWeek	Gets the day of the week represented by this instance.
DayOfYear	Gets the day of the year represented by this instance.
Hour	Gets the hour component of the date represented by this instance.
Kind	Gets a value that indicates whether the time represented by this instance is based on local time, Coordinated Universal Time (UTC), or neither.

Millisecond	Gets the milliseconds component of the date represented by this instance.
Minute	Gets the minute component of the date represented by this instance.
Month	Gets the month component of the date represented by this instance.
Now	Gets a DateTime object that is set to the current date and time on this computer, expressed as the local time.
Second	Gets the seconds component of the date represented by this instance.
Ticks	Gets the number of ticks that represent the date and time of this instance.
TimeOfDay	Gets the time of day for this instance.
Today	Gets the current date.
UtcNow	Gets a DateTime object that is set to the current date and time on this computer, expressed as the Coordinated Universal Time (UTC).
Year	Gets the year component of the date represented by this instance.

Methods

Add(TimeSpan)	Returns a new DateTime that adds the value of the specified TimeSpan to the value of this instance.
AddDays(Double)	Returns a new DateTime that adds the specified number of days to the value of this instance.
AddHours(Double)	Returns a new DateTime that adds the specified number of hours to the value of this instance.
AddMilliseconds(Double)	Returns a new DateTime that adds the specified number of milliseconds to the value of this instance.
AddMinutes(Double)	Returns a new DateTime that adds the specified number of minutes to the value of this instance.
AddMonths(Int32)	Returns a new DateTime that adds the specified number of months to the value of this instance.
AddSeconds(Double)	Returns a new DateTime that adds the specified number of seconds to the value of this instance.
AddTicks(Int64)	Returns a new DateTime that adds the specified number of ticks to the value of this instance.
AddYears(Int32)	Returns a new DateTime that adds the specified number of years to the value of this instance.

Compare(DateTime, DateTime)	Compares two instances of DateTime and returns an integer that indicates whether the first instance is earlier than, the same as, or later than the second instance.
CompareTo(DateTime)	Compares the value of this instance to a specified DateTime value and returns an integer that indicates whether this instance is earlier than, the same as, or later than the specified DateTime value.
CompareTo(Object)	Compares the value of this instance to a specified object that contains a specified DateTime value, and returns an integer that indicates whether this instance is earlier than, the same as, or later than the specified DateTime value.
DaysInMonth(Int32, Int32)	Returns the number of days in the specified month and year.
Equals(DateTime)	Returns a value indicating whether the value of this instance is equal to the value of the specified DateTime instance.
Equals(DateTime, DateTime)	Returns a value indicating whether two DateTime instances have the same date and time value.
Equals(Object)	Returns a value indicating whether this instance is equal to a specified object.
FromBinary(Int64)	Deserializes a 64-bit binary value and recreates an original serialized DateTime object.
FromFileTime(Int64)	Converts the specified Windows file time to an equivalent local time.
FromFileTimeUtc(Int64)	Converts the specified Windows file time to an equivalent UTC time.
FromOADate(Double)	Returns a DateTime equivalent to the specified OLE Automation Date.
GetDateTimeFormats()	Converts the value of this instance to all the string representations supported by the standard date and time format specifiers.
GetDateTimeFormats(Char)	Converts the value of this instance to all the string representations supported by the specified standard date and time format specifier.
GetDateTimeFormats(Char, IFormatProvider)	Converts the value of this instance to all the string representations supported by the specified standard date and time format specifier and culture-specific formatting information.
GetDateTimeFormats(IFormatProvider)	Converts the value of this instance to all the string representations supported by the standard date and time format specifiers and the specified culture-specific formatting information.
GetHashCode()	Returns the hash code for this instance.
GetTypeCode()	Returns the TypeCode for value type DateTime.
IsDaylightSavingTime()	Indicates whether this instance of DateTime is within the daylight saving time range for the current time zone.
IsLeapYear(Int32)	Returns an indication whether the specified year is a leap year.

Parse(String)	Converts the string representation of a date and time to its DateTime equivalent by using the conventions of the current culture.
Parse(String, IFormatProvider)	Converts the string representation of a date and time to its DateTime equivalent by using culture-specific format information.
Parse(String, IFormatProvider, DateTimeStyles)	Converts the string representation of a date and time to its DateTime equivalent by using culture-specific format information and a formatting style.
ParseExact(String, String, IFormat Provider)	Converts the specified string representation of a date and time to its DateTime equivalent using the specified format and culture-specific format information. The format of the string representation must match the specified format exactly.
ParseExact(String, String, IFormat Provider, DateTimeStyles)	Converts the specified string representation of a date and time to its DateTime equivalent using the specified format, culture-specific format information, and style. The format of the string representation must match the specified format exactly or an exception is thrown.
ParseExact(String, String[], IFormat Provider, DateTimeStyles)	Converts the specified string representation of a date and time to its DateTime equivalent using the specified array of formats, culture-specific format information, and style. The format of the string representation must match at least one of the specified formats exactly or an exception is thrown.
SpecifyKind(DateTime, DateTime Kind)	Creates a new DateTime object that has the same number of ticks as the specified DateTime, but is designated as either local time, Coordinated Universal Time (UTC), or neither, as indicated by the specified DateTimeKind value.
Subtract(DateTime)	Returns a new TimeSpan that subtracts the specified date and time from the value of this instance.
Subtract(TimeSpan)	Returns a new DateTime that subtracts the specified duration from the value of this instance.
ToBinary()	Serializes the current DateTime object to a 64-bit binary value that subsequently can be used to recreate the DateTime object.
ToFileTime()	Converts the value of the current DateTime object to a Windows file time.
ToFileTimeUtc()	Converts the value of the current DateTime object to a Windows file time.
ToLocalTime()	Converts the value of the current DateTime object to local time.
ToLongDateString()	Converts the value of the current DateTime object to its equivalent long date string representation.
ToLongTimeString()	Converts the value of the current DateTime object to its equivalent long time string representation.

ToOADate()	Converts the value of this instance to the equivalent OLE Automation date.
ToShortDateString()	Converts the value of the current DateTime object to its equivalent short date string representation.
ToShortTimeString()	Converts the value of the current DateTime object to its equivalent short time string representation.
ToString()	Converts the value of the current DateTime object to its equivalent string representation using the formatting conventions of the current culture.
ToString(IFormatProvider)	Converts the value of the current DateTime object to its equivalent string representation using the specified culture-specific format information.
ToString(String)	Converts the value of the current DateTime object to its equivalent string representation using the specified format and the formatting conventions of the current culture.
ToString(String, IFormatProvider)	Converts the value of the current DateTime object to its equivalent string representation using the specified format and culture-specific format information.
ToUniversalTime()	Converts the value of the current DateTime object to Coordinated Universal Time (UTC).
TryParse(String, DateTime)	Converts the specified string representation of a date and time to its DateTime equivalent and returns a value that indicates whether the conversion succeeded.
TryParse(String, IFormatProvider, DateTimeStyles, DateTime)	Converts the specified string representation of a date and time to its DateTime equivalent using the specified culture-specific format information and formatting style, and returns a value that indicates whether the conversion succeeded.
TryParseExact(String, String, IFormatProvider, DateTimeStyles, DateTime)	Converts the specified string representation of a date and time to its DateTime equivalent using the specified format, culture-specific format information, and style. The format of the string representation must match the specified format exactly. The method returns a value that indicates whether the conversion succeeded.
TryParseExact(String, String[], IFormatProvider, DateTimeStyles, DateTime)	Converts the specified string representation of a date and time to its DateTime equivalent using the specified array of formats, culture-specific format information, and style. The format of the string representation must match at least one of the specified formats exactly. The method returns a value that indicates whether the conversion succeeded.

Operators

Addition(DateTime, TimeSpan)	Adds a specified time interval to a specified date and time, yielding a new date and time.

Equality(DateTime, DateTime)	Determines whether two specified instances of DateTime are equal.
GreaterThan(DateTime, DateTime)	Determines whether one specified DateTime is later than another specified DateTime.
GreaterThanOrEqual(DateTime, DateTime)	Determines whether one specified DateTime represents a date and time that is the same as or later than another specified DateTime.
Inequality(DateTime, DateTime)	Determines whether two specified instances of DateTime are not equal.
LessThan(DateTime, DateTime)	Determines whether one specified DateTime is earlier than another specified DateTime.
LessThanOrEqual(DateTime, DateTime)	Determines whether one specified DateTime represents a date and time that is the same as or earlier than another specified DateTime.
Subtraction(DateTime, DateTime)	Subtracts a specified date and time from another specified date and time and returns a time interval.
Subtraction(DateTime, TimeSpan)	Subtracts a specified time interval from a specified date and time and returns a new date and time.

Explicit Interface Implementations

IConvertible.ToBoolean(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToByte(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToChar(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToDateTime(IFormatProvider)	Returns the current DateTime object.
IConvertible.ToDecimal(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToDouble(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToInt16(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToInt32(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToInt64(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToSByte(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.

IConvertible.ToSingle(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToType(Type, IFormatProvider)	Converts the current DateTime object to an object of a specified type.
IConvertible.ToUInt16(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToUInt32(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToUInt64(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
ISerializable.GetObjectData(SerializationInfo, StreamingContext)	Populates a SerializationInfo object with the data needed to serialize the current DateTime object.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8
.NET Framework	1.1, 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0
Xamarin.iOS	10.8
Xamarin.Mac	3.0

Thread Safety

All members of this type are thread safe. Members that appear to modify instance state actually return a new instance initialized with the new value. As with any other type, reading and writing to a shared variable that contains an instance of this type must be protected by a lock to guarantee thread safety.

See also

- DateTimeOffset
- TimeSpan
- Calendar
- GetUtcOffset(DateTime)

- TimeZoneInfo
- Choosing Between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo
- Working with Calendars
- Sample: .NET Core WinForms Formatting Utility (C#)
- Sample: .NET Core WinForms Formatting Utility (Visual Basic)

DateTime Struct

Reference

Definition

Namespace: System

Assembly: mscorlib.dll

Represents an instant in time, typically expressed as a date and time of day.

C#

```
[System.Serializable]
public struct DateTime : IComparable, IComparable<DateTime>, IConvertible,
IEquatable<DateTime>, IFormattable, System.Runtime.Serialization.ISerializable
```

Inheritance Object → ValueType → DateTime

Attributes SerializableAttribute

Implements IComparable , IComparable<DateTime> , IConvertible , IEquatable<DateTime> , IFormattable , ISerializable

Remarks

ⓘ **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**.

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

Formatting `DateTime` objects as strings

- 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

Parsing strings as `DateTime` objects

- 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

- Explore the resolution of date and time values
- Comparing for equality within a tolerance

Culture and calendars

- 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

- 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

Quick links to Remarks topics

This section contains topics for many common uses of the `DateTime` struct:

- Initialize a DateTime object
- DateTime values and their string representations
- Parse DateTime values from strings
- DateTime values
- DateTime operations
- DateTime resolution
- DateTime values and calendars
- Persist DateTime values
- DateTime vs. TimeSpan
- Compare for equality within tolerance
- COM interop considerations

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 are 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# from the docs repository on GitHub.

> **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**.

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#

```csharp
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#

```csharp
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#

```csharp
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#

```csharp
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

```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#

```csharp
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#

```csharp
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString(System.Globalization.CultureInfo.CreateSpecificCultu
re("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.Now.Milliseconds` property changes only after the call to Thread.Sleep.

C#

```csharp
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, "O", 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:

- If the DateTime values represent single moments in time, convert them to UTC by calling the ToUniversalTime method.
- 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#

```csharp
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. That means two values must be expressed as the same number of ticks to be considered equal. 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. The .NET Framework 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 the .NET Framework assumes a base date of January, 1, 0001 C.E.

When only a time is passed from the .NET Framework 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 the .NET Framework, 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, the .NET Framework and COM preserve the date.

The behavior of the .NET Framework 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.

Constructors

DateTime(Int32, Int32, Int32)	Initializes a new instance of the DateTime structure to the specified year, month, and day.
DateTime(Int32, Int32, Int32, Calendar)	Initializes a new instance of the DateTime structure to the specified year, month, and day for the specified calendar.
DateTime(Int32, Int32, Int32, Int32, Int32, Int32)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, and second.

DateTime(Int32, Int32, Int32, Int32, Int32, Int32, Calendar)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, and second for the specified calendar.
DateTime(Int32, Int32, Int32, Int32, Int32, Int32, DateTimeKind)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, second, and Coordinated Universal Time (UTC) or local time.
DateTime(Int32, Int32, Int32, Int32, Int32, Int32, Int32)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, second, and millisecond.
DateTime(Int32, Int32, Int32, Int32, Int32, Int32, Int32, Calendar)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, second, and millisecond for the specified calendar.
DateTime(Int32, Int32, Int32, Int32, Int32, Int32, Int32, Calendar, DateTimeKind)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, second, millisecond, and Coordinated Universal Time (UTC) or local time for the specified calendar.
DateTime(Int32, Int32, Int32, Int32, Int32, Int32, Int32, DateTimeKind)	Initializes a new instance of the DateTime structure to the specified year, month, day, hour, minute, second, millisecond, and Coordinated Universal Time (UTC) or local time.
DateTime(Int64)	Initializes a new instance of the DateTime structure to a specified number of ticks.
DateTime(Int64, DateTimeKind)	Initializes a new instance of the DateTime structure to a specified number of ticks and to Coordinated Universal Time (UTC) or local time.

Fields

MaxValue	Represents the largest possible value of DateTime. This field is read-only.
MinValue	Represents the smallest possible value of DateTime. This field is read-only.

Properties

Date	Gets the date component of this instance.
Day	Gets the day of the month represented by this instance.
DayOfWeek	Gets the day of the week represented by this instance.
DayOfYear	Gets the day of the year represented by this instance.
Hour	Gets the hour component of the date represented by this instance.
Kind	Gets a value that indicates whether the time represented by this instance is based on local time, Coordinated Universal Time (UTC), or neither.

Millisecond	Gets the milliseconds component of the date represented by this instance.
Minute	Gets the minute component of the date represented by this instance.
Month	Gets the month component of the date represented by this instance.
Now	Gets a DateTime object that is set to the current date and time on this computer, expressed as the local time.
Second	Gets the seconds component of the date represented by this instance.
Ticks	Gets the number of ticks that represent the date and time of this instance.
TimeOfDay	Gets the time of day for this instance.
Today	Gets the current date.
UtcNow	Gets a DateTime object that is set to the current date and time on this computer, expressed as the Coordinated Universal Time (UTC).
Year	Gets the year component of the date represented by this instance.

Methods

Add(TimeSpan)	Returns a new DateTime that adds the value of the specified TimeSpan to the value of this instance.
AddDays(Double)	Returns a new DateTime that adds the specified number of days to the value of this instance.
AddHours(Double)	Returns a new DateTime that adds the specified number of hours to the value of this instance.
AddMilliseconds(Double)	Returns a new DateTime that adds the specified number of milliseconds to the value of this instance.
AddMinutes(Double)	Returns a new DateTime that adds the specified number of minutes to the value of this instance.
AddMonths(Int32)	Returns a new DateTime that adds the specified number of months to the value of this instance.
AddSeconds(Double)	Returns a new DateTime that adds the specified number of seconds to the value of this instance.
AddTicks(Int64)	Returns a new DateTime that adds the specified number of ticks to the value of this instance.
AddYears(Int32)	Returns a new DateTime that adds the specified number of years to the value of this instance.

Compare(DateTime, DateTime)	Compares two instances of DateTime and returns an integer that indicates whether the first instance is earlier than, the same as, or later than the second instance.
CompareTo(DateTime)	Compares the value of this instance to a specified DateTime value and returns an integer that indicates whether this instance is earlier than, the same as, or later than the specified DateTime value.
CompareTo(Object)	Compares the value of this instance to a specified object that contains a specified DateTime value, and returns an integer that indicates whether this instance is earlier than, the same as, or later than the specified DateTime value.
DaysInMonth(Int32, Int32)	Returns the number of days in the specified month and year.
Equals(DateTime)	Returns a value indicating whether the value of this instance is equal to the value of the specified DateTime instance.
Equals(DateTime, DateTime)	Returns a value indicating whether two DateTime instances have the same date and time value.
Equals(Object)	Returns a value indicating whether this instance is equal to a specified object.
FromBinary(Int64)	Deserializes a 64-bit binary value and recreates an original serialized DateTime object.
FromFileTime(Int64)	Converts the specified Windows file time to an equivalent local time.
FromFileTimeUtc(Int64)	Converts the specified Windows file time to an equivalent UTC time.
FromOADate(Double)	Returns a DateTime equivalent to the specified OLE Automation Date.
GetDateTimeFormats()	Converts the value of this instance to all the string representations supported by the standard date and time format specifiers.
GetDateTimeFormats(Char)	Converts the value of this instance to all the string representations supported by the specified standard date and time format specifier.
GetDateTimeFormats(Char, IFormatProvider)	Converts the value of this instance to all the string representations supported by the specified standard date and time format specifier and culture-specific formatting information.
GetDateTimeFormats(IFormatProvider)	Converts the value of this instance to all the string representations supported by the standard date and time format specifiers and the specified culture-specific formatting information.
GetHashCode()	Returns the hash code for this instance.
GetTypeCode()	Returns the TypeCode for value type DateTime.
IsDaylightSavingTime()	Indicates whether this instance of DateTime is within the daylight saving time range for the current time zone.
IsLeapYear(Int32)	Returns an indication whether the specified year is a leap year.

Parse(String)	Converts the string representation of a date and time to its DateTime equivalent by using the conventions of the current culture.
Parse(String, IFormatProvider)	Converts the string representation of a date and time to its DateTime equivalent by using culture-specific format information.
Parse(String, IFormatProvider, DateTimeStyles)	Converts the string representation of a date and time to its DateTime equivalent by using culture-specific format information and a formatting style.
ParseExact(String, String, IFormatProvider)	Converts the specified string representation of a date and time to its DateTime equivalent using the specified format and culture-specific format information. The format of the string representation must match the specified format exactly.
ParseExact(String, String, IFormatProvider, DateTimeStyles)	Converts the specified string representation of a date and time to its DateTime equivalent using the specified format, culture-specific format information, and style. The format of the string representation must match the specified format exactly or an exception is thrown.
ParseExact(String, String[], IFormatProvider, DateTimeStyles)	Converts the specified string representation of a date and time to its DateTime equivalent using the specified array of formats, culture-specific format information, and style. The format of the string representation must match at least one of the specified formats exactly or an exception is thrown.
SpecifyKind(DateTime, DateTimeKind)	Creates a new DateTime object that has the same number of ticks as the specified DateTime, but is designated as either local time, Coordinated Universal Time (UTC), or neither, as indicated by the specified DateTimeKind value.
Subtract(DateTime)	Returns a new TimeSpan that subtracts the specified date and time from the value of this instance.
Subtract(TimeSpan)	Returns a new DateTime that subtracts the specified duration from the value of this instance.
ToBinary()	Serializes the current DateTime object to a 64-bit binary value that subsequently can be used to recreate the DateTime object.
ToFileTime()	Converts the value of the current DateTime object to a Windows file time.
ToFileTimeUtc()	Converts the value of the current DateTime object to a Windows file time.
ToLocalTime()	Converts the value of the current DateTime object to local time.
ToLongDateString()	Converts the value of the current DateTime object to its equivalent long date string representation.
ToLongTimeString()	Converts the value of the current DateTime object to its equivalent long time string representation.

ToOADate()	Converts the value of this instance to the equivalent OLE Automation date.
ToShortDateString()	Converts the value of the current DateTime object to its equivalent short date string representation.
ToShortTimeString()	Converts the value of the current DateTime object to its equivalent short time string representation.
ToString()	Converts the value of the current DateTime object to its equivalent string representation using the formatting conventions of the current culture.
ToString(IFormatProvider)	Converts the value of the current DateTime object to its equivalent string representation using the specified culture-specific format information.
ToString(String)	Converts the value of the current DateTime object to its equivalent string representation using the specified format and the formatting conventions of the current culture.
ToString(String, IFormatProvider)	Converts the value of the current DateTime object to its equivalent string representation using the specified format and culture-specific format information.
ToUniversalTime()	Converts the value of the current DateTime object to Coordinated Universal Time (UTC).
TryParse(String, DateTime)	Converts the specified string representation of a date and time to its DateTime equivalent and returns a value that indicates whether the conversion succeeded.
TryParse(String, IFormatProvider, DateTimeStyles, DateTime)	Converts the specified string representation of a date and time to its DateTime equivalent using the specified culture-specific format information and formatting style, and returns a value that indicates whether the conversion succeeded.
TryParseExact(String, String, IFormatProvider, DateTimeStyles, DateTime)	Converts the specified string representation of a date and time to its DateTime equivalent using the specified format, culture-specific format information, and style. The format of the string representation must match the specified format exactly. The method returns a value that indicates whether the conversion succeeded.
TryParseExact(String, String[], IFormatProvider, DateTimeStyles, DateTime)	Converts the specified string representation of a date and time to its DateTime equivalent using the specified array of formats, culture-specific format information, and style. The format of the string representation must match at least one of the specified formats exactly. The method returns a value that indicates whether the conversion succeeded.

Operators

Addition(DateTime, TimeSpan)	Adds a specified time interval to a specified date and time, yielding a new date and time.

Equality(DateTime, DateTime)	Determines whether two specified instances of DateTime are equal.
GreaterThan(DateTime, DateTime)	Determines whether one specified DateTime is later than another specified DateTime.
GreaterThanOrEqual(DateTime, DateTime)	Determines whether one specified DateTime represents a date and time that is the same as or later than another specified DateTime.
Inequality(DateTime, DateTime)	Determines whether two specified instances of DateTime are not equal.
LessThan(DateTime, DateTime)	Determines whether one specified DateTime is earlier than another specified DateTime.
LessThanOrEqual(DateTime, DateTime)	Determines whether one specified DateTime represents a date and time that is the same as or earlier than another specified DateTime.
Subtraction(DateTime, DateTime)	Subtracts a specified date and time from another specified date and time and returns a time interval.
Subtraction(DateTime, TimeSpan)	Subtracts a specified time interval from a specified date and time and returns a new date and time.

Explicit Interface Implementations

IConvertible.ToBoolean(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToByte(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToChar(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToDateTime(IFormatProvider)	Returns the current DateTime object.
IConvertible.ToDecimal(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToDouble(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToInt16(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToInt32(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToInt64(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToSByte(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.

IConvertible.ToSingle(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToType(Type, IFormatProvider)	Converts the current DateTime object to an object of a specified type.
IConvertible.ToUInt16(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToUInt32(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
IConvertible.ToUInt64(IFormatProvider)	This conversion is not supported. Attempting to use this method throws an InvalidCastException.
ISerializable.GetObjectData(SerializationInfo, StreamingContext)	Populates a SerializationInfo object with the data needed to serialize the current DateTime object.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8
.NET Framework	1.1, 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0
Xamarin.iOS	10.8
Xamarin.Mac	3.0

Thread Safety

All members of this type are thread safe. Members that appear to modify instance state actually return a new instance initialized with the new value. As with any other type, reading and writing to a shared variable that contains an instance of this type must be protected by a lock to guarantee thread safety.

See also

- DateTimeOffset
- TimeSpan
- Calendar
- GetUtcOffset(DateTime)

- TimeZoneInfo
- Choosing Between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo
- Working with Calendars
- Sample: .NET Core WinForms Formatting Utility (C#)
- Sample: .NET Core WinForms Formatting Utility (Visual Basic)