Contents

Working with Base Types in .NET

5/2/2018 • 2 minutes to read • Edit Online

This section describes .NET base type operations, including formatting, conversion, and common operations.

In This Section

Type Conversion in the .NET Framework
Describes how to convert from one type to another.

Formatting Types
Describes how to format strings using the string format specifiers.

Manipulating Strings
Describes how to manipulate and format strings.

Parsing Strings
Describes how to convert strings into .NET Framework types.

Related Sections

Common Type System
Describes types used by the .NET Framework.

Dates, Times, and Time Zones
Describes how to work with time zones and time zone conversions in time zone-aware applications.

Common Type System

9/6/2018 • 23 minutes to read • Edit Online

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

- Establishes a framework that helps enable cross-language integration, type safety, and high-performance code execution.
- Provides an object-oriented model that supports the complete implementation of many programming languages.
- Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.
- Provides a library that contains the primitive data types (such as Boolean, Byte, Char, Int32, and UInt64) used in application development.

This topic contains the following sections:

- Types in .NET
- Type Definitions
- Type Members
- Characteristics of Type Members

Types in .NET

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

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

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

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

- Classes
- Structures
- Enumerations
- Interfaces
- Delegates

Classes

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

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

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

NOTE

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

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

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

Structures

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

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

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

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

Enumerations

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

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

The following additional restrictions apply to enumerations:

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

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

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

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

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

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

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

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

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

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

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

```
Imports System.Collections.Generic

' A traditional enumeration of some root vegetables.
Public Enum SomeRootVegetables
   HorseRadish
   Radish
   Turnip
End Enum

' A bit field or flag enumeration of harvesting seasons.
<Flags()> Public Enum Seasons
   None = 0
   Summer = 1
   Autumn = 2
   Winter = 4
   Spring = 8
   All = Summer Or Autumn Or Winter Or Spring
End Enum

' Entry point.
Public Class Example
   Public Shared Sub Main()
      ' Hash table of when vegetables are available.
      Dim AvailableIn As New Dictionary(Of SomeRootVegetables, Seasons)()

      AvailableIn(SomeRootVegetables.HorseRadish) = Seasons.All
      AvailableIn(SomeRootVegetables.Radish) = Seasons.Spring
      AvailableIn(SomeRootVegetables.Turnip) = Seasons.Spring Or _
                                              Seasons.Autumn

      ' Array of the seasons, using the enumeration.
      Dim theSeasons() As Seasons = {Seasons.Summer, Seasons.Autumn, _
                                    Seasons.Winter, Seasons.Spring}

      ' Print information of what vegetables are available each season.
      For Each season As Seasons In theSeasons
         Console.WriteLine(String.Format( _
              "The following root vegetables are harvested in {0}:", _
              season.ToString("G")))
         For Each item As KeyValuePair(Of SomeRootVegetables, Seasons) In AvailableIn
            ' A bitwise comparison.
            If(CType(item.Value, Seasons) And season) > 0 Then
               Console.WriteLine("  " + _
                     CType(item.Key, SomeRootVegetables).ToString("G"))
            End If
         Next
      Next
   End Sub
End Class
' The example displays the following output:
'    The following root vegetables are harvested in Summer:
'      HorseRadish
'    The following root vegetables are harvested in Autumn:
'      Turnip
'      HorseRadish
'    The following root vegetables are harvested in Winter:
'      HorseRadish
'    The following root vegetables are harvested in Spring:
'      Turnip
'      Radish
'      HorseRadish
```

Interfaces

An interface defines a contract that specifies a "can do" relationship or a "has a" relationship. Interfaces are often used to implement functionality, such as comparing and sorting (the IComparable and IComparable<T>

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

The following restrictions apply to interfaces:

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

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

Delegates

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

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

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

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

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

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

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

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

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

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

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

Type Definitions

A type definition includes the following:

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

Attributes

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

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

Type Accessibility

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

ACCESSIBILITY	DESCRIPTION
public	The type is accessible by all assemblies.
assembly	The type is accessible only from within its assembly.

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

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

- If the declared accessibility of `M` is `public`, the accessibility domain of `M` is the accessibility domain of `T`.

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

Type Names

The common type system imposes only two restrictions on names:

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

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

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

Base Types and Interfaces

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

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

Type Members

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

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

Fields

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

- Constants, whose value is assigned at design time. These are static members of a class, although they are not defined using the `static` (`Shared` in Visual Basic) keyword.

- Read-only variables, whose values can be assigned in the class constructor.

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

```
using System;

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

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

public class Example
{
   public static void Main()
   {
      Constants con = new Constants(new DateTime(1974, 8, 18));
      Console.Write(Constants.Pi + "\n");
      Console.Write(con.BirthDate.ToString("d") + "\n");
   }
}
// The example displays the following output if run on a system whose current
// culture is en-US:
//    3.1416
//    8/18/1974
```

```
Public Class Constants
   Public Const Pi As Double = 3.1416
   Public ReadOnly BirthDate As Date

   Public Sub New(birthDate As Date)
      Me.BirthDate = birthDate
   End Sub
End Class

Public Module Example
   Public Sub Main()
      Dim con As New Constants(#8/18/1974#)
      Console.WriteLine(Constants.Pi.ToString())
      Console.WriteLine(con.BirthDate.ToString("d"))
   End Sub
End Module
' The example displays the following output if run on a system whose current
' culture is en-US:
'    3.1416
'    8/18/1974
```

Properties

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

```
using System;

public class Person
{
   private int m_Age;

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

```
Public Class Person
   Private m_Age As Integer

   Public Property Age As Integer
      Get
         Return m_Age
      End Get
      Set
         If value < 0 Or value > 125 Then
            Throw New ArgumentOutOfRangeException("The value of the Age property must be between 0 and 125.")
         Else
            m_Age = value
         End If
      End Set
   End Property
End Class
```

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

Methods

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

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

Constructors

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

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

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

Events

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

Nested Types

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

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

Characteristics of Type Members

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

CHARACTERISTIC	CAN APPLY TO	DESCRIPTION
abstract	Methods, properties, and events	The type does not supply the method's implementation. Types that inherit or implement abstract methods must supply an implementation for the method. The only exception is when the derived type is itself an abstract type. All abstract methods are virtual.

CHARACTERISTIC	CAN APPLY TO	DESCRIPTION
private, family, assembly, family and assembly, family or assembly, or public	All	Defines the accessibility of the member: private Accessible only from within the same type as the member, or within a nested type. family Accessible from within the same type as the member, and from derived types that inherit from it. assembly Accessible only in the assembly in which the type is defined. family and assembly Accessible only from types that qualify for both family and assembly access. family or assembly Accessible only from types that qualify for either family or assembly access. public Accessible from any type.
final	Methods, properties, and events	The virtual method cannot be overridden in a derived type.
initialize-only	Fields	The value can only be initialized, and cannot be written after initialization.
instance	Fields, methods, properties, and events	If a member is not marked as `static` (C# and C++), `Shared` (Visual Basic), `virtual` (C# and C++), or `Overridable` (Visual Basic), it is an instance member (there is no instance keyword). There will be as many copies of such members in memory as there are objects that use it.
literal	Fields	The value assigned to the field is a fixed value, known at compile time, of a built-in value type. Literal fields are sometimes referred to as constants.

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

Overloading

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

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

Inheriting, Overriding, and Hiding Members

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

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

marked as `final` and the new method is at least as accessible as the virtual method.

See also

- .NET API Browser
- Common Language Runtime
- Type Conversion in .NET

Type Conversion in the .NET Framework

9/6/2018 • 29 minutes to read • Edit Online

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

The .NET Framework automatically supports the following conversions:

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

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

- The `Implicit` operator, which defines the available widening conversions between types. For more information, see the Implicit Conversion with the Implicit Operator section.
- The `Explicit` operator, which defines the available narrowing conversions between types. For more information, see the Explicit Conversion with the Explicit Operator section.
- The IConvertible interface, which defines conversions to each of the base .NET Framework data types. For more information, see The IConvertible Interface section.
- The Convert class, which provides a set of methods that implement the methods in the IConvertible interface. For more information, see The Convert Class section.
- The TypeConverter class, which is a base class that can be extended to support the conversion of a specified type to any other type. For more information, see The TypeConverter Class section.

Implicit Conversion with the Implicit Operator

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

NOTE

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

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

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

decimal decimalValue;

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

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

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

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

decimalValue = ulongValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  longValue.GetType().Name, decimalValue);
// The example displays the following output:
//    After assigning a Byte value, the Decimal value is 16.
//    After assigning a Int16 value, the Decimal value is -1024.
//    After assigning a Int32 value, the Decimal value is -1034000.
//    After assigning a Int64 value, the Decimal value is 1152921504606846976.
//    After assigning a Int64 value, the Decimal value is 18446744073709551615.
```

```
Dim byteValue As Byte = 16
Dim shortValue As Short = -1024
Dim intValue As Integer = -1034000
Dim longValue As Long = CLng(1024^6)
Dim ulongValue As ULong = ULong.MaxValue

Dim decimalValue As Decimal

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

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

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

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

decimalValue = ulongValue
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  longValue.GetType().Name, decimalValue)
' The example displays the following output:
'    After assigning a Byte value, the Decimal value is 16.
'    After assigning a Int16 value, the Decimal value is -1024.
'    After assigning a Int32 value, the Decimal value is -1034000.
'    After assigning a Int64 value, the Decimal value is 1152921504606846976.
'    After assigning a Int64 value, the Decimal value is 18446744073709551615.
```

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

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

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

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

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

```
Public Structure ByteWithSign
   Private signValue As SByte
   Private value As Byte

   Public Overloads Shared Widening Operator CType(value As SByte) As ByteWithSign
      Dim newValue As ByteWithSign
      newValue.signValue = CSByte(Math.Sign(value))
      newValue.value = CByte(Math.Abs(value))
      Return newValue
   End Operator

   Public Overloads Shared Widening Operator CType(value As Byte) As ByteWithSign
      Dim NewValue As ByteWithSign
      newValue.signValue = 1
      newValue.value = value
      Return newValue
   End Operator

   Public Overrides Function ToString() As String
      Return (signValue * value).ToString()
   End Function
End Structure
```

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

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

```
Dim sbyteValue As SByte = -120
Dim value As ByteWithSign = sbyteValue
Console.WriteLine(value.ToString())
value = Byte.MaxValue
Console.WriteLine(value.ToString())
' The example displays the following output:
'       -120
'       255
```

Back to top

Explicit Conversion with the Explicit Operator

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

NOTE

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

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

TYPE	COMPARISON WITH RANGE OF INT32
Int64	Int64.MaxValue is greater than Int32.MaxValue, and Int64.MinValue is less than (has a greater negative range than) Int32.MinValue.
UInt32	UInt32.MaxValue is greater than Int32.MaxValue.
UInt64	UInt64.MaxValue is greater than Int32.MaxValue.

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

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

int intNumber;

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

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

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

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

```vb
Dim number1 As Long = Integer.MaxValue + 20L
Dim number2 As UInteger = Integer.MaxValue - 1000
Dim number3 As ULong = Integer.MaxValue

Dim intNumber As Integer

Try
   intNumber = CInt(number1)
   Console.WriteLine("After assigning a {0} value, the Integer value is {1}.",
                     number1.GetType().Name, intNumber)
Catch e As OverflowException
   If number1 > Integer.MaxValue Then
      Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                                         number1, Integer.MaxValue)
   Else
      Console.WriteLine("Conversion failed: {0} is less than {1}.\n",
                                         number1, Integer.MinValue)
   End If
End Try

Try
   intNumber = CInt(number2)
   Console.WriteLine("After assigning a {0} value, the Integer value is {1}.",
                     number2.GetType().Name, intNumber)
Catch e As OverflowException
   Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                                      number2, Integer.MaxValue)
End Try

Try
   intNumber = CInt(number3)
   Console.WriteLine("After assigning a {0} value, the Integer value is {1}.",
                     number3.GetType().Name, intNumber)
Catch e As OverflowException
   Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                                      number1, Integer.MaxValue)
End Try
' The example displays the following output:
'    Conversion failed: 2147483667 exceeds 2147483647.
'    After assigning a UInt32 value, the Integer value is 2147482647.
'    After assigning a UInt64 value, the Integer value is 2147483647.
```

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

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

> **NOTE**
>
> In C#, checked conversions can be performed by using the `checked` keyword together with a casting operator, or by specifying the `/checked+` compiler option. Conversely, unchecked conversions can be performed by using the `unchecked` keyword together with the casting operator, or by specifying the `/checked-` compiler option. By default, explicit conversions are unchecked. In Visual Basic, checked conversions can be performed by clearing the **Remove integer overflow checks** check box in the project's **Advanced Compiler Settings** dialog box, or by specifying the `/removeintchecks-` compiler option. Conversely, unchecked conversions can be performed by selecting the **Remove integer overflow checks** check box in the project's **Advanced Compiler Settings** dialog box or by specifying the `/removeintchecks+` compiler option. By default, explicit conversions are checked.

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

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

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

try {
   newValue = checked((byte) largeValue);
   Console.WriteLine("Converted the {0} value {1} to the {2} value {3}.",
                     largeValue.GetType().Name, largeValue,
                     newValue.GetType().Name, newValue);
}
catch (OverflowException) {
   Console.WriteLine("{0} is outside the range of the Byte data type.",
                     largeValue);
}
// The example displays the following output:
//    Converted the Int32 value 2147483647 to the Byte value 255.
//    2147483647 is outside the range of the Byte data type.
```

If a particular language compiler supports custom overloaded operators, you can also define explicit conversions in your own custom types. The following example provides a partial implementation of a signed byte data type named `ByteWithSign` that uses sign-and-magnitude representation. It supports explicit conversion of Int32 and UInt32 values to `ByteWithSign` values.

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

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

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

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

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

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

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

```
Public Structure ByteWithSign
   Private signValue As SByte
   Private value As Byte

   Private Const MaxValue As Byte = Byte.MaxValue
   Private Const MinValue As Integer = -1 * Byte.MaxValue

   Public Overloads Shared Narrowing Operator CType(value As Integer) As ByteWithSign
      ' Check for overflow.
      If value > ByteWithSign.MaxValue Or value < ByteWithSign.MinValue Then
         Throw New OverflowException(String.Format("'{0}' is out of range of the ByteWithSign data type.",
value))
      End If

      Dim newValue As ByteWithSign

      newValue.signValue = CSByte(Math.Sign(value))
      newValue.value = CByte(Math.Abs(value))
      Return newValue
   End Operator

   Public Overloads Shared Narrowing Operator CType(value As UInteger) As ByteWithSign
      If value > ByteWithSign.MaxValue Then
         Throw New OverflowException(String.Format("'{0}' is out of range of the ByteWithSign data type.",
value))
      End If

      Dim NewValue As ByteWithSign

      newValue.signValue = 1
      newValue.value = CByte(value)
      Return newValue
   End Operator

   Public Overrides Function ToString() As String
      Return (signValue * value).ToString()
   End Function
End Structure
```

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

```
ByteWithSign value;

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

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

```
Dim value As ByteWithSign

Try
   Dim intValue As Integer = -120
   value = CType(intValue, ByteWithSign)
   Console.WriteLine(value)
Catch e As OverflowException
   Console.WriteLine(e.Message)
End Try

Try
   Dim uintValue As UInteger = 1024
   value = CType(uintValue, ByteWithSign)
   Console.WriteLine(value)
Catch e As OverflowException
   Console.WriteLine(e.Message)
End Try
' The example displays the following output:
'       -120
'       '1024' is out of range of the ByteWithSign data type.
```

Back to top

The IConvertible Interface

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

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

Each common language runtime base type (that is, the Boolean, Byte, Char, DateTime, Decimal, Double, Int16, Int32, Int64, SByte, Single, String, UInt16, UInt32, and UInt64), as well as the DBNull and Enum types, implement the IConvertible interface. However, these are explicit interface implementations; the conversion method can be called only through an IConvertible interface variable, as the following example shows. This example converts an Int32 value to its equivalent Char value.

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

```
Dim codePoint As Integer = 1067
Dim iConv As IConvertible = codePoint
Dim ch As Char = iConv.ToChar(Nothing)
Console.WriteLine("Converted {0} to {1}.", codePoint, ch)
```

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

NOTE

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

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

Back to top

The Convert Class

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

Conversions Between Base Types

The Convert class provides a language-neutral way to perform conversions between base types and is available to all languages that target the common language runtime. It provides a complete set of methods for both widening and narrowing conversions, and throws an InvalidCastException for conversions that are not supported (such as the conversion of a DateTime value to an integer value). Narrowing conversions are performed in a checked context, and an OverflowException is thrown if the conversion fails.

IMPORTANT

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

The following example illustrates the use of the System.Convert class to perform several widening and narrowing conversions between .NET Framework base types.

```
// Convert an Int32 value to a Decimal (a widening conversion).
int integralValue = 12534;
decimal decimalValue = Convert.ToDecimal(integralValue);
Console.WriteLine("Converted the {0} value {1} to " +
                                  "the {2} value {3:N2}.",
                                  integralValue.GetType().Name,
                                  integralValue,
                                  decimalValue.GetType().Name,
                                  decimalValue);
// Convert a Byte value to an Int32 value (a widening conversion).
byte byteValue = Byte.MaxValue;
int integralValue2 = Convert.ToInt32(byteValue);
Console.WriteLine("Converted the {0} value {1} to " +
                                  "the {2} value {3:G}.",
                                  byteValue.GetType().Name,
                                  byteValue,
                                  integralValue2.GetType().Name,
                                  integralValue2);

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

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

```vb
' Convert an Int32 value to a Decimal (a widening conversion).
Dim integralValue As Integer = 12534
Dim decimalValue As Decimal = Convert.ToDecimal(integralValue)
Console.WriteLine("Converted the {0} value {1} to the {2} value {3:N2}.",
                  integralValue.GetType().Name,
                  integralValue,
                  decimalValue.GetType().Name,
                  decimalValue)

' Convert a Byte value to an Int32 value (a widening conversion).
Dim byteValue As Byte = Byte.MaxValue
Dim integralValue2 As Integer = Convert.ToInt32(byteValue)
Console.WriteLine("Converted the {0} value {1} to " +
                                  "the {2} value {3:G}.",
                                  byteValue.GetType().Name,
                                  byteValue,
                                  integralValue2.GetType().Name,
                                  integralValue2)

' Convert a Double value to an Int32 value (a narrowing conversion).
Dim doubleValue As Double = 16.32513e12
Try
   Dim longValue As Long = Convert.ToInt64(doubleValue)
   Console.WriteLine("Converted the {0} value {1:E} to " +
                                     "the {2} value {3:N0}.",
                                     doubleValue.GetType().Name,
                                     doubleValue,
                                     longValue.GetType().Name,
                                     longValue)
Catch e As OverflowException
   Console.WriteLine("Unable to convert the {0:E} value {1}.",
                                     doubleValue.GetType().Name, doubleValue)
End Try

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

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

```csharp
double doubleValue;

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

doubleValue = 42.72;
try {
   int integerValue = Convert.ToInt32(doubleValue);
   Console.WriteLine("{0} converted to {1}.",
                                     doubleValue, integerValue);
}
catch (OverflowException) {
   Console.WriteLine("Unable to convert {0} to an integer.",
                                     doubleValue);
}
// The example displays the following output:
//       13956810.96702888123451471211 converted to 13956810.9670289.
//       42.72 converted to 43.
```

```vb
Dim doubleValue As Double

' Convert a Double to a Decimal.
Dim decimalValue As Decimal = 13956810.96702888123451471211d
doubleValue = Convert.ToDouble(decimalValue)
Console.WriteLine("{0} converted to {1}.", decimalValue, doubleValue)

doubleValue = 42.72
Try
   Dim integerValue As Integer = Convert.ToInt32(doubleValue)
   Console.WriteLine("{0} converted to {1}.",
                                     doubleValue, integerValue)
Catch e As OverflowException
   Console.WriteLine("Unable to convert {0} to an integer.",
                                     doubleValue)
End Try
' The example displays the following output:
'       13956810.96702888123451471211 converted to 13956810.9670289.
'       42.72 converted to 43.
```

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

Custom Conversions with the ChangeType Method

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

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

The following example illustrates a possible implementation of the IConvertible interface that allows a

`TemperatureCelsius` object to be converted to a `TemperatureFahrenheit` object and vice versa. The example defines a base class, `Temperature`, that implements the IConvertible interface and overrides the Object.ToString method. The derived `TemperatureCelsius` and `TemperatureFahrenheit` classes each override the `ToType` and the `ToString` methods of the base class.

```
using System;

public abstract class Temperature : IConvertible
{
   protected decimal temp;

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

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

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

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

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

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

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

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

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

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

   public short ToInt16(IFormatProvider provider) {
      if (temp < Int16.MinValue || temp > Int16.MaxValue)
         throw new OverflowException(String.Format("{0} is out of range of the Int16 data type.", temp));
      else
         return (short) Math.Round(temp);
   }

   public int ToInt32(IFormatProvider provider) {
```

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

public long ToInt64(IFormatProvider provider) {
   if (temp < Int64.MinValue || temp > Int64.MaxValue)
      throw new OverflowException(String.Format("{0} is out of range of the Int64 data type.", temp));
   else
      return (long) Math.Round(temp);
}

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

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

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

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

```
            return this.ToUInt64(provider);
         default:
            throw new InvalidCastException("Conversion not supported.");
      }
   }

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

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

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

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

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

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

   // If conversionType is a implemented by another IConvertible method, call it.
   public override object ToType(Type conversionType, IFormatProvider provider) {
      // For non-objects, call base method.
      if (Type.GetTypeCode(conversionType) != TypeCode.Object) {
         return base.ToType(conversionType, provider);
      }
      else
      {
         if (conversionType.Equals(typeof(TemperatureCelsius)))
            return this;
         else if (conversionType.Equals(typeof(TemperatureFahrenheit)))
            return new TemperatureFahrenheit((decimal) this.temp * 9 / 5 + 32);
         else
            throw new InvalidCastException(String.Format("Cannot convert from Temperature to {0}.",
                                           conversionType.Name));
      }
   }
}

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

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

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

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

```
Public MustInherit Class Temperature
   Implements IConvertible

   Protected temp As Decimal

   Public Sub New(temperature As Decimal)
      Me.temp = temperature
   End Sub

   Public Property Value As Decimal
      Get
         Return Me.temp
      End Get
      Set
         Me.temp = Value
      End Set
   End Property

   Public Overrides Function ToString() As String
      Return temp.ToString() & "º"
   End Function

   ' IConvertible implementations.
   Public Function GetTypeCode() As TypeCode Implements IConvertible.GetTypeCode
      Return TypeCode.Object
   End Function

   Public Function ToBoolean(provider As IFormatProvider) As Boolean Implements IConvertible.ToBoolean
      Throw New InvalidCastException(String.Format("Temperature-to-Boolean conversion is not supported."))
   End Function

   Public Function ToByte(provider As IFormatProvider) As Byte Implements IConvertible.ToByte
      If temp < Byte.MinValue Or temp > Byte.MaxValue Then
         Throw New OverflowException(String.Format("{0} is out of range of the Byte data type.", temp))
      Else
```

```
      Else
         Return CByte(temp)
      End If
   End Function

   Public Function ToChar(provider As IFormatProvider) As Char Implements IConvertible.ToChar
      Throw New InvalidCastException("Temperature-to-Char conversion is not supported.")
   End Function

   Public Function ToDateTime(provider As IFormatProvider) As DateTime Implements IConvertible.ToDateTime
      Throw New InvalidCastException("Temperature-to-DateTime conversion is not supported.")
   End Function

   Public Function ToDecimal(provider As IFormatProvider) As Decimal Implements IConvertible.ToDecimal
      Return temp
   End Function

   Public Function ToDouble(provider As IFormatProvider) As Double Implements IConvertible.ToDouble
      Return CDbl(temp)
   End Function

   Public Function ToInt16(provider As IFormatProvider) As Int16 Implements IConvertible.ToInt16
      If temp < Int16.MinValue Or temp > Int16.MaxValue Then
         Throw New OverflowException(String.Format("{0} is out of range of the Int16 data type.", temp))
      End If
      Return CShort(Math.Round(temp))
   End Function

   Public Function ToInt32(provider As IFormatProvider) As Int32 Implements IConvertible.ToInt32
      If temp < Int32.MinValue Or temp > Int32.MaxValue Then
         Throw New OverflowException(String.Format("{0} is out of range of the Int32 data type.", temp))
      End If
      Return CInt(Math.Round(temp))
   End Function

   Public Function ToInt64(provider As IFormatProvider) As Int64 Implements IConvertible.ToInt64
      If temp < Int64.MinValue Or temp > Int64.MaxValue Then
         Throw New OverflowException(String.Format("{0} is out of range of the Int64 data type.", temp))
      End If
      Return CLng(Math.Round(temp))
   End Function

   Public Function ToSByte(provider As IFormatProvider) As SByte Implements IConvertible.ToSByte
      If temp < SByte.MinValue Or temp > SByte.MaxValue Then
         Throw New OverflowException(String.Format("{0} is out of range of the SByte data type.", temp))
      Else
         Return CSByte(temp)
      End If
   End Function

   Public Function ToSingle(provider As IFormatProvider) As Single Implements IConvertible.ToSingle
      Return CSng(temp)
   End Function

   Public Overridable Overloads Function ToString(provider As IFormatProvider) As String Implements
IConvertible.ToString
      Return temp.ToString(provider) & " °C"
   End Function

   ' If conversionType is a implemented by another IConvertible method, call it.
   Public Overridable Function ToType(conversionType As Type, provider As IFormatProvider) As Object
Implements IConvertible.ToType
      Select Case Type.GetTypeCode(conversionType)
         Case TypeCode.Boolean
            Return Me.ToBoolean(provider)
         Case TypeCode.Byte
            Return Me.ToByte(provider)
         Case TypeCode.Char
            Return Me.ToChar(provider)
```

```
         Case TypeCode.DateTime
            Return Me.ToDateTime(provider)
         Case TypeCode.Decimal
            Return Me.ToDecimal(provider)
         Case TypeCode.Double
            Return Me.ToDouble(provider)
         Case TypeCode.Empty
            Throw New NullReferenceException("The target type is null.")
         Case TypeCode.Int16
            Return Me.ToInt16(provider)
         Case TypeCode.Int32
            Return Me.ToInt32(provider)
         Case TypeCode.Int64
            Return Me.ToInt64(provider)
         Case TypeCode.Object
            ' Leave conversion of non-base types to derived classes.
            Throw New InvalidCastException(String.Format("Cannot convert from Temperature to {0}.", _
                                           conversionType.Name))
         Case TypeCode.SByte
            Return Me.ToSByte(provider)
         Case TypeCode.Single
            Return Me.ToSingle(provider)
         Case TypeCode.String
            Return Me.ToString(provider)
         Case TypeCode.UInt16
            Return Me.ToUInt16(provider)
         Case TypeCode.UInt32
            Return Me.ToUInt32(provider)
         Case TypeCode.UInt64
            Return Me.ToUInt64(provider)
         Case Else
            Throw New InvalidCastException("Conversion not supported.")
      End Select
   End Function

   Public Function ToUInt16(provider As IFormatProvider) As UInt16 Implements IConvertible.ToUInt16
      If temp < UInt16.MinValue Or temp > UInt16.MaxValue Then
         Throw New OverflowException(String.Format("{0} is out of range of the UInt16 data type.", temp))
      End If
      Return CUShort(Math.Round(temp))
   End Function

   Public Function ToUInt32(provider As IFormatProvider) As UInt32 Implements IConvertible.ToUInt32
      If temp < UInt32.MinValue Or temp > UInt32.MaxValue Then
         Throw New OverflowException(String.Format("{0} is out of range of the UInt32 data type.", temp))
      End If
      Return CUInt(Math.Round(temp))
   End Function

   Public Function ToUInt64(provider As IFormatProvider) As UInt64 Implements IConvertible.ToUInt64
      If temp < UInt64.MinValue Or temp > UInt64.MaxValue Then
         Throw New OverflowException(String.Format("{0} is out of range of the UInt64 data type.", temp))
      End If
      Return CULng(Math.Round(temp))
   End Function
End Class

Public Class TemperatureCelsius : Inherits Temperature : Implements IConvertible
   Public Sub New(value As Decimal)
      MyBase.New(value)
   End Sub

   ' Override ToString methods.
   Public Overrides Function ToString() As String
      Return Me.ToString(Nothing)
   End Function

   Public Overrides Function ToString(provider As IFormatProvider ) As String
      Return temp.ToString(provider) + "°C"
```

```vb
   End Function

   ' If conversionType is a implemented by another IConvertible method, call it.
   Public Overrides Function ToType(conversionType As Type, provider As IFormatProvider) As Object
      ' For non-objects, call base method.
      If Type.GetTypeCode(conversionType) <> TypeCode.Object Then
         Return MyBase.ToType(conversionType, provider)
      Else
         If conversionType.Equals(GetType(TemperatureCelsius)) Then
            Return Me
         ElseIf conversionType.Equals(GetType(TemperatureFahrenheit))
            Return New TemperatureFahrenheit(CDec(Me.temp * 9 / 5 + 32))
         ' Unspecified object type: throw an InvalidCastException.
         Else
            Throw New InvalidCastException(String.Format("Cannot convert from Temperature to {0}.", _
                                           conversionType.Name))
         End If
      End If
   End Function
End Class

Public Class TemperatureFahrenheit : Inherits Temperature : Implements IConvertible
   Public Sub New(value As Decimal)
      MyBase.New(value)
   End Sub

   ' Override ToString methods.
   Public Overrides Function ToString() As String
      Return Me.ToString(Nothing)
   End Function

   Public Overrides Function ToString(provider As IFormatProvider ) As String
      Return temp.ToString(provider) + "°F"
   End Function

   Public Overrides Function ToType(conversionType As Type, provider As IFormatProvider) As Object
      ' For non-objects, call base method.
      If Type.GetTypeCode(conversionType) <> TypeCode.Object Then
         Return MyBase.ToType(conversionType, provider)
      Else
         ' Handle conversion between derived classes.
         If conversionType.Equals(GetType(TemperatureFahrenheit)) Then
            Return Me
         ElseIf conversionType.Equals(GetType(TemperatureCelsius))
            Return New TemperatureCelsius(CDec((MyBase.temp - 32) * 5 / 9))
         ' Unspecified object type: throw an InvalidCastException.
         Else
            Throw New InvalidCastException(String.Format("Cannot convert from Temperature to {0}.", _
                                           conversionType.Name))
         End If
      End If
   End Function
End Class
```

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

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

```
Dim tempC1 As New TemperatureCelsius(0)
Dim tempF1 As TemperatureFahrenheit = CType(Convert.ChangeType(tempC1, GetType(TemperatureFahrenheit),
Nothing), TemperatureFahrenheit)
Console.WriteLine("{0} equals {1}.", tempC1, tempF1)
Dim tempC2 As TemperatureCelsius = CType(Convert.ChangeType(tempC1, GetType(TemperatureCelsius), Nothing),
TemperatureCelsius)
Console.WriteLine("{0} equals {1}.", tempC1, tempC2)
Dim tempF2 As New TemperatureFahrenheit(212)
Dim tempC3 As TEmperatureCelsius = CType(Convert.ChangeType(tempF2, GEtType(TemperatureCelsius), Nothing),
TemperatureCelsius)
Console.WriteLine("{0} equals {1}.", tempF2, tempC3)
Dim tempF3 As TemperatureFahrenheit = CType(Convert.ChangeType(tempF2, GetType(TemperatureFahrenheit),
Nothing), TemperatureFahrenheit)
Console.WriteLine("{0} equals {1}.", tempF2, tempF3)
' The example displays the following output:
'       0°C equals 32°F.
'       0°C equals 0°C.
'       212°F equals 100°C.
'       212°F equals 212°F.
```

Back to top

The TypeConverter Class

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

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

CONVERSION USING TYPECONVERTER	CONVERSION USING ICONVERTIBLE
Is implemented for a custom type by deriving a separate class from TypeConverter. This derived class is associated with the custom type by applying a TypeConverterAttribute attribute.	Is implemented by a custom type to perform conversion. A user of the type invokes an IConvertible conversion method on the type.
Can be used both at design time and at run time.	Can be used only at run time.

CONVERSION USING TYPECONVERTER	CONVERSION USING ICONVERTIBLE
Uses reflection; therefore, is slower than conversion enabled by IConvertible.	Does not use reflection.
Allows two-way type conversions from the custom type to other data types, and from other data types to the custom type. For example, a TypeConverter defined for `MyType` allows conversions from `MyType` to String, and from String to `MyType`.	Allows conversion from a custom type to other data types, but not from other data types to the custom type.

For more information about using type converters to perform conversions, see System.ComponentModel.TypeConverter.

See also

- System.Convert
- IConvertible
- Type Conversion Tables

Type Conversion Tables in .NET

9/6/2018 • 2 minutes to read • Edit Online

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

Widening Conversions

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

TYPE	CAN BE CONVERTED WITHOUT DATA LOSS TO
Byte	UInt16, Int16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
SByte	Int16, Int32, Int64, Single, Double, Decimal
Int16	Int32, Int64, Single, Double, Decimal
UInt16	UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Char	UInt16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Int32	Int64, Double, Decimal
UInt32	Int64, UInt64, Double, Decimal
Int64	Decimal
UInt64	Decimal
Single	Double

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

TYPE	CAN BE CONVERTED TO
Int32	Single
UInt32	Single
Int64	Single, Double
UInt64	Single, Double
Decimal	Single, Double

Narrowing Conversions

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

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

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

TYPE	CAN BE CONVERTED TO
Byte	SByte
SByte	Byte, UInt16, UInt32, UInt64
Int16	Byte, SByte, UInt16
UInt16	Byte, SByte, Int16
Int32	Byte, SByte, Int16, UInt16,UInt32
UInt32	Byte, SByte, Int16, UInt16, Int32
Int64	Byte, SByte, Int16, UInt16, Int32,UInt32,UInt64
UInt64	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64
Decimal	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64
Single	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64
Double	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64

See also

- System.Convert
- Type Conversion in .NET

Formatting Types in .NET

8/31/2018 • 38 minutes to read • Edit Online

Formatting is the process of converting an instance of a class, structure, or enumeration value to its string representation, often so that the resulting string can be displayed to users or deserialized to restore the original data type. This conversion can pose a number of challenges:

- The way that values are stored internally does not necessarily reflect the way that users want to view them. For example, a telephone number might be stored in the form 8009999999, which is not user-friendly. It should instead be displayed as 800-999-9999. See the Custom Format Strings section for an example that formats a number in this way.
- Sometimes the conversion of an object to its string representation is not intuitive. For example, it is not clear how the string representation of a Temperature object or a Person object should appear. For an example that formats a Temperature object in a variety of ways, see the Standard Format Strings section.
- Values often require culture-sensitive formatting. For example, in an application that uses numbers to reflect monetary values, numeric strings should include the current culture's currency symbol, group separator (which, in most cultures, is the thousands separator), and decimal symbol. For an example, see the Culture-Sensitive Formatting with Format Providers and the IFormatProvider Interface section.
- An application may have to display the same value in different ways. For example, an application may represent an enumeration member by displaying a string representation of its name or by displaying its underlying value. For an example that formats a member of the DayOfWeek enumeration in different ways, see the Standard Format Strings section.

> **NOTE**
> Formatting converts the value of a type into a string representation. Parsing is the inverse of formatting. A parsing operation creates an instance of a data type from its string representation. For information about converting strings to other data types, see Parsing Strings.

.NET provides rich formatting support that enables developers to address these requirements.

This overview contains the following sections:

- The IFormattable Interface
- Composite Formatting
- Custom Formatting with ICustomFormatter
- Related Topics
- Reference

Formatting in .NET

The basic mechanism for formatting is the default implementation of the Object.ToString method, which is discussed in the Default Formatting Using the ToString Method section later in this topic. However, .NET provides several ways to modify and extend its default formatting support. These include the following:

- Overriding the Object.ToString method to define a custom string representation of an object's value. For more information, see the Overriding the ToString Method section later in this topic.
- Defining format specifiers that enable the string representation of an object's value to take multiple forms. For example, the "X" format specifier in the following statement converts an integer to the string representation of a hexadecimal value.

  ```
  int integerValue = 60312;
  Console.WriteLine(integerValue.ToString("X"));   // Displays EB98.
  ```

  ```
  Dim integerValue As Integer = 60312
  Console.WriteLine(integerValue.ToString("X"))   ' Displays EB98.
  ```

  For more information about format specifiers, see the ToString Method and Format Strings section.

- Using format providers to take advantage of the formatting conventions of a specific culture. For example, the following statement displays a currency value by using the formatting conventions of the en-US culture.

  ```
  double cost = 1632.54;
  Console.WriteLine(cost.ToString("C",
                    new System.Globalization.CultureInfo("en-US")));
  // The example displays the following output:
  //       $1,632.54
  ```

  ```
  Dim cost As Double = 1632.54
  Console.WriteLine(cost.ToString("C", New System.Globalization.CultureInfo("en-US")))
  ' The example displays the following output:
  '       $1,632.54
  ```

  For more information about formatting with format providers, see the Format Providers and the IFormatProvider Interface section.

- Implementing the IFormattable interface to support both string conversion with the Convert class and composite formatting. For more information, see the IFormattable Interface section.
- Using composite formatting to embed the string representation of a value in a larger string. For more information, see the Composite Formatting section.
- Implementing ICustomFormatter and IFormatProvider to provide a complete custom formatting solution.

For more information, see the Custom Formatting with ICustomFormatter section.

The following sections examine these methods for converting an object to its string representation.

Back to top

Default Formatting Using the ToString Method

Every type that is derived from System.Object automatically inherits a parameterless `ToString` method, which returns the name of the type by default. The following example illustrates the default `ToString` method. It defines a class named `Automobile` that has no implementation. When the class is instantiated and its `ToString` method is called, it displays its type name. Note that the `ToString` method is not explicitly called in the example. The Console.WriteLine(Object) method implicitly calls the `ToString` method of the object passed to it as an argument.

```
using System;

public class Automobile
{
   // No implementation. All members are inherited from Object.
}

public class Example
{
   public static void Main()
   {
      Automobile firstAuto = new Automobile();
      Console.WriteLine(firstAuto);
   }
}
// The example displays the following output:
//       Automobile
```

```
Public Class Automobile
   ' No implementation. All members are inherited from Object.
End Class

Module Example
   Public Sub Main()
      Dim firstAuto As New Automobile()
      Console.WriteLine(firstAuto)
   End Sub
End Module
' The example displays the following output:
'       Automobile
```

WARNING

Starting with Windows 8.1, the Windows Runtime includes an IStringable interface with a single method, IStringable.ToString, which provides default formatting support. However, we recommend that managed types do not implement the `IStringable` interface. For more information, see "The Windows Runtime and the `IStringable` Interface" section on the Object.ToString reference page.

Because all types other than interfaces are derived from Object, this functionality is automatically provided to your custom classes or structures. However, the functionality offered by the default `ToString` method, is limited: Although it identifies the type, it fails to provide any information about an instance of the type. To provide a string representation of an object that provides information about that object, you must override the `ToString` method.

NOTE

Structures inherit from ValueType, which in turn is derived from Object. Although ValueType overrides Object.ToString, its implementation is identical.

Back to top

Overriding the ToString Method

Displaying the name of a type is often of limited use and does not allow consumers of your types to differentiate one instance from another. However, you can override the `ToString` method to provide a more useful representation of an object's value. The following example defines a `Temperature` object and overrides its `ToString` method to display the temperature in degrees Celsius.

```
using System;

public class Temperature
{
   private decimal temp;

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

   public override string ToString()
   {
      return this.temp.ToString("N1") + "°C";
   }
}

public class Example
{
   public static void Main()
   {
      Temperature currentTemperature = new Temperature(23.6m);
      Console.WriteLine("The current temperature is " +
                        currentTemperature.ToString());
   }
}
// The example displays the following output:
//       The current temperature is 23.6°C.
```

```vb
Public Class Temperature
   Private temp As Decimal

   Public Sub New(temperature As Decimal)
      Me.temp = temperature
   End Sub

   Public Overrides Function ToString() As String
      Return Me.temp.ToString("N1") + "°C"
   End Function
End Class

Module Example
   Public Sub Main()
      Dim currentTemperature As New Temperature(23.6d)
      Console.WriteLine("The current temperature is " +
                        currentTemperature.ToString())
   End Sub
End Module
' The example displays the following output:
'       The current temperature is 23.6°C.
```

In .NET, the `ToString` method of each primitive value type has been overridden to display the object's value instead of its name. The following table shows the override for each primitive type. Note that most of the overridden methods call another overload of the `ToString` method and pass it the "G" format specifier, which defines the general format for its type, and an IFormatProvider object that represents the current culture.

TYPE	TOSTRING OVERRIDE
Boolean	Returns either Boolean.TrueString or Boolean.FalseString.
Byte	Calls `Byte.ToString("G", NumberFormatInfo.CurrentInfo)` to format the Byte value for the current culture.
Char	Returns the character as a string.
DateTime	Calls `DateTime.ToString("G", DatetimeFormatInfo.CurrentInfo)` to format the date and time value for the current culture.
Decimal	Calls `Decimal.ToString("G", NumberFormatInfo.CurrentInfo)` to format the Decimal value for the current culture.
Double	Calls `Double.ToString("G", NumberFormatInfo.CurrentInfo)` to format the Double value for the current culture.
Int16	Calls `Int16.ToString("G", NumberFormatInfo.CurrentInfo)` to format the Int16 value for the current culture.
Int32	Calls `Int32.ToString("G", NumberFormatInfo.CurrentInfo)` to format the Int32 value for the current culture.

TYPE	TOSTRING OVERRIDE
Int64	Calls `Int64.ToString("G", NumberFormatInfo.CurrentInfo)` to format the Int64 value for the current culture.
SByte	Calls `SByte.ToString("G", NumberFormatInfo.CurrentInfo)` to format the SByte value for the current culture.
Single	Calls `Single.ToString("G", NumberFormatInfo.CurrentInfo)` to format the Single value for the current culture.
UInt16	Calls `UInt16.ToString("G", NumberFormatInfo.CurrentInfo)` to format the UInt16 value for the current culture.
UInt32	Calls `UInt32.ToString("G", NumberFormatInfo.CurrentInfo)` to format the UInt32 value for the current culture.
UInt64	Calls `UInt64.ToString("G", NumberFormatInfo.CurrentInfo)` to format the UInt64 value for the current culture.

Back to top

The ToString Method and Format Strings

Relying on the default `ToString` method or overriding `ToString` is appropriate when an object has a single string representation. However, the value of an object often has multiple representations. For example, a temperature can be expressed in degrees Fahrenheit, degrees Celsius, or kelvins. Similarly, the integer value 10 can be represented in numerous ways, including 10, 10.0, 1.0e01, or $10.00.

To enable a single value to have multiple string representations, .NET uses format strings. A format string is a string that contains one or more predefined format specifiers, which are single characters or groups of characters that define how the `ToString` method should format its output. The format string is then passed as a parameter to the object's `ToString` method and determines how the string representation of that object's value should appear.

All numeric types, date and time types, and enumeration types in .NET support a predefined set of format specifiers. You can also use format strings to define multiple string representations of your application-defined data types.

Standard Format Strings

A standard format string contains a single format specifier, which is an alphabetic character that defines the string representation of the object to which it is applied, along with an optional precision specifier that affects how many digits are displayed in the result string. If the precision specifier is omitted or is not supported, a standard format specifier is equivalent to a standard format string.

.NET defines a set of standard format specifiers for all numeric types, all date and time types, and all enumeration types. For example, each of these categories supports a "G" standard format specifier, which defines a general string representation of a value of that type.

Standard format strings for enumeration types directly control the string representation of a value. The format strings passed to an enumeration value's `ToString` method determine whether the value is displayed using its string name (the "G" and "F" format specifiers), its underlying integral value (the "D" format specifier), or its hexadecimal value (the "X" format specifier). The following example illustrates the use of standard format strings to format a DayOfWeek enumeration value.

```
DayOfWeek thisDay = DayOfWeek.Monday;
string[] formatStrings = {"G", "F", "D", "X"};

foreach (string formatString in formatStrings)
   Console.WriteLine(thisDay.ToString(formatString));
// The example displays the following output:
//       Monday
//       Monday
//       1
//       00000001
```

```
Dim thisDay As DayOfWeek = DayOfWeek.Monday
Dim formatStrings() As String = {"G", "F", "D", "X"}

For Each formatString As String In formatStrings
   Console.WriteLine(thisDay.ToString(formatString))
Next
' The example displays the following output:
'       Monday
'       Monday
'       1
'       00000001
```

For information about enumeration format strings, see Enumeration Format Strings.

Standard format strings for numeric types usually define a result string whose precise appearance is controlled by one or more property values. For example, the "C" format specifier formats a number as a currency value. When you call the `ToString` method with the "C" format specifier as the only parameter, the following property values from the current culture's NumberFormatInfo object are used to define the string representation of the numeric value:

- The CurrencySymbol property, which specifies the current culture's currency symbol.
- The CurrencyNegativePattern or CurrencyPositivePattern property, which returns an integer that determines the following:
  - The placement of the currency symbol.
  - Whether negative values are indicated by a leading negative sign, a trailing negative sign, or parentheses.
  - Whether a space appears between the numeric value and the currency symbol.
- The CurrencyDecimalDigits property, which defines the number of fractional digits in the result string.
- The CurrencyDecimalSeparator property, which defines the decimal separator symbol in the result string.
- The CurrencyGroupSeparator property, which defines the group separator symbol.
- The CurrencyGroupSizes property, which defines the number of digits in each group to the left of the decimal.
- The NegativeSign property, which determines the negative sign used in the result string if parentheses are not used to indicate negative values.

In addition, numeric format strings may include a precision specifier. The meaning of this specifier depends on the format string with which it is used, but it typically indicates either the total number of digits or the number of fractional digits that should appear in the result string. For example, the following example uses the "X4" standard numeric string and a precision specifier to create a string value that has four hexadecimal digits.

```
byte[] byteValues = { 12, 163, 255 };
foreach (byte byteValue in byteValues)
   Console.WriteLine(byteValue.ToString("X4"));
// The example displays the following output:
//       000C
//       00A3
//       00FF
```

```
Dim byteValues() As Byte = { 12, 163, 255 }
For Each byteValue As Byte In byteValues
   Console.WriteLine(byteValue.ToString("X4"))
Next
' The example displays the following output:
'       000C
'       00A3
'       00FF
```

For more information about standard numeric formatting strings, see Standard Numeric Format Strings.

Standard format strings for date and time values are aliases for custom format strings stored by a particular DateTimeFormatInfo property. For example, calling the `ToString` method of a date and time value with the "D" format specifier displays the date and time by using the custom format string stored in the current culture's DateTimeFormatInfo.LongDatePattern property. (For more information about custom format strings, see the next section.) The following example illustrates this relationship.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      DateTime date1 = new DateTime(2009, 6, 30);
      Console.WriteLine("D Format Specifier:     {0:D}", date1);
      string longPattern = CultureInfo.CurrentCulture.DateTimeFormat.LongDatePattern;
      Console.WriteLine("'{0}' custom format string:     {1}",
                        longPattern, date1.ToString(longPattern));
   }
}
// The example displays the following output when run on a system whose
// current culture is en-US:
//    D Format Specifier:     Tuesday, June 30, 2009
//    'dddd, MMMM dd, yyyy' custom format string:     Tuesday, June 30, 2009
```

```
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim date1 As Date = #6/30/2009#
      Console.WriteLine("D Format Specifier:     {0:D}", date1)
      Dim longPattern As String = CultureInfo.CurrentCulture.DateTimeFormat.LongDatePattern
      Console.WriteLine("'{0}' custom format string:     {1}", _
                        longPattern, date1.ToString(longPattern))
   End Sub
End Module
' The example displays the following output when run on a system whose
' current culture is en-US:
'    D Format Specifier:     Tuesday, June 30, 2009
'    'dddd, MMMM dd, yyyy' custom format string:     Tuesday, June 30, 2009
```

For more information about standard date and time format strings, see Standard Date and Time Format Strings.

You can also use standard format strings to define the string representation of an application-defined object that is produced by the object's `ToString(String)` method. You can define the specific standard format specifiers that your object supports, and you can determine whether they are case-sensitive or case-insensitive. Your implementation of the `ToString(String)` method should support the following:

- A "G" format specifier that represents a customary or common format of the object. The parameterless overload of your object's `ToString` method should call its `ToString(String)` overload and pass it the "G" standard format string.

- Support for a format specifier that is equal to a null reference (`Nothing` in Visual Basic). A format specifier that is equal to a null reference should be considered equivalent to the "G" format specifier.

For example, a `Temperature` class can internally store the temperature in degrees Celsius and use format specifiers to represent the value of the `Temperature` object in degrees Celsius, degrees Fahrenheit, and kelvins. The following example provides an illustration.

```
using System;

public class Temperature
{
   private decimal m_Temp;

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

   public decimal Celsius
   {
      get { return this.m_Temp; }
   }

   public decimal Kelvin
   {
      get { return this.m_Temp + 273.15m; }
   }

   public decimal Fahrenheit
   {
      get { return Math.Round(((decimal) (this.m_Temp * 9 / 5 + 32)), 2); }
   }

   public override string ToString()
   {
      return this.ToString("C");
```

```
   }

   public string ToString(string format)
   {
      // Handle null or empty string.
      if (String.IsNullOrEmpty(format)) format = "C";
      // Remove spaces and convert to uppercase.
      format = format.Trim().ToUpperInvariant();

      // Convert temperature to Fahrenheit and return string.
      switch (format)
      {
         // Convert temperature to Fahrenheit and return string.
         case "F":
            return this.Fahrenheit.ToString("N2") + " °F";
         // Convert temperature to Kelvin and return string.
         case "K":
            return this.Kelvin.ToString("N2") + " K";
         // return temperature in Celsius.
         case "G":
         case "C":
            return this.Celsius.ToString("N2") + " °C";
         default:
            throw new FormatException(String.Format("The '{0}' format string is not supported.", format));
      }
   }
}

public class Example
{
   public static void Main()
   {
      Temperature temp1 = new Temperature(0m);
      Console.WriteLine(temp1.ToString());
      Console.WriteLine(temp1.ToString("G"));
      Console.WriteLine(temp1.ToString("C"));
      Console.WriteLine(temp1.ToString("F"));
      Console.WriteLine(temp1.ToString("K"));

      Temperature temp2 = new Temperature(-40m);
      Console.WriteLine(temp2.ToString());
      Console.WriteLine(temp2.ToString("G"));
      Console.WriteLine(temp2.ToString("C"));
      Console.WriteLine(temp2.ToString("F"));
      Console.WriteLine(temp2.ToString("K"));

      Temperature temp3 = new Temperature(16m);
      Console.WriteLine(temp3.ToString());
      Console.WriteLine(temp3.ToString("G"));
      Console.WriteLine(temp3.ToString("C"));
      Console.WriteLine(temp3.ToString("F"));
      Console.WriteLine(temp3.ToString("K"));

      Console.WriteLine(String.Format("The temperature is now {0:F}.", temp3));
   }
}
// The example displays the following output:
//       0.00 °C
//       0.00 °C
//       0.00 °C
//       32.00 °F
//       273.15 K
//       -40.00 °C
//       -40.00 °C
//       -40.00 °C
//       -40.00 °F
//       233.15 K
//       16.00 °C
//       16.00 °C
```

```
//       16.00 °C
//       60.80 °F
//       289.15 K
//       The temperature is now 16.00 °C.
```

```
Public Class Temperature
   Private m_Temp As Decimal

   Public Sub New(temperature As Decimal)
      Me.m_Temp = temperature
   End Sub

   Public ReadOnly Property Celsius() As Decimal
      Get
         Return Me.m_Temp
      End Get
   End Property

   Public ReadOnly Property Kelvin() As Decimal
      Get
         Return Me.m_Temp + 273.15d
      End Get
   End Property

   Public ReadOnly Property Fahrenheit() As Decimal
      Get
         Return Math.Round(CDec(Me.m_Temp * 9 / 5 + 32), 2)
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return Me.ToString("C")
   End Function

   Public Overloads Function ToString(format As String) As String
      ' Handle null or empty string.
      If String.IsNullOrEmpty(format) Then format = "C"
      ' Remove spaces and convert to uppercase.
      format = format.Trim().ToUpperInvariant()

      Select Case format
         Case "F"
           ' Convert temperature to Fahrenheit and return string.
            Return Me.Fahrenheit.ToString("N2") & " °F"
         Case "K"
            ' Convert temperature to Kelvin and return string.
            Return Me.Kelvin.ToString("N2") & " K"
         Case "C", "G"
            ' Return temperature in Celsius.
            Return Me.Celsius.ToString("N2") & " °C"
         Case Else
            Throw New FormatException(String.Format("The '{0}' format string is not supported.", format))
      End Select
   End Function
End Class

Public Module Example
   Public Sub Main()
      Dim temp1 As New Temperature(0d)
      Console.WriteLine(temp1.ToString())
      Console.WriteLine(temp1.ToString("G"))
      Console.WriteLine(temp1.ToString("C"))
      Console.WriteLine(temp1.ToString("F"))
      Console.WriteLine(temp1.ToString("K"))

      Dim temp2 As New Temperature(-40d)
      Console.WriteLine(temp2.ToString())
```

```
        Console.WriteLine(temp2.ToString("G"))
        Console.WriteLine(temp2.ToString("C"))
        Console.WriteLine(temp2.ToString("F"))
        Console.WriteLine(temp2.ToString("K"))

        Dim temp3 As New Temperature(16d)
        Console.WriteLine(temp3.ToString())
        Console.WriteLine(temp3.ToString("G"))
        Console.WriteLine(temp3.ToString("C"))
        Console.WriteLine(temp3.ToString("F"))
        Console.WriteLine(temp3.ToString("K"))

        Console.WriteLine(String.Format("The temperature is now {0:F}.", temp3))
    End Sub
End Module
' The example displays the following output:
'       0.00 °C
'       0.00 °C
'       0.00 °C
'       32.00 °F
'       273.15 K
'       -40.00 °C
'       -40.00 °C
'       -40.00 °C
'       -40.00 °F
'       233.15 K
'       16.00 °C
'       16.00 °C
'       16.00 °C
'       60.80 °F
'       289.15 K
'       The temperature is now 16.00 °C.
```

Back to top

Custom Format Strings

In addition to the standard format strings, .NET defines custom format strings for both numeric values and date and time values. A custom format string consists of one or more custom format specifiers that define the string representation of a value. For example, the custom date and time format string "yyyy/mm/dd hh:mm:ss.ffff t zzz" converts a date to its string representation in the form "2008/11/15 07:45:00.0000 P -08:00" for the en-US culture. Similarly, the custom format string "0000" converts the integer value 12 to "0012". For a complete list of custom format strings, see Custom Date and Time Format Strings and Custom Numeric Format Strings.

If a format string consists of a single custom format specifier, the format specifier should be preceded by the percent (%) symbol to avoid confusion with a standard format specifier. The following example uses the "M" custom format specifier to display a one-digit or two-digit number of the month of a particular date.

```
DateTime date1 = new DateTime(2009, 9, 8);
Console.WriteLine(date1.ToString("%M"));       // Displays 9
```

```
Dim date1 As Date = #09/08/2009#
Console.WriteLine(date1.ToString("%M"))      ' Displays 9
```

Many standard format strings for date and time values are aliases for custom format strings that are defined by properties of the DateTimeFormatInfo object. Custom format strings also offer considerable flexibility in providing application-defined formatting for numeric values or date and time values. You can define your own custom result strings for both numeric values and date and time values by combining multiple custom format specifiers into a single custom format string. The following example defines a custom format string that displays the day of the week in parentheses after the month name, day, and year.

```
string customFormat = "MMMM dd, yyyy (dddd)";
DateTime date1 = new DateTime(2009, 8, 28);
Console.WriteLine(date1.ToString(customFormat));
// The example displays the following output if run on a system
// whose language is English:
//       August 28, 2009 (Friday)
```

```
Dim customFormat As String = "MMMM dd, yyyy (dddd)"
Dim date1 As Date = #8/28/2009#
Console.WriteLine(date1.ToString(customFormat))
' The example displays the following output if run on a system
' whose language is English:
'       August 28, 2009 (Friday)
```

The following example defines a custom format string that displays an Int64 value as a standard, seven-digit U.S. telephone number along with its area code.

```
using System;

public class Example
{
   public static void Main()
   {
      long number = 8009999999;
      string fmt = "000-000-0000";
      Console.WriteLine(number.ToString(fmt));
   }
}
// The example displays the following output:
//        800-999-9999
```

```
Module Example
   Public Sub Main()
      Dim number As Long = 8009999999
      Dim fmt As String = "000-000-0000"
      Console.WriteLine(number.ToString(fmt))
   End Sub
End Module
' The example displays the following output:

' The example displays the following output:
'       800-999-9999
```

Although standard format strings can generally handle most of the formatting needs for your application-defined types, you may also define custom format specifiers to format your types.

Back to top

Format Strings and .NET Types

All numeric types (that is, the Byte, Decimal, Double, Int16, Int32, Int64, SByte, Single, UInt16, UInt32, UInt64, and BigInteger types), as well as the DateTime, DateTimeOffset, TimeSpan, Guid, and all enumeration types, support formatting with format strings. For information on the specific format strings supported by each type, see the following topics:

TITLE	DEFINITION

TITLE	DEFINITION
Standard Numeric Format Strings	Describes standard format strings that create commonly used string representations of numeric values.
Custom Numeric Format Strings	Describes custom format strings that create application-specific formats for numeric values.
Standard Date and Time Format Strings	Describes standard format strings that create commonly used string representations of DateTime and DateTimeOffset values.
Custom Date and Time Format Strings	Describes custom format strings that create application-specific formats for DateTime and DateTimeOffset values.
Standard TimeSpan Format Strings	Describes standard format strings that create commonly used string representations of time intervals.
Custom TimeSpan Format Strings	Describes custom format strings that create application-specific formats for time intervals.
Enumeration Format Strings	Describes standard format strings that are used to create string representations of enumeration values.
Guid.ToString(String)	Describes standard format strings for Guid values.

Culture-Sensitive Formatting with Format Providers and the IFormatProvider Interface

Although format specifiers let you customize the formatting of objects, producing a meaningful string representation of objects often requires additional formatting information. For example, formatting a number as a currency value by using either the "C" standard format string or a custom format string such as "$ #,#.00" requires, at a minimum, information about the correct currency symbol, group separator, and decimal separator to be available to include in the formatted string. In .NET, this additional formatting information is made available through the IFormatProvider interface, which is provided as a parameter to one or more overloads of the `ToString` method of numeric types and date and time types. IFormatProvider implementations are used in .NET to support culture-specific formatting. The following example illustrates how the string representation of an object changes when it is formatted with three IFormatProvider objects that represent different cultures.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      decimal value = 1603.42m;
      Console.WriteLine(value.ToString("C3", new CultureInfo("en-US")));
      Console.WriteLine(value.ToString("C3", new CultureInfo("fr-FR")));
      Console.WriteLine(value.ToString("C3", new CultureInfo("de-DE")));
   }
}
// The example displays the following output:
//       $1,603.420
//       1 603,420 €
//       1.603,420 €
```

```
Imports System.Globalization

Public Module Example
   Public Sub Main()
      Dim value As Decimal = 1603.42d
      Console.WriteLine(value.ToString("C3", New CultureInfo("en-US")))
      Console.WriteLine(value.ToString("C3", New CultureInfo("fr-FR")))
      Console.WriteLine(value.ToString("C3", New CultureInfo("de-DE")))
   End Sub
End Module
' The example displays the following output:
'       $1,603.420
'       1 603,420 €
'       1.603,420 €
```

The IFormatProvider interface includes one method, GetFormat(Type), which has a single parameter that specifies the type of object that provides formatting information. If the method can provide an object of that type, it returns it. Otherwise, it returns a null reference (`Nothing` in Visual Basic).

IFormatProvider.GetFormat is a callback method. When you call a `ToString` method overload that includes an IFormatProvider parameter, it calls the GetFormat method of that IFormatProvider object. The GetFormat method is responsible for returning an object that provides the necessary formatting information, as specified by its `formatType` parameter, to the `ToString` method.

A number of formatting or string conversion methods include a parameter of type IFormatProvider, but in many cases the value of the parameter is ignored when the method is called. The following table lists some of the formatting methods that use the parameter and the type of the Type object that they pass to the IFormatProvider.GetFormat method.

METHOD	TYPE OF `FORMATTYPE` PARAMETER
`ToString` method of numeric types	System.Globalization.NumberFormatInfo
`ToString` method of date and time types	System.Globalization.DateTimeFormatInfo
String.Format	System.ICustomFormatter
StringBuilder.AppendFormat	System.ICustomFormatter

NOTE

The `ToString` methods of the numeric types and date and time types are overloaded, and only some of the overloads include an IFormatProvider parameter. If a method does not have a parameter of type IFormatProvider, the object that is returned by the CultureInfo.CurrentCulture property is passed instead. For example, a call to the default Int32.ToString() method ultimately results in a method call such as the following: `Int32.ToString("G", System.Globalization.CultureInfo.CurrentCulture)`.

.NET provides three classes that implement IFormatProvider:

- DateTimeFormatInfo, a class that provides formatting information for date and time values for a specific culture. Its IFormatProvider.GetFormat implementation returns an instance of itself.
- NumberFormatInfo, a class that provides numeric formatting information for a specific culture. Its IFormatProvider.GetFormat implementation returns an instance of itself.
- CultureInfo. Its IFormatProvider.GetFormat implementation can return either a NumberFormatInfo

object to provide numeric formatting information or a DateTimeFormatInfo object to provide formatting information for date and time values.

You can also implement your own format provider to replace any one of these classes. However, your implementation's GetFormat method must return an object of the type listed in the previous table if it has to provide formatting information to the `ToString` method.

Back to top

Culture-Sensitive Formatting of Numeric Values

By default, the formatting of numeric values is culture-sensitive. If you do not specify a culture when you call a formatting method, the formatting conventions of the current thread culture are used. This is illustrated in the following example, which changes the current thread culture four times and then calls the Decimal.ToString(String) method. In each case, the result string reflects the formatting conventions of the current culture. This is because the `ToString` and `ToString(String)` methods wrap calls to each numeric type's `ToString(String, IFormatProvider)` method.

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string[] cultureNames = { "en-US", "fr-FR", "es-MX", "de-DE" };
      Decimal value = 1043.17m;

      foreach (var cultureName in cultureNames) {
         // Change the current thread culture.
         Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName);
         Console.WriteLine("The current culture is {0}",
                           Thread.CurrentThread.CurrentCulture.Name);
         Console.WriteLine(value.ToString("C2"));
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       The current culture is en-US
//       $1,043.17
//
//       The current culture is fr-FR
//       1 043,17 €
//
//       The current culture is es-MX
//       $1,043.17
//
//       The current culture is de-DE
//       1.043,17 €
```

```vb
Imports System.Globalization
Imports System.Threading

Module Example
   Public Sub Main()
      Dim cultureNames() As String = { "en-US", "fr-FR", "es-MX", "de-DE" }
      Dim value As Decimal = 1043.17d

      For Each cultureName In cultureNames
         ' Change the current thread culture.
         Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName)
         Console.WriteLine("The current culture is {0}",
                           Thread.CurrentThread.CurrentCulture.Name)
         Console.WriteLine(value.ToString("C2"))
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays the following output:
'       The current culture is en-US
'       $1,043.17
'
'       The current culture is fr-FR
'       1 043,17 €
'
'       The current culture is es-MX
'       $1,043.17
'
'       The current culture is de-DE
'       1.043,17 €
```

You can also format a numeric value for a specific culture by calling a `ToString` overload that has a `provider` parameter and passing it either of the following:

- A CultureInfo object that represents the culture whose formatting conventions are to be used. Its CultureInfo.GetFormat method returns the value of the CultureInfo.NumberFormat property, which is the NumberFormatInfo object that provides culture-specific formatting information for numeric values.

- A NumberFormatInfo object that defines the culture-specific formatting conventions to be used. Its GetFormat method returns an instance of itself.

The following example uses NumberFormatInfo objects that represent the English (United States) and English (Great Britain) cultures and the French and Russian neutral cultures to format a floating-point number.

```csharp
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      Double value = 1043.62957;
      string[] cultureNames = { "en-US", "en-GB", "ru", "fr" };

      foreach (var name in cultureNames) {
         NumberFormatInfo nfi = CultureInfo.CreateSpecificCulture(name).NumberFormat;
         Console.WriteLine("{0,-6} {1}", name + ":", value.ToString("N3", nfi));
      }
   }
}
// The example displays the following output:
//       en-US: 1,043.630
//       en-GB: 1,043.630
//       ru:    1 043,630
//       fr:    1 043,630
```

```vb
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim value As Double = 1043.62957
      Dim cultureNames() As String = { "en-US", "en-GB", "ru", "fr" }

      For Each name In cultureNames
         Dim nfi As NumberFormatInfo = CultureInfo.CreateSpecificCulture(name).NumberFormat
         Console.WriteLine("{0,-6} {1}", name + ":", value.ToString("N3", nfi))
      Next
   End Sub
End Module
' The example displays the following output:
'       en-US: 1,043.630
'       en-GB: 1,043.630
'       ru:    1 043,630
'       fr:    1 043,630
```

Culture-Sensitive Formatting of Date and Time Values

By default, the formatting of date and time values is culture-sensitive. If you do not specify a culture when you call a formatting method, the formatting conventions of the current thread culture are used. This is illustrated in the following example, which changes the current thread culture four times and then calls the DateTime.ToString(String) method. In each case, the result string reflects the formatting conventions of the current culture. This is because the DateTime.ToString(), DateTime.ToString(String), DateTimeOffset.ToString(), and DateTimeOffset.ToString(String) methods wrap calls to the DateTime.ToString(String, IFormatProvider) and DateTimeOffset.ToString(String, IFormatProvider) methods.

```csharp
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string[] cultureNames = { "en-US", "fr-FR", "es-MX", "de-DE" };
      DateTime dateToFormat = new DateTime(2012, 5, 28, 11, 30, 0);

      foreach (var cultureName in cultureNames) {
         // Change the current thread culture.
         Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName);
         Console.WriteLine("The current culture is {0}",
                           Thread.CurrentThread.CurrentCulture.Name);
         Console.WriteLine(dateToFormat.ToString("F"));
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       The current culture is en-US
//       Monday, May 28, 2012 11:30:00 AM
//
//       The current culture is fr-FR
//       lundi 28 mai 2012 11:30:00
//
//       The current culture is es-MX
//       lunes, 28 de mayo de 2012 11:30:00 a.m.
//
//       The current culture is de-DE
//       Montag, 28. Mai 2012 11:30:00
```

```vb
Imports System.Globalization
Imports System.Threading

Module Example
   Public Sub Main()
      Dim cultureNames() As String = { "en-US", "fr-FR", "es-MX", "de-DE" }
      Dim dateToFormat As Date = #5/28/2012 11:30AM#

      For Each cultureName In cultureNames
         ' Change the current thread culture.
         Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName)
         Console.WriteLine("The current culture is {0}",
                           Thread.CurrentThread.CurrentCulture.Name)
         Console.WriteLine(dateToFormat.ToString("F"))
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays the following output:
'       The current culture is en-US
'       Monday, May 28, 2012 11:30:00 AM
'
'       The current culture is fr-FR
'       lundi 28 mai 2012 11:30:00
'
'       The current culture is es-MX
'       lunes, 28 de mayo de 2012 11:30:00 a.m.
'
'       The current culture is de-DE
'       Montag, 28. Mai 2012 11:30:00
```

You can also format a date and time value for a specific culture by calling a DateTime.ToString or DateTimeOffset.ToString overload that has a `provider` parameter and passing it either of the following:

- A CultureInfo object that represents the culture whose formatting conventions are to be used. Its CultureInfo.GetFormat method returns the value of the CultureInfo.DateTimeFormat property, which is the DateTimeFormatInfo object that provides culture-specific formatting information for date and time values.

- A DateTimeFormatInfo object that defines the culture-specific formatting conventions to be used. Its GetFormat method returns an instance of itself.

The following example uses DateTimeFormatInfo objects that represent the English (United States) and English (Great Britain) cultures and the French and Russian neutral cultures to format a date.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      DateTime dat1 = new DateTime(2012, 5, 28, 11, 30, 0);
      string[] cultureNames = { "en-US", "en-GB", "ru", "fr" };

      foreach (var name in cultureNames) {
         DateTimeFormatInfo dtfi = CultureInfo.CreateSpecificCulture(name).DateTimeFormat;
         Console.WriteLine("{0}: {1}", name, dat1.ToString(dtfi));
      }
   }
}
// The example displays the following output:
//       en-US: 5/28/2012 11:30:00 AM
//       en-GB: 28/05/2012 11:30:00
//       ru: 28.05.2012 11:30:00
//       fr: 28/05/2012 11:30:00
```

```
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim dat1 As Date = #5/28/2012 11:30AM#
      Dim cultureNames() As String = { "en-US", "en-GB", "ru", "fr" }

      For Each name In cultureNames
         Dim dtfi As DateTimeFormatInfo = CultureInfo.CreateSpecificCulture(name).DateTimeFormat
         Console.WriteLine("{0}: {1}", name, dat1.ToString(dtfi))
      Next
   End Sub
End Module
' The example displays the following output:
'       en-US: 5/28/2012 11:30:00 AM
'       en-GB: 28/05/2012 11:30:00
'       ru: 28.05.2012 11:30:00
'       fr: 28/05/2012 11:30:00
```

The IFormattable Interface

Typically, types that overload the `ToString` method with a format string and an IFormatProvider parameter also implement the IFormattable interface. This interface has a single member, IFormattable.ToString(String, IFormatProvider), that includes both a format string and a format provider as parameters.

Implementing the IFormattable interface for your application-defined class offers two advantages:

- Support for string conversion by the Convert class. Calls to the Convert.ToString(Object) and Convert.ToString(Object, IFormatProvider) methods call your IFormattable implementation automatically.

- Support for composite formatting. If a format item that includes a format string is used to format your custom type, the common language runtime automatically calls your IFormattable implementation and passes it the format string. For more information about composite formatting with methods such as String.Format or Console.WriteLine, see the Composite Formatting section.

The following example defines a `Temperature` class that implements the IFormattable interface. It supports the "C" or "G" format specifiers to display the temperature in Celsius, the "F" format specifier to display the temperature in Fahrenheit, and the "K" format specifier to display the temperature in Kelvin.

```
using System;
using System.Globalization;

public class Temperature : IFormattable
{
   private decimal m_Temp;

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

   public decimal Celsius
   {
      get { return this.m_Temp; }
   }

   public decimal Kelvin
   {
      get { return this.m_Temp + 273.15m; }
   }

   public decimal Fahrenheit
   {
      get { return Math.Round((decimal) this.m_Temp * 9 / 5 + 32, 2); }
   }

   public override string ToString()
   {
      return this.ToString("G", null);
   }

   public string ToString(string format)
   {
      return this.ToString(format, null);
   }

   public string ToString(string format, IFormatProvider provider)
   {
      // Handle null or empty arguments.
      if (String.IsNullOrEmpty(format))
         format = "G";
      // Remove any white space and covert to uppercase.
      format = format.Trim().ToUpperInvariant();

      if (provider == null)
         provider = NumberFormatInfo.CurrentInfo;

      switch (format)
      {
         // Convert temperature to Fahrenheit and return string.
         case "F":
            return this.Fahrenheit.ToString("N2", provider) + "°F";
         // Convert temperature to Kelvin and return string.
         case "K":
            return this.Kelvin.ToString("N2", provider) + "K";
         // Return temperature in Celsius.
         case "C":
         case "G":
            return this.Celsius.ToString("N2", provider) + "°C";
         default:
            throw new FormatException(String.Format("The '{0}' format string is not supported.", format));
      }
   }
}
```

```vb
Imports System.Globalization

Public Class Temperature : Implements IFormattable
   Private m_Temp As Decimal

   Public Sub New(temperature As Decimal)
      Me.m_Temp = temperature
   End Sub

   Public ReadOnly Property Celsius() As Decimal
      Get
         Return Me.m_Temp
      End Get
   End Property

   Public ReadOnly Property Kelvin() As Decimal
      Get
         Return Me.m_Temp + 273.15d
      End Get
   End Property

   Public ReadOnly Property Fahrenheit() As Decimal
      Get
         Return Math.Round(CDec(Me.m_Temp * 9 / 5 + 32), 2)
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return Me.ToString("G", Nothing)
   End Function

   Public Overloads Function ToString(format As String) As String
      Return Me.ToString(format, Nothing)
   End Function

   Public Overloads Function ToString(format As String, provider As IFormatProvider) As String _
      Implements IFormattable.ToString

      ' Handle null or empty arguments.
      If String.IsNullOrEmpty(format) Then format = "G"
      ' Remove any white space and convert to uppercase.
      format = format.Trim().ToUpperInvariant()

      If provider Is Nothing Then provider = NumberFormatInfo.CurrentInfo

      Select Case format
         ' Convert temperature to Fahrenheit and return string.
         Case "F"
            Return Me.Fahrenheit.ToString("N2", provider) & "°F"
         ' Convert temperature to Kelvin and return string.
         Case "K"
            Return Me.Kelvin.ToString("N2", provider) & "K"
         ' Return temperature in Celsius.
         Case "C", "G"
            Return Me.Celsius.ToString("N2", provider) & "°C"
         Case Else
            Throw New FormatException(String.Format("The '{0}' format string is not supported.", format))
      End Select
   End Function
End Class
```

The following example instantiates a `Temperature` object. It then calls the ToString method and uses several composite format strings to obtain different string representations of a `Temperature` object. Each of these method calls, in turn, calls the IFormattable implementation of the `Temperature` class.

```
public class Example
{
   public static void Main()
   {
      CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo("en-US");
      Temperature temp = new Temperature(22m);
      Console.WriteLine(Convert.ToString(temp, new CultureInfo("ja-JP")));
      Console.WriteLine("Temperature: {0:K}", temp);
      Console.WriteLine("Temperature: {0:F}", temp);
      Console.WriteLine(String.Format(new CultureInfo("fr-FR"), "Temperature: {0:F}", temp));
   }
}
// The example displays the following output:
//       22.00°C
//       Temperature: 295.15K
//       Temperature: 71.60°F
//       Temperature: 71,60°F
```

```
Public Module Example
   Public Sub Main()
      Dim temp As New Temperature(22d)
      CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo("en-US")
      Console.WriteLine(Convert.ToString(temp1, New CultureInfo("ja-JP")))
      Console.WriteLine("Temperature: {0:K}", temp)
      Console.WriteLine("Temperature: {0:F}", temp)
      Console.WriteLine(String.Format(New CultureInfo("fr-FR"), "Temperature: {0:F}", temp))
   End Sub
End Module
' The example displays the following output:
'       22.00°C
'       Temperature: 295.15K
'       Temperature: 71.60°F
'       Temperature: 71,60°F
```

Back to top

Composite Formatting

Some methods, such as String.Format and StringBuilder.AppendFormat, support composite formatting. A composite format string is a kind of template that returns a single string that incorporates the string representation of zero, one, or more objects. Each object is represented in the composite format string by an indexed format item. The index of the format item corresponds to the position of the object that it represents in the method's parameter list. Indexes are zero-based. For example, in the following call to the String.Format method, the first format item, `{0:D}`, is replaced by the string representation of `thatDate`; the second format item, `{1}`, is replaced by the string representation of `item1`; and the third format item, `{2:C2}`, is replaced by the string representation of `item1.Value`.

```
result = String.Format("On {0:d}, the inventory of {1} was worth {2:C2}.",
                       thatDate, item1, item1.Value);
Console.WriteLine(result);
// The example displays output like the following if run on a system
// whose current culture is en-US:
//       On 5/1/2009, the inventory of WidgetA was worth $107.44.
```

```
result = String.Format("On {0:d}, the inventory of {1} was worth {2:C2}.", _
                       thatDate, item1, item1.Value)
Console.WriteLine(result)
' The example displays output like the following if run on a system
' whose current culture is en-US:
'       On 5/1/2009, the inventory of WidgetA was worth $107.44.
```

In addition to replacing a format item with the string representation of its corresponding object, format items also let you control the following:

- The specific way in which an object is represented as a string, if the object implements the IFormattable interface and supports format strings. You do this by following the format item's index with a `:` (colon) followed by a valid format string. The previous example did this by formatting a date value with the "d" (short date pattern) format string (e.g., `{0:d}`) and by formatting a numeric value with the "C2" format string (e.g., `{2:C2}` to represent the number as a currency value with two fractional decimal digits.

- The width of the field that contains the object's string representation, and the alignment of the string representation in that field. You do this by following the format item's index with a `,` (comma) followed the field width. The string is right-aligned in the field if the field width is a positive value, and it is left-aligned if the field width is a negative value. The following example left-aligns date values in a 20-character field, and it right-aligns decimal values with one fractional digit in an 11-character field.

```
DateTime startDate = new DateTime(2015, 8, 28, 6, 0, 0);
decimal[] temps = { 73.452m, 68.98m, 72.6m, 69.24563m,
                   74.1m, 72.156m, 72.228m };
Console.WriteLine("{0,-20} {1,11}\n", "Date", "Temperature");
for (int ctr = 0; ctr < temps.Length; ctr++)
   Console.WriteLine("{0,-20:g} {1,11:N1}", startDate.AddDays(ctr), temps[ctr]);

// The example displays the following output:
//       Date                 Temperature
//
//       8/28/2015 6:00 AM           73.5
//       8/29/2015 6:00 AM           69.0
//       8/30/2015 6:00 AM           72.6
//       8/31/2015 6:00 AM           69.2
//       9/1/2015 6:00 AM            74.1
//       9/2/2015 6:00 AM            72.2
//       9/3/2015 6:00 AM            72.2
```

```
Dim startDate As New Date(2015, 8, 28, 6, 0, 0)
Dim temps() As Decimal = { 73.452, 68.98, 72.6, 69.24563,
                           74.1, 72.156, 72.228 }
Console.WriteLine("{0,-20} {1,11}", "Date", "Temperature")
Console.WriteLine()
For ctr As Integer = 0 To temps.Length - 1
   Console.WriteLine("{0,-20:g} {1,11:N1}", startDate.AddDays(ctr), temps(ctr))
Next
' The example displays the following output:
'       Date                 Temperature
'
'       8/28/2015 6:00 AM           73.5
'       8/29/2015 6:00 AM           69.0
'       8/30/2015 6:00 AM           72.6
'       8/31/2015 6:00 AM           69.2
'       9/1/2015 6:00 AM            74.1
'       9/2/2015 6:00 AM            72.2
'       9/3/2015 6:00 AM            72.2
```

Note that, if both the alignment string component and the format string component are present, the former precedes the latter (for example, `{0,-20:g}`.

For more information about composite formatting, see Composite Formatting.

Back to top

Custom Formatting with ICustomFormatter

Two composite formatting methods, String.Format(IFormatProvider, String, Object[]) and StringBuilder.AppendFormat(IFormatProvider, String, Object[]), include a format provider parameter that supports custom formatting. When either of these formatting methods is called, it passes a Type object that represents an ICustomFormatter interface to the format provider's GetFormat method. The GetFormat method is then responsible for returning the ICustomFormatter implementation that provides custom formatting.

The ICustomFormatter interface has a single method, Format(String, Object, IFormatProvider), that is called automatically by a composite formatting method, once for each format item in a composite format string. The Format(String, Object, IFormatProvider) method has three parameters: a format string, which represents the `formatString` argument in a format item, an object to format, and an IFormatProvider object that provides formatting services. Typically, the class that implements ICustomFormatter also implements IFormatProvider, so this last parameter is a reference to the custom formatting class itself. The method returns a custom formatted string representation of the object to be formatted. If the method cannot format the object, it should return a null reference (`Nothing` in Visual Basic).

The following example provides an ICustomFormatter implementation named `ByteByByteFormatter` that displays integer values as a sequence of two-digit hexadecimal values followed by a space.

```
public class ByteByByteFormatter : IFormatProvider, ICustomFormatter
{
   public object GetFormat(Type formatType)
   {
      if (formatType == typeof(ICustomFormatter))
         return this;
      else
         return null;
   }

   public string Format(string format, object arg,
                          IFormatProvider formatProvider)
   {
      if (! formatProvider.Equals(this)) return null;

      // Handle only hexadecimal format string.
      if (! format.StartsWith("X")) return null;

      byte[] bytes;
      string output = null;

      // Handle only integral types.
      if (arg is Byte)
         bytes = BitConverter.GetBytes((Byte) arg);
      else if (arg is Int16)
         bytes = BitConverter.GetBytes((Int16) arg);
      else if (arg is Int32)
         bytes = BitConverter.GetBytes((Int32) arg);
      else if (arg is Int64)
         bytes = BitConverter.GetBytes((Int64) arg);
      else if (arg is SByte)
         bytes = BitConverter.GetBytes((SByte) arg);
      else if (arg is UInt16)
         bytes = BitConverter.GetBytes((UInt16) arg);
      else if (arg is UInt32)
         bytes = BitConverter.GetBytes((UInt32) arg);
      else if (arg is UInt64)
         bytes = BitConverter.GetBytes((UInt64) arg);
      else
         return null;

      for (int ctr = bytes.Length - 1; ctr >= 0; ctr--)
         output += String.Format("{0:X2} ", bytes[ctr]);

      return output.Trim();
   }
}
```

```vb
Public Class ByteByByteFormatter : Implements IFormatProvider, ICustomFormatter
   Public Function GetFormat(formatType As Type) As Object _
                   Implements IFormatProvider.GetFormat
      If formatType Is GetType(ICustomFormatter) Then
         Return Me
      Else
         Return Nothing
      End If
   End Function

   Public Function Format(fmt As String, arg As Object,
                          formatProvider As IFormatProvider) As String _
                          Implements ICustomFormatter.Format

      If Not formatProvider.Equals(Me) Then Return Nothing

      ' Handle only hexadecimal format string.
      If Not fmt.StartsWith("X") Then
            Return Nothing
      End If

      ' Handle only integral types.
      If Not typeof arg Is Byte AndAlso
         Not typeof arg Is Int16 AndAlso
         Not typeof arg Is Int32 AndAlso
         Not typeof arg Is Int64 AndAlso
         Not typeof arg Is SByte AndAlso
         Not typeof arg Is UInt16 AndAlso
         Not typeof arg Is UInt32 AndAlso
         Not typeof arg Is UInt64 Then _
            Return Nothing

      Dim bytes() As Byte = BitConverter.GetBytes(arg)
      Dim output As String = Nothing

      For ctr As Integer = bytes.Length - 1 To 0 Step -1
         output += String.Format("{0:X2} ", bytes(ctr))
      Next

      Return output.Trim()
   End Function
End Class
```

The following example uses the `ByteByByteFormatter` class to format integer values. Note that the ICustomFormatter.Format method is called more than once in the second String.Format(IFormatProvider, String, Object[]) method call, and that the default NumberFormatInfo provider is used in the third method call because the . `ByteByByteFormatter.Format` method does not recognize the "N0" format string and returns a null reference (`Nothing` in Visual Basic).

```
public class Example
{
   public static void Main()
   {
      long value = 3210662321;
      byte value1 = 214;
      byte value2 = 19;

      Console.WriteLine(String.Format(new ByteByByteFormatter(), "{0:X}", value));
      Console.WriteLine(String.Format(new ByteByByteFormatter(), "{0:X} And {1:X} = {2:X} ({2:000})",
                                      value1, value2, value1 & value2));
      Console.WriteLine(String.Format(new ByteByByteFormatter(), "{0,10:N0}", value));
   }
}
// The example displays the following output:
//       00 00 00 00 BF 5E D1 B1
//       00 D6 And 00 13 = 00 12 (018)
//       3,210,662,321
```

```
Public Module Example
   Public Sub Main()
      Dim value As Long = 3210662321
      Dim value1 As Byte = 214
      Dim value2 As Byte = 19

      Console.WriteLine((String.Format(New ByteByByteFormatter(), "{0:X}", value)))
      Console.WriteLine((String.Format(New ByteByByteFormatter(), "{0:X} And {1:X} = {2:X} ({2:000})",
                                      value1, value2, value1 And value2)))
      Console.WriteLine(String.Format(New ByteByByteFormatter(), "{0,10:N0}", value))
   End Sub
End Module
' The example displays the following output:
'       00 00 00 00 BF 5E D1 B1
'       00 D6 And 00 13 = 00 12 (018)
'       3,210,662,321
```

Back to top

Related Topics

TITLE	DEFINITION
Standard Numeric Format Strings	Describes standard format strings that create commonly used string representations of numeric values.
Custom Numeric Format Strings	Describes custom format strings that create application-specific formats for numeric values.
Standard Date and Time Format Strings	Describes standard format strings that create commonly used string representations of DateTime values.
Custom Date and Time Format Strings	Describes custom format strings that create application-specific formats for DateTime values.
Standard TimeSpan Format Strings	Describes standard format strings that create commonly used string representations of time intervals.
Custom TimeSpan Format Strings	Describes custom format strings that create application-specific formats for time intervals.

TITLE	DEFINITION
Enumeration Format Strings	Describes standard format strings that are used to create string representations of enumeration values.
Composite Formatting	Describes how to embed one or more formatted values in a string. The string can subsequently be displayed on the console or written to a stream.
Performing Formatting Operations	Lists topics that provide step-by-step instructions for performing specific formatting operations.
Parsing Strings	Describes how to initialize objects to the values described by string representations of those objects. Parsing is the inverse operation of formatting.

Back to top

Reference

System.IFormattable

System.IFormatProvider

System.ICustomFormatter

Standard Numeric Format Strings

9/6/2018 • 26 minutes to read • Edit Online

Standard numeric format strings are used to format common numeric types. A standard numeric format string takes the form `Axx`, where:

- `A` is a single alphabetic character called the *format specifier*. Any numeric format string that contains more than one alphabetic character, including white space, is interpreted as a custom numeric format string. For more information, see Custom Numeric Format Strings.

- `xx` is an optional integer called the *precision specifier*. The precision specifier ranges from 0 to 99 and affects the number of digits in the result. Note that the precision specifier controls the number of digits in the string representation of a number. It does not round the number itself. To perform a rounding operation, use the Math.Ceiling, Math.Floor, or Math.Round method.

  When *precision specifier* controls the number of fractional digits in the result string, the result string reflects a number that is rounded to a representable result nearest to the infinitely precise result. If there are two equally near representable results:

  - **On the .NET Framework and .NET Core up to .NET Core 2.0**, the runtime selects the result with the greater least significant digit (that is, using MidpointRounding.AwayFromZero).
  - **On .NET Core 2.1 and later**, the runtime selects the result with an even least significant digit (that is, using MidpointRounding.ToEven).

> **NOTE**
>
> The precision specifier determines the number of digits in the result string. To pad a result string with leading or trailing spaces, use the composite formatting feature and define an *alignment component* in the format item.

Standard numeric format strings are supported by:

- Some overloads of the `ToString` method of all numeric types. For example, you can supply a numeric format string to the Int32.ToString(String) and Int32.ToString(String, IFormatProvider) methods.

- The .NET composite formatting feature, which is used by some `Write` and `WriteLine` methods of the Console and StreamWriter classes, the String.Format method, and the StringBuilder.AppendFormat method. The composite format feature allows you to include the string representation of multiple data items in a single string, to specify field width, and to align numbers in a field. For more information, see Composite Formatting.

- Interpolated strings in C# and Visual Basic, which provide a simplified syntax when compared to composite format strings.

> **TIP**
>
> You can download the Formatting Utility, an application that enables you to apply format strings to either numeric or date and time values and displays the result string.

The following table describes the standard numeric format specifiers and displays sample output produced by each format specifier. See the Notes section for additional information about using standard numeric format strings, and the Example section for a comprehensive illustration of their use.

FORMAT SPECIFIER	NAME	DESCRIPTION	EXAMPLES
"C" or "c"	Currency	Result: A currency value. Supported by: All numeric types. Precision specifier: Number of decimal digits. Default precision specifier: Defined by NumberFormatInfo.CurrencyDecimalDigits. More information: The Currency ("C") Format Specifier.	123.456 ("C", en-US) -> $123.46 123.456 ("C", fr-FR) -> 123,46 € 123.456 ("C", ja-JP) -> ¥123 -123.456 ("C3", en-US) -> ($123.456) -123.456 ("C3", fr-FR) -> -123,456 € -123.456 ("C3", ja-JP) -> -¥123.456
"D" or "d"	Decimal	Result: Integer digits with optional negative sign. Supported by: Integral types only. Precision specifier: Minimum number of digits. Default precision specifier: Minimum number of digits required. More information: The Decimal("D") Format Specifier.	1234 ("D") -> 1234 -1234 ("D6") -> -001234
"E" or "e"	Exponential (scientific)	Result: Exponential notation. Supported by: All numeric types. Precision specifier: Number of decimal digits. Default precision specifier: 6. More information: The Exponential ("E") Format Specifier.	1052.0329112756 ("E", en-US) -> 1.052033E+003 1052.0329112756 ("e", fr-FR) -> 1,052033e+003 -1052.0329112756 ("e2", en-US) -> -1.05e+003 -1052.0329112756 ("E2", fr-FR) -> -1,05E+003

FORMAT SPECIFIER	NAME	DESCRIPTION	EXAMPLES
"F" or "f"	Fixed-point	Result: Integral and decimal digits with optional negative sign. Supported by: All numeric types. Precision specifier: Number of decimal digits. Default precision specifier: Defined by NumberFormatInfo.NumberDecimalDigits. More information: The Fixed-Point ("F") Format Specifier.	1234.567 ("F", en-US) -> 1234.57 1234.567 ("F", de-DE) -> 1234,57 1234 ("F1", en-US) -> 1234.0 1234 ("F1", de-DE) -> 1234,0 -1234.56 ("F4", en-US) -> -1234.5600 -1234.56 ("F4", de-DE) -> -1234,5600
"G" or "g"	General	Result: The more compact of either fixed-point or scientific notation. Supported by: All numeric types. Precision specifier: Number of significant digits. Default precision specifier: Depends on numeric type. More information: The General ("G") Format Specifier.	-123.456 ("G", en-US) -> -123.456 -123.456 ("G", sv-SE) -> -123,456 123.4546 ("G4", en-US) -> 123.5 123.4546 ("G4", sv-SE) -> 123,5 -1.234567890e-25 ("G", en-US) -> -1.23456789E-25 -1.234567890e-25 ("G", sv-SE) -> -1,23456789E-25
"N" or "n"	Number	Result: Integral and decimal digits, group separators, and a decimal separator with optional negative sign. Supported by: All numeric types. Precision specifier: Desired number of decimal places. Default precision specifier: Defined by NumberFormatInfo.NumberDecimalDigits. More information: The Numeric ("N") Format Specifier.	1234.567 ("N", en-US) -> 1,234.57 1234.567 ("N", ru-RU) -> 1 234,57 1234 ("N1", en-US) -> 1,234.0 1234 ("N1", ru-RU) -> 1 234,0 -1234.56 ("N3", en-US) -> -1,234.560 -1234.56 ("N3", ru-RU) -> -1 234,560

FORMAT SPECIFIER	NAME	DESCRIPTION	EXAMPLES
"P" or "p"	Percent	Result: Number multiplied by 100 and displayed with a percent symbol. Supported by: All numeric types. Precision specifier: Desired number of decimal places. Default precision specifier: Defined by NumberFormatInfo.PercentDecimalDigits. More information: The Percent ("P") Format Specifier.	1 ("P", en-US) -> 100.00 % 1 ("P", fr-FR) -> 100,00 % -0.39678 ("P1", en-US) -> -39.7 % -0.39678 ("P1", fr-FR) -> -39,7 %
"R" or "r"	Round-trip	Result: A string that can round-trip to an identical number. Supported by: Single, Double, and BigInteger. Note: Recommended for the BigInteger type only. For Double types, use "G17"; for Single types, use "G9". Precision specifier: Ignored. More information: The Round-trip ("R") Format Specifier.	123456789.12345678 ("R") -> 123456789.12345678 -1234567890.12345678 ("R") -> -1234567890.1234567
"X" or "x"	Hexadecimal	Result: A hexadecimal string. Supported by: Integral types only. Precision specifier: Number of digits in the result string. More information: The HexaDecimal ("X") Format Specifier.	255 ("X") -> FF -1 ("x") -> ff 255 ("x4") -> 00ff -1 ("X4") -> 00FF
Any other single character	Unknown specifier	Result: Throws a FormatException at run time.	

Using Standard Numeric Format Strings

NOTE

The 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.

A standard numeric format string can be used to define the formatting of a numeric value in one of two ways:

- It can be passed to an overload of the `ToString` method that has a `format` parameter. The following example formats a numeric value as a currency string in the current culture (in this case, the en-US culture).

```
Decimal value = static_cast<Decimal>(123.456);
Console::WriteLine(value.ToString("C2"));
// Displays $123.46
```

```
decimal value = 123.456m;
Console.WriteLine(value.ToString("C2"));
// Displays $123.46
```

```
Dim value As Decimal = 123.456d
Console.WriteLine(value.ToString("C2"))
' Displays $123.46
```

- It can be supplied as the `formatString` argument in a format item used with such methods as String.Format, Console.WriteLine, and StringBuilder.AppendFormat. For more information, see Composite Formatting. The following example uses a format item to insert a currency value in a string.

```
Decimal value = static_cast<Decimal>(123.456);
Console::WriteLine("Your account balance is {0:C2}.", value);
// Displays "Your account balance is $123.46."
```

```
decimal value = 123.456m;
Console.WriteLine("Your account balance is {0:C2}.", value);
// Displays "Your account balance is $123.46."
```

```
Dim value As Decimal = 123.456d
Console.WriteLine("Your account balance is {0:C2}.", value)
' Displays "Your account balance is $123.46."
```

Optionally, you can supply an `alignment` argument to specify the width of the numeric field and whether its value is right- or left-aligned. The following example left-aligns a currency value in a 28-character field, and it right-aligns a currency value in a 14-character field.

```
array<Decimal>^ amounts = { static_cast<Decimal>(16305.32), 
                            static_cast<Decimal>(18794.16) };
Console::WriteLine("   Beginning Balance           Ending Balance");
Console::WriteLine("   {0,-28:C2}{1,14:C2}", amounts[0], amounts[1]);
// Displays:
//        Beginning Balance           Ending Balance
//        $16,305.32                      $18,794.16
```

```
decimal[] amounts = { 16305.32m, 18794.16m };
Console.WriteLine("   Beginning Balance           Ending Balance");
Console.WriteLine("   {0,-28:C2}{1,14:C2}", amounts[0], amounts[1]);
// Displays:
//        Beginning Balance           Ending Balance
//        $16,305.32                      $18,794.16
```

```
Dim amounts() As Decimal = { 16305.32d, 18794.16d }
Console.WriteLine("   Beginning Balance           Ending Balance")
Console.WriteLine("   {0,-28:C2}{1,14:C2}", amounts(0), amounts(1))
' Displays:
'        Beginning Balance           Ending Balance
'        $16,305.32                      $18,794.16
```

- It can be supplied as the `formatString` argument in an interpolated expression item of an interpolated string. For more information, see the String interpolation topic in the C# reference or the Interpolated strings topic in the Visual Basic reference.

The following sections provide detailed information about each of the standard numeric format strings.

The Currency ("C") Format Specifier

The "C" (or currency) format specifier converts a number to a string that represents a currency amount. The precision specifier indicates the desired number of decimal places in the result string. If the precision specifier is omitted, the default precision is defined by the NumberFormatInfo.CurrencyDecimalDigits property.

If the value to be formatted has more than the specified or default number of decimal places, the fractional value is rounded in the result string. If the value to the right of the number of specified decimal places is 5 or greater, the last digit in the result string is rounded away from zero.

The result string is affected by the formatting information of the current NumberFormatInfo object. The following table lists the NumberFormatInfo properties that control the formatting of the returned string.

NUMBERFORMATINFO PROPERTY	DESCRIPTION
CurrencyPositivePattern	Defines the placement of the currency symbol for positive values.
CurrencyNegativePattern	Defines the placement of the currency symbol for negative values, and specifies whether the negative sign is represented by parentheses or the NegativeSign property.
NegativeSign	Defines the negative sign used if CurrencyNegativePattern indicates that parentheses are not used.
CurrencySymbol	Defines the currency symbol.

NUMBERFORMATINFO PROPERTY	DESCRIPTION
CurrencyDecimalDigits	Defines the default number of decimal digits in a currency value. This value can be overridden by using the precision specifier.
CurrencyDecimalSeparator	Defines the string that separates integral and decimal digits.
CurrencyGroupSeparator	Defines the string that separates groups of integral numbers.
CurrencyGroupSizes	Defines the number of integer digits that appear in a group.

The following example formats a Double value with the currency format specifier.

```
double value = 12345.6789;
Console::WriteLine(value.ToString("C", CultureInfo::CurrentCulture));

Console::WriteLine(value.ToString("C3", CultureInfo::CurrentCulture));

Console::WriteLine(value.ToString("C3",
                  CultureInfo::CreateSpecificCulture("da-DK")));
// The example displays the following output on a system whose
// current culture is English (United States):
//       $12,345.68
//       $12,345.679
//       kr 12.345,679
```

```
double value = 12345.6789;
Console.WriteLine(value.ToString("C", CultureInfo.CurrentCulture));

Console.WriteLine(value.ToString("C3", CultureInfo.CurrentCulture));

Console.WriteLine(value.ToString("C3",
                  CultureInfo.CreateSpecificCulture("da-DK")));
// The example displays the following output on a system whose
// current culture is English (United States):
//       $12,345.68
//       $12,345.679
//       kr 12.345,679
```

```
Dim value As Double = 12345.6789
Console.WriteLine(value.ToString("C", CultureInfo.CurrentCulture))

Console.WriteLine(value.ToString("C3", CultureInfo.CurrentCulture))

Console.WriteLine(value.ToString("C3", _
                  CultureInfo.CreateSpecificCulture("da-DK")))
' The example displays the following output on a system whose
' current culture is English (United States):
'       $12,345.68
'       $12,345.679
'       kr 12.345,679
```

Back to table

The Decimal ("D") Format Specifier

The "D" (or decimal) format specifier converts a number to a string of decimal digits (0-9), prefixed by a minus sign if the number is negative. This format is supported only for integral types.

The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier. If no precision specifier is specified, the default is the minimum value required to represent the integer without leading zeros.

The result string is affected by the formatting information of the current NumberFormatInfo object. As the following table shows, a single property affects the formatting of the result string.

NUMBERFORMATINFO PROPERTY	DESCRIPTION
NegativeSign	Defines the string that indicates that a number is negative.

The following example formats an Int32 value with the decimal format specifier.

```
int value;

value = 12345;
Console::WriteLine(value.ToString("D"));
// Displays 12345
Console::WriteLine(value.ToString("D8"));
// Displays 00012345

value = -12345;
Console::WriteLine(value.ToString("D"));
// Displays -12345
Console::WriteLine(value.ToString("D8"));
// Displays -00012345
```

```
int value;

value = 12345;
Console.WriteLine(value.ToString("D"));
// Displays 12345
Console.WriteLine(value.ToString("D8"));
// Displays 00012345

value = -12345;
Console.WriteLine(value.ToString("D"));
// Displays -12345
Console.WriteLine(value.ToString("D8"));
// Displays -00012345
```

```
Dim value As Integer

value = 12345
Console.WriteLine(value.ToString("D"))
' Displays 12345
Console.WriteLine(value.ToString("D8"))
' Displays 00012345

value = -12345
Console.WriteLine(value.ToString("D"))
' Displays -12345
Console.WriteLine(value.ToString("D8"))
' Displays -00012345
```

Back to table

The Exponential ("E") Format Specifier

The exponential ("E") format specifier converts a number to a string of the form "-d.ddd…E+ddd" or "-d.ddd…e+ddd", where each "d" indicates a digit (0-9). The string starts with a minus sign if the number is negative. Exactly one digit always precedes the decimal point.

The precision specifier indicates the desired number of digits after the decimal point. If the precision specifier is omitted, a default of six digits after the decimal point is used.

The case of the format specifier indicates whether to prefix the exponent with an "E" or an "e". The exponent always consists of a plus or minus sign and a minimum of three digits. The exponent is padded with zeros to meet this minimum, if required.

The result string is affected by the formatting information of the current NumberFormatInfo object. The following table lists the NumberFormatInfo properties that control the formatting of the returned string.

NUMBERFORMATINFO PROPERTY	DESCRIPTION
NegativeSign	Defines the string that indicates that a number is negative for both the coefficient and exponent.
NumberDecimalSeparator	Defines the string that separates the integral digit from decimal digits in the coefficient.
PositiveSign	Defines the string that indicates that an exponent is positive.

The following example formats a Double value with the exponential format specifier.

```
double value = 12345.6789;
Console::WriteLine(value.ToString("E", CultureInfo::InvariantCulture));
// Displays 1.234568E+004

Console::WriteLine(value.ToString("E10", CultureInfo::InvariantCulture));
// Displays 1.2345678900E+004

Console::WriteLine(value.ToString("e4", CultureInfo::InvariantCulture));
// Displays 1.2346e+004

Console::WriteLine(value.ToString("E",
                  CultureInfo::CreateSpecificCulture("fr-FR")));
// Displays 1,234568E+004
```

```
double value = 12345.6789;
Console.WriteLine(value.ToString("E", CultureInfo.InvariantCulture));
// Displays 1.234568E+004

Console.WriteLine(value.ToString("E10", CultureInfo.InvariantCulture));
// Displays 1.2345678900E+004

Console.WriteLine(value.ToString("e4", CultureInfo.InvariantCulture));
// Displays 1.2346e+004

Console.WriteLine(value.ToString("E",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 1,234568E+004
```

```
Dim value As Double = 12345.6789
Console.WriteLine(value.ToString("E", CultureInfo.InvariantCulture))
' Displays 1.234568E+004

Console.WriteLine(value.ToString("E10", CultureInfo.InvariantCulture))
' Displays 1.2345678900E+004

Console.WriteLine(value.ToString("e4", CultureInfo.InvariantCulture))
' Displays 1.2346e+004

Console.WriteLine(value.ToString("E", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays 1,234568E+004
```

Back to table

The Fixed-Point ("F") Format Specifier

The fixed-point ("F") format specifier converts a number to a string of the form "-ddd.ddd…" where each "d" indicates a digit (0-9). The string starts with a minus sign if the number is negative.

The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the current NumberFormatInfo.NumberDecimalDigits property supplies the numeric precision.

The result string is affected by the formatting information of the current NumberFormatInfo object. The following table lists the properties of the NumberFormatInfo object that control the formatting of the result string.

NUMBERFORMATINFO PROPERTY	DESCRIPTION
NegativeSign	Defines the string that indicates that a number is negative.
NumberDecimalSeparator	Defines the string that separates integral digits from decimal digits.
NumberDecimalDigits	Defines the default number of decimal digits. This value can be overridden by using the precision specifier.

The following example formats a Double and an Int32 value with the fixed-point format specifier.

```cpp
int integerNumber;
integerNumber = 17843;
Console::WriteLine(integerNumber.ToString("F",
                  CultureInfo::InvariantCulture));
// Displays 17843.00

integerNumber = -29541;
Console::WriteLine(integerNumber.ToString("F3",
                  CultureInfo::InvariantCulture));
// Displays -29541.000

double doubleNumber;
doubleNumber = 18934.1879;
Console::WriteLine(doubleNumber.ToString("F", CultureInfo::InvariantCulture));
// Displays 18934.19

Console::WriteLine(doubleNumber.ToString("F0", CultureInfo::InvariantCulture));
// Displays 18934

doubleNumber = -1898300.1987;
Console::WriteLine(doubleNumber.ToString("F1", CultureInfo::InvariantCulture));
// Displays -1898300.2

Console::WriteLine(doubleNumber.ToString("F3",
                  CultureInfo::CreateSpecificCulture("es-ES")));
// Displays -1898300,199
```

```csharp
int integerNumber;
integerNumber = 17843;
Console.WriteLine(integerNumber.ToString("F",
                  CultureInfo.InvariantCulture));
// Displays 17843.00

integerNumber = -29541;
Console.WriteLine(integerNumber.ToString("F3",
                  CultureInfo.InvariantCulture));
// Displays -29541.000

double doubleNumber;
doubleNumber = 18934.1879;
Console.WriteLine(doubleNumber.ToString("F", CultureInfo.InvariantCulture));
// Displays 18934.19

Console.WriteLine(doubleNumber.ToString("F0", CultureInfo.InvariantCulture));
// Displays 18934

doubleNumber = -1898300.1987;
Console.WriteLine(doubleNumber.ToString("F1", CultureInfo.InvariantCulture));
// Displays -1898300.2

Console.WriteLine(doubleNumber.ToString("F3",
                  CultureInfo.CreateSpecificCulture("es-ES")));
// Displays -1898300,199
```

```
Dim integerNumber As Integer
integerNumber = 17843
Console.WriteLine(integerNumber.ToString("F", CultureInfo.InvariantCulture))
' Displays 17843.00

integerNumber = -29541
Console.WriteLine(integerNumber.ToString("F3", CultureInfo.InvariantCulture))
' Displays -29541.000

Dim doubleNumber As Double
doubleNumber = 18934.1879
Console.WriteLine(doubleNumber.ToString("F", CultureInfo.InvariantCulture))
' Displays 18934.19

Console.WriteLine(doubleNumber.ToString("F0", CultureInfo.InvariantCulture))
' Displays 18934

doubleNumber = -1898300.1987
Console.WriteLine(doubleNumber.ToString("F1", CultureInfo.InvariantCulture))
' Displays -1898300.2

Console.WriteLine(doubleNumber.ToString("F3", _
                  CultureInfo.CreateSpecificCulture("es-ES")))
' Displays -1898300,199
```

Back to table

The General ("G") Format Specifier

The general ("G") format specifier converts a number to the more compact of either fixed-point or scientific notation, depending on the type of the number and whether a precision specifier is present. The precision specifier defines the maximum number of significant digits that can appear in the result string. If the precision specifier is omitted or zero, the type of the number determines the default precision, as indicated in the following table.

NUMERIC TYPE	DEFAULT PRECISION
Byte or SByte	3 digits
Int16 or UInt16	5 digits
Int32 or UInt32	10 digits
Int64	19 digits
UInt64	20 digits
BigInteger	Unlimited (same as "R")
Single	7 digits
Double	15 digits
Decimal	29 digits

Fixed-point notation is used if the exponent that would result from expressing the number in scientific notation is greater than -5 and less than the precision specifier; otherwise, scientific notation is used. The result contains a decimal point if required, and trailing zeros after the decimal point are omitted. If the precision specifier is present and the number of significant digits in the result exceeds the specified precision, the excess trailing digits are removed by rounding.

However, if the number is a Decimal and the precision specifier is omitted, fixed-point notation is always used and trailing zeros are preserved.

If scientific notation is used, the exponent in the result is prefixed with "E" if the format specifier is "G", or "e" if the format specifier is "g". The exponent contains a minimum of two digits. This differs from the format for scientific notation that is produced by the exponential format specifier, which includes a minimum of three digits in the exponent.

Note that, when used with a Double value, the "G17" format specifier ensures that the original Double value successfully round-trips. This is because Double is an IEEE 754-2008-compliant double-precision (`binary64`) floating point number that gives up to 17 significant digits of precision. We recommend its use instead of the "R" format specifier, since in some cases "R" fails to successfully round-trip double-precision floating point values. The following example illustrates one such case.

```
using System;

public class Example
{
   public static void Main()
   {
      double original = 0.84551240822557006;
      var rSpecifier = original.ToString("R");
      var g17Specifier = original.ToString("G17");

      var rValue = Double.Parse(rSpecifier);
      var g17Value = Double.Parse(g17Specifier);

      Console.WriteLine($"{original:G17} = {rSpecifier} (R): {original.Equals(rValue)}");
      Console.WriteLine($"{original:G17} = {g17Specifier} (G17): {original.Equals(g17Value)}");
   }
}
// The example displays the following output:
//     0.84551240822557006 = 0.84551240822557: False
//     0.84551240822557006 = 0.84551240822557006: True
```

```
Module Example
   Public Sub Main()
      Dim original As Double = 0.84551240822557006
      Dim rSpecifier = original.ToString("R")
      Dim g17Specifier = original.ToString("G17")

      Dim rValue = Double.Parse(rSpecifier)
      Dim g17Value = Double.Parse(g17Specifier)

      Console.WriteLine($"{original:G17} = {rSpecifier} (R): {original.Equals(rValue)}")
      Console.WriteLine($"{original:G17} = {g17Specifier} (G17): {original.Equals(g17Value)}")
   End Sub
End Module
' The example displays the following output:
'     0.84551240822557006 = 0.84551240822557 (R): False
'     0.84551240822557006 = 0.84551240822557006 (G17): True
```

When used with a Single value, the "G9" format specifier ensures that the original Single value successfully round-trips. This is because Single is an IEEE 754-2008-compliant single-precision (`binary32`) floating point number that gives up to nine significant digits of precision. For performance reasons, we recommend its use instead of the "R" format specifier.

The result string is affected by the formatting information of the current NumberFormatInfo object. The following table lists the NumberFormatInfo properties that control the formatting of the result string.

NUMBERFORMATINFO PROPERTY	DESCRIPTION
NegativeSign	Defines the string that indicates that a number is negative.
NumberDecimalSeparator	Defines the string that separates integral digits from decimal digits.
PositiveSign	Defines the string that indicates that an exponent is positive.

The following example formats assorted floating-point values with the general format specifier.

```
double number;

number = 12345.6789;
Console::WriteLine(number.ToString("G", CultureInfo::InvariantCulture));
// Displays  12345.6789
Console::WriteLine(number.ToString("G",
                  CultureInfo::CreateSpecificCulture("fr-FR")));
// Displays 12345,6789

Console::WriteLine(number.ToString("G7", CultureInfo::InvariantCulture));
// Displays 12345.68

number = .0000023;
Console::WriteLine(number.ToString("G", CultureInfo::InvariantCulture));
// Displays 2.3E-06
Console::WriteLine(number.ToString("G",
                  CultureInfo::CreateSpecificCulture("fr-FR")));
// Displays 2,3E-06

number = .0023;
Console::WriteLine(number.ToString("G", CultureInfo::InvariantCulture));
// Displays 0.0023

number = 1234;
Console::WriteLine(number.ToString("G2", CultureInfo::InvariantCulture));
// Displays 1.2E+03

number = Math::PI;
Console::WriteLine(number.ToString("G5", CultureInfo::InvariantCulture));
// Displays 3.1416
```

```
double number;

number = 12345.6789;
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture));
// Displays  12345.6789
Console.WriteLine(number.ToString("G",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 12345,6789

Console.WriteLine(number.ToString("G7", CultureInfo.InvariantCulture));
// Displays 12345.68

number = .0000023;
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture));
// Displays 2.3E-06
Console.WriteLine(number.ToString("G",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 2,3E-06

number = .0023;
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture));
// Displays 0.0023

number = 1234;
Console.WriteLine(number.ToString("G2", CultureInfo.InvariantCulture));
// Displays 1.2E+03

number = Math.PI;
Console.WriteLine(number.ToString("G5", CultureInfo.InvariantCulture));
// Displays 3.1416
```

```
Dim number As Double

number = 12345.6789
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture))
' Displays  12345.6789
Console.WriteLine(number.ToString("G", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays 12345,6789

Console.WriteLine(number.ToString("G7", CultureInfo.InvariantCulture))
' Displays 12345.68

number = .0000023
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture))
' Displays 2.3E-06
Console.WriteLine(number.ToString("G", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays 2,3E-06

number = .0023
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture))
' Displays 0.0023

number = 1234
Console.WriteLine(number.ToString("G2", CultureInfo.InvariantCulture))
' Displays 1.2E+03

number = Math.Pi
Console.WriteLine(number.ToString("G5", CultureInfo.InvariantCulture))
' Displays 3.1416
```

Back to table

The Numeric ("N") Format Specifier

The numeric ("N") format specifier converts a number to a string of the form "-d,ddd,ddd.ddd…", where "-" indicates a negative number symbol if required, "d" indicates a digit (0-9), "," indicates a group separator, and "." indicates a decimal point symbol. The precision specifier indicates the desired number of digits after the decimal point. If the precision specifier is omitted, the number of decimal places is defined by the current NumberFormatInfo.NumberDecimalDigits property.

The result string is affected by the formatting information of the current NumberFormatInfo object. The following table lists the NumberFormatInfo properties that control the formatting of the result string.

NUMBERFORMATINFO PROPERTY	DESCRIPTION
NegativeSign	Defines the string that indicates that a number is negative.
NumberNegativePattern	Defines the format of negative values, and specifies whether the negative sign is represented by parentheses or the NegativeSign property.
NumberGroupSizes	Defines the number of integral digits that appear between group separators.
NumberGroupSeparator	Defines the string that separates groups of integral numbers.
NumberDecimalSeparator	Defines the string that separates integral and decimal digits.
NumberDecimalDigits	Defines the default number of decimal digits. This value can be overridden by using a precision specifier.

The following example formats assorted floating-point values with the number format specifier.

```
double dblValue = -12445.6789;
Console::WriteLine(dblValue.ToString("N", CultureInfo::InvariantCulture));
// Displays -12,445.68
Console::WriteLine(dblValue.ToString("N1",
                  CultureInfo::CreateSpecificCulture("sv-SE")));
// Displays -12 445,7

int intValue = 123456789;
Console::WriteLine(intValue.ToString("N1", CultureInfo::InvariantCulture));
// Displays 123,456,789.0
```

```
double dblValue = -12445.6789;
Console.WriteLine(dblValue.ToString("N", CultureInfo.InvariantCulture));
// Displays -12,445.68
Console.WriteLine(dblValue.ToString("N1",
                  CultureInfo.CreateSpecificCulture("sv-SE")));
// Displays -12 445,7

int intValue = 123456789;
Console.WriteLine(intValue.ToString("N1", CultureInfo.InvariantCulture));
// Displays 123,456,789.0
```

```
Dim dblValue As Double = -12445.6789
Console.WriteLine(dblValue.ToString("N", CultureInfo.InvariantCulture))
' Displays -12,445.68
Console.WriteLine(dblValue.ToString("N1", _
                  CultureInfo.CreateSpecificCulture("sv-SE")))
' Displays -12 445,7

Dim intValue As Integer = 123456789
Console.WriteLine(intValue.ToString("N1", CultureInfo.InvariantCulture))
' Displays 123,456,789.0
```

Back to table

The Percent ("P") Format Specifier

The percent ("P") format specifier multiplies a number by 100 and converts it to a string that represents a percentage. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision supplied by the current PercentDecimalDigits property is used.

The following table lists the NumberFormatInfo properties that control the formatting of the returned string.

NUMBERFORMATINFO PROPERTY	DESCRIPTION
PercentPositivePattern	Defines the placement of the percent symbol for positive values.
PercentNegativePattern	Defines the placement of the percent symbol and the negative symbol for negative values.
NegativeSign	Defines the string that indicates that a number is negative.
PercentSymbol	Defines the percent symbol.
PercentDecimalDigits	Defines the default number of decimal digits in a percentage value. This value can be overridden by using the precision specifier.
PercentDecimalSeparator	Defines the string that separates integral and decimal digits.
PercentGroupSeparator	Defines the string that separates groups of integral numbers.
PercentGroupSizes	Defines the number of integer digits that appear in a group.

The following example formats floating-point values with the percent format specifier.

```
double number = .2468013;
Console::WriteLine(number.ToString("P", CultureInfo::InvariantCulture));
// Displays 24.68 %
Console::WriteLine(number.ToString("P",
                  CultureInfo::CreateSpecificCulture("hr-HR")));
// Displays 24,68%
Console::WriteLine(number.ToString("P1", CultureInfo::InvariantCulture));
// Displays 24.7 %
```

```
double number = .2468013;
Console.WriteLine(number.ToString("P", CultureInfo.InvariantCulture));
// Displays 24.68 %
Console.WriteLine(number.ToString("P",
                  CultureInfo.CreateSpecificCulture("hr-HR")));
// Displays 24,68%
Console.WriteLine(number.ToString("P1", CultureInfo.InvariantCulture));
// Displays 24.7 %
```

```
Dim number As Double = .2468013
Console.WriteLine(number.ToString("P", CultureInfo.InvariantCulture))
' Displays 24.68 %
Console.WriteLine(number.ToString("P", _
                  CultureInfo.CreateSpecificCulture("hr-HR")))
' Displays 24,68%
Console.WriteLine(number.ToString("P1", CultureInfo.InvariantCulture))
' Displays 24.7 %
```

Back to table

The Round-trip ("R") Format Specifier

The round-trip ("R") format specifier attempts to ensure that a numeric value that is converted to a string is parsed back into the same numeric value. This format is supported only for the Single, Double, and BigInteger types.

For Double values, the "R" format specifier in some cases fails to successfully round-trip the original value. For both Double and Single values, it also offers relatively poor performance. Instead, we recommend that you use the "G17" format specifier for Double values and the "G9" format specifier to successfully round-trip Single values.

When a BigInteger value is formatted using this specifier, its string representation contains all the significant digits in the BigInteger value.

Although you can include a precision specifier, it is ignored. Round trips are given precedence over precision when using this specifier.
The result string is affected by the formatting information of the current NumberFormatInfo object. The following table lists the NumberFormatInfo properties that control the formatting of the result string.

NUMBERFORMATINFO PROPERTY	DESCRIPTION
NegativeSign	Defines the string that indicates that a number is negative.
NumberDecimalSeparator	Defines the string that separates integral digits from decimal digits.
PositiveSign	Defines the string that indicates that an exponent is positive.

The following example formats a BigInteger value with the round-trip format specifier.

```
#using <System.Numerics.dll>

using namespace System;
using namespace System::Numerics;

void main()
{
   BigInteger value = BigInteger::Pow(Int64::MaxValue, 2);
   Console::WriteLine(value.ToString("R"));
}
// The example displays the following output:
//      85070591730234615847396907784232501249
```

```
using System;
using System.Numerics;

public class Example
{
   public static void Main()
   {
      var value = BigInteger.Pow(Int64.MaxValue, 2);
      Console.WriteLine(value.ToString("R"));
   }
}                                                                                        // The example
displays the following output:
//      85070591730234615847396907784232501249
```

```
Imports System.Numerics

Module Example
   Public Sub Main()
      Dim value = BigInteger.Pow(Int64.MaxValue, 2)
      Console.WriteLine(value.ToString("R"))
   End Sub
End Module
' The example displays the following output:
'      85070591730234615847396907784232501249
```

IMPORTANT

In some cases, Double values formatted with the "R" standard numeric format string do not successfully round-trip if compiled using the `/platform:x64` or `/platform:anycpu` switches and run on 64-bit systems. See the following paragraph for more information.

To work around the problem of Double values formatted with the "R" standard numeric format string not successfully round-tripping if compiled using the `/platform:x64` or `/platform:anycpu` switches and run on 64-bit systems., you can format Double values by using the "G17" standard numeric format string. The following example uses the "R" format string with a Double value that does not round-trip successfully, and also uses the "G17" format string to successfully round-trip the original value.

```csharp
using System;
using System.Globalization;

public class Example
{
   static void Main(string[] args)
   {
      Console.WriteLine("Attempting to round-trip a Double with 'R':");
      double initialValue = 0.6822871999174;
      string valueString = initialValue.ToString("R",
                                               CultureInfo.InvariantCulture);
      double roundTripped = double.Parse(valueString,
                                         CultureInfo.InvariantCulture);
      Console.WriteLine("{0:R} = {1:R}: {2}\n",
                        initialValue, roundTripped, initialValue.Equals(roundTripped));

      Console.WriteLine("Attempting to round-trip a Double with 'G17':");
      string valueString17 = initialValue.ToString("G17",
                                                 CultureInfo.InvariantCulture);
      double roundTripped17 = double.Parse(valueString17,
                                           CultureInfo.InvariantCulture);
      Console.WriteLine("{0:R} = {1:R}: {2}\n",
                        initialValue, roundTripped17, initialValue.Equals(roundTripped17));
   }
}
// If compiled to an application that targets anycpu or x64 and run on an x64 system,
// the example displays the following output:
//       Attempting to round-trip a Double with 'R':
//       0.6822871999174 = 0.68228719991740006: False
//
//       Attempting to round-trip a Double with 'G17':
//       0.6822871999174 = 0.6822871999174: True
```

```vb
Imports System.Globalization

Module Example
   Public Sub Main()
      Console.WriteLine("Attempting to round-trip a Double with 'R':")
      Dim initialValue As Double = 0.6822871999174
      Dim valueString As String = initialValue.ToString("R",
                                               CultureInfo.InvariantCulture)
      Dim roundTripped As Double = Double.Parse(valueString,
                                                CultureInfo.InvariantCulture)
      Console.WriteLine("{0:R} = {1:R}: {2}",
                        initialValue, roundTripped, initialValue.Equals(roundTripped))
      Console.WriteLine()

      Console.WriteLine("Attempting to round-trip a Double with 'G17':")
      Dim valueString17 As String = initialValue.ToString("G17",
                                                 CultureInfo.InvariantCulture)
      Dim roundTripped17 As Double = double.Parse(valueString17,
                                            CultureInfo.InvariantCulture)
      Console.WriteLine("{0:R} = {1:R}: {2}",
                        initialValue, roundTripped17, initialValue.Equals(roundTripped17))
   End Sub
End Module
' If compiled to an application that targets anycpu or x64 and run on an x64 system,
' the example displays the following output:
'       Attempting to round-trip a Double with 'R':
'       0.6822871999174 = 0.68228719991740006: False
'
'       Attempting to round-trip a Double with 'G17':
'       0.6822871999174 = 0.6822871999174: True
```

Back to table

The Hexadecimal ("X") Format Specifier

The hexadecimal ("X") format specifier converts a number to a string of hexadecimal digits. The case of the format specifier indicates whether to use uppercase or lowercase characters for hexadecimal digits that are greater than 9. For example, use "X" to produce "ABCDEF", and "x" to produce "abcdef". This format is supported only for integral types.

The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier.

The result string is not affected by the formatting information of the current NumberFormatInfo object.

The following example formats Int32 values with the hexadecimal format specifier.

```
int value;

value = 0x2045e;
Console::WriteLine(value.ToString("x"));
// Displays 2045e
Console::WriteLine(value.ToString("X"));
// Displays 2045E
Console::WriteLine(value.ToString("X8"));
// Displays 0002045E

value = 123456789;
Console::WriteLine(value.ToString("X"));
// Displays 75BCD15
Console::WriteLine(value.ToString("X2"));
// Displays 75BCD15
```

```
int value;

value = 0x2045e;
Console.WriteLine(value.ToString("x"));
// Displays 2045e
Console.WriteLine(value.ToString("X"));
// Displays 2045E
Console.WriteLine(value.ToString("X8"));
// Displays 0002045E

value = 123456789;
Console.WriteLine(value.ToString("X"));
// Displays 75BCD15
Console.WriteLine(value.ToString("X2"));
// Displays 75BCD15
```

```
Dim value As Integer

value = &h2045e
Console.WriteLine(value.ToString("x"))
' Displays 2045e
Console.WriteLine(value.ToString("X"))
' Displays 2045E
Console.WriteLine(value.ToString("X8"))
' Displays 0002045E

value = 123456789
Console.WriteLine(value.ToString("X"))
' Displays 75BCD15
Console.WriteLine(value.ToString("X2"))
' Displays 75BCD15
```

Back to table

Notes

Control Panel Settings

The settings in the **Regional and Language Options** item in Control Panel influence the result string produced by a formatting operation. Those settings are used to initialize the NumberFormatInfo object associated with the current thread culture, which provides values used to govern formatting. Computers that use different settings generate different result strings.

In addition, if the CultureInfo.CultureInfo(String) constructor is used to instantiate a new CultureInfo object that represents the same culture as the current system culture, any customizations established by the **Regional and Language Options** item in Control Panel will be applied to the new CultureInfo object. You can use the CultureInfo.CultureInfo(String, Boolean) constructor to create a CultureInfo object that does not reflect a system's customizations.

NumberFormatInfo Properties

Formatting is influenced by the properties of the current NumberFormatInfo object, which is provided implicitly by the current thread culture or explicitly by the IFormatProvider parameter of the method that invokes formatting. Specify a NumberFormatInfo or CultureInfo object for that parameter.

> **NOTE**
>
> For information about customizing the patterns or strings used in formatting numeric values, see the NumberFormatInfo class topic.

Integral and Floating-Point Numeric Types

Some descriptions of standard numeric format specifiers refer to integral or floating-point numeric types. The integral numeric types are Byte, SByte, Int16, Int32, Int64, UInt16, UInt32, UInt64, and BigInteger. The floating-point numeric types are Decimal, Single, and Double.

Floating-Point Infinities and NaN

Regardless of the format string, if the value of a Single or Double floating-point type is positive infinity, negative infinity, or not a number (NaN), the formatted string is the value of the respective PositiveInfinitySymbol, NegativeInfinitySymbol, or NaNSymbol property that is specified by the currently applicable NumberFormatInfo object.

Example

> **NOTE**
>
> The 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 following example formats an integral and a floating-point numeric value using the en-US culture and all the standard numeric format specifiers. This example uses two particular numeric types (Double and Int32), but would yield similar results for any of the other numeric base types (Byte, SByte, Int16, Int32, Int64, UInt16, UInt32, UInt64, BigInteger, Decimal, and Single).

```
using System;
using System.Globalization;
using System.Threading;

public class NumericFormats
{
   public static void Main()
   {
      // Display string representations of numbers for en-us culture
      CultureInfo ci = new CultureInfo("en-us");

      // Output floating point values
      double floating = 10761.937554;
      Console.WriteLine("C: {0}",
              floating.ToString("C", ci));           // Displays "C: $10,761.94"
      Console.WriteLine("E: {0}",
              floating.ToString("E03", ci));         // Displays "E: 1.076E+004"
      Console.WriteLine("F: {0}",
              floating.ToString("F04", ci));         // Displays "F: 10761.9376"
      Console.WriteLine("G: {0}",
              floating.ToString("G", ci));           // Displays "G: 10761.937554"
      Console.WriteLine("N: {0}",
              floating.ToString("N03", ci));         // Displays "N: 10,761.938"
      Console.WriteLine("P: {0}",
              (floating/10000).ToString("P02", ci)); // Displays "P: 107.62 %"
      Console.WriteLine("R: {0}",
              floating.ToString("R", ci));           // Displays "R: 10761.937554"
      Console.WriteLine();

      // Output integral values
      int integral = 8395;
      Console.WriteLine("C: {0}",
              integral.ToString("C", ci));           // Displays "C: $8,395.00"
      Console.WriteLine("D: {0}",
              integral.ToString("D6", ci));          // Displays "D: 008395"
      Console.WriteLine("E: {0}",
              integral.ToString("E03", ci));         // Displays "E: 8.395E+003"
      Console.WriteLine("F: {0}",
              integral.ToString("F01", ci));         // Displays "F: 8395.0"
      Console.WriteLine("G: {0}",
              integral.ToString("G", ci));           // Displays "G: 8395"
      Console.WriteLine("N: {0}",
              integral.ToString("N01", ci));         // Displays "N: 8,395.0"
      Console.WriteLine("P: {0}",
              (integral/10000.0).ToString("P02", ci)); // Displays "P: 83.95 %"
      Console.WriteLine("X: 0x{0}",
              integral.ToString("X", ci));           // Displays "X: 0x20CB"
      Console.WriteLine();
   }
}
```

```
Option Strict On

Imports System.Globalization
Imports System.Threading

Module NumericFormats
   Public Sub Main()
      ' Display string representations of numbers for en-us culture
      Dim ci As New CultureInfo("en-us")

      ' Output floating point values
      Dim floating As Double = 10761.937554
      Console.WriteLine("C: {0}", _
              floating.ToString("C", ci))           ' Displays "C: $10,761.94"
      Console.WriteLine("E: {0}", _
              floating.ToString("E03", ci))         ' Displays "E: 1.076E+004"
      Console.WriteLine("F: {0}", _
              floating.ToString("F04", ci))         ' Displays "F: 10761.9376"
      Console.WriteLine("G: {0}", _
              floating.ToString("G", ci))           ' Displays "G: 10761.937554"
      Console.WriteLine("N: {0}", _
              floating.ToString("N03", ci))         ' Displays "N: 10,761.938"
      Console.WriteLine("P: {0}", _
              (floating/10000).ToString("P02", ci)) ' Displays "P: 107.62 %"
      Console.WriteLine("R: {0}", _
              floating.ToString("R", ci))           ' Displays "R: 10761.937554"
      Console.WriteLine()

      ' Output integral values
      Dim integral As Integer = 8395
      Console.WriteLine("C: {0}", _
              integral.ToString("C", ci))           ' Displays "C: $8,395.00"
      Console.WriteLine("D: {0}", _
              integral.ToString("D6"))              ' Displays "D: 008395"
      Console.WriteLine("E: {0}", _
              integral.ToString("E03", ci))         ' Displays "E: 8.395E+003"
      Console.WriteLine("F: {0}", _
              integral.ToString("F01", ci))         ' Displays "F: 8395.0"
      Console.WriteLine("G: {0}", _
              integral.ToString("G", ci))           ' Displays "G: 8395"
      Console.WriteLine("N: {0}", _
              integral.ToString("N01", ci))         ' Displays "N: 8,395.0"
      Console.WriteLine("P: {0}", _
              (integral/10000).ToString("P02", ci)) ' Displays "P: 83.95 %"
      Console.WriteLine("X: 0x{0}", _
              integral.ToString("X", ci))           ' Displays "X: 0x20CB"
      Console.WriteLine()
   End Sub
End Module
```

See also

- NumberFormatInfo
- Custom Numeric Format Strings
- Formatting Types
- How to: Pad a Number with Leading Zeros
- Sample: .NET Framework 4 Formatting Utility
- Composite Formatting

Custom numeric format strings

9/6/2018 • 22 minutes to read • Edit Online

You can create a custom numeric format string, which consists of one or more custom numeric specifiers, to define how to format numeric data. A custom numeric format string is any format string that is not a standard numeric format string.

Custom numeric format strings are supported by some overloads of the `ToString` method of all numeric types. For example, you can supply a numeric format string to the ToString(String) and ToString(String, IFormatProvider) methods of the Int32 type. Custom numeric format strings are also supported by the .NET composite formatting feature, which is used by some `Write` and `WriteLine` methods of the Console and StreamWriter classes, the String.Format method, and the StringBuilder.AppendFormat method. String interpolation feature also supports custom numeric format strings.

> **TIP**
> You can download the Formatting Utility, an application that enables you to apply format strings to either numeric or date and time values and displays the result string.

The following table describes the custom numeric format specifiers and displays sample output produced by each format specifier. See the Notes section for additional information about using custom numeric format strings, and the Example section for a comprehensive illustration of their use.

FORMAT SPECIFIER	NAME	DESCRIPTION	EXAMPLES
"0"	Zero placeholder	Replaces the zero with the corresponding digit if one is present; otherwise, zero appears in the result string. More information: The "0" Custom Specifier.	1234.5678 ("00000") -> 01235 0.45678 ("0.00", en-US) -> 0.46 0.45678 ("0.00", fr-FR) -> 0,46
"#"	Digit placeholder	Replaces the "#" symbol with the corresponding digit if one is present; otherwise, no digit appears in the result string. Note that no digit appears in the result string if the corresponding digit in the input string is a non-significant 0. For example, 0003 ("####") -> 3. More information: The "#" Custom Specifier.	1234.5678 ("#####") -> 1235 0.45678 ("#.##", en-US) -> .46 0.45678 ("#.##", fr-FR) -> ,46

FORMAT SPECIFIER	NAME	DESCRIPTION	EXAMPLES
"."	Decimal point	Determines the location of the decimal separator in the result string. More information: The "." Custom Specifier.	0.45678 ("0.00", en-US) -> 0.46 0.45678 ("0.00", fr-FR) -> 0,46
","	Group separator and number scaling	Serves as both a group separator and a number scaling specifier. As a group separator, it inserts a localized group separator character between each group. As a number scaling specifier, it divides a number by 1000 for each comma specified. More information: The "," Custom Specifier.	Group separator specifier: 2147483647 ("##,#", en-US) -> 2,147,483,647 2147483647 ("##,#", es-ES) -> 2.147.483.647 Scaling specifier: 2147483647 ("#,#,,", en-US) -> 2,147 2147483647 ("#,#,,", es-ES) -> 2.147
"%"	Percentage placeholder	Multiplies a number by 100 and inserts a localized percentage symbol in the result string. More information: The "%" Custom Specifier.	0.3697 ("%#0.00", en-US) -> %36.97 0.3697 ("%#0.00", el-GR) -> %36,97 0.3697 ("##.0 %", en-US) -> 37.0 % 0.3697 ("##.0 %", el-GR) -> 37,0 %
"‰"	Per mille placeholder	Multiplies a number by 1000 and inserts a localized per mille symbol in the result string. More information: The "‰" Custom Specifier.	0.03697 ("#0.00‰", en-US) -> 36.97‰ 0.03697 ("#0.00‰", ru-RU) -> 36,97‰

FORMAT SPECIFIER	NAME	DESCRIPTION	EXAMPLES
"E0" "E+0" "E-0" "e0" "e+0" "e-0"	Exponential notation	If followed by at least one 0 (zero), formats the result using exponential notation. The case of "E" or "e" indicates the case of the exponent symbol in the result string. The number of zeros following the "E" or "e" character determines the minimum number of digits in the exponent. A plus sign (+) indicates that a sign character always precedes the exponent. A minus sign (-) indicates that a sign character precedes only negative exponents. More information: The "E" and "e" Custom Specifiers.	987654 ("#0.0e0") -> 98.8e4 1503.92311 ("0.0##e+00") -> 1.504e+03 1.8901385E-16 ("0.0e+00") -> 1.9e-16
"\\"	Escape character	Causes the next character to be interpreted as a literal rather than as a custom format specifier. More information: The "\\" Escape Character.	987654 ("\###00\#") -> #987654#
'*string*' "*string*"	Literal string delimiter	Indicates that the enclosed characters should be copied to the result string unchanged. More information: Character literals.	68 ("# 'degrees'") -> 68 degrees 68 ("#' degrees'") -> 68 degrees
;	Section separator	Defines sections with separate format strings for positive, negative, and zero numbers. More information: The ";" Section Separator.	12.345 ("#0.0#;(#0.0#);-\0-") -> 12.35 0 ("#0.0#;(#0.0#);-\0-") -> -0- -12.345 ("#0.0#;(#0.0#);-\0-") -> (12.35) 12.345 ("#0.0#;(#0.0#)") -> 12.35 0 ("#0.0#;(#0.0#)") -> 0.0 -12.345 ("#0.0#;(#0.0#)") -> (12.35)
Other	All other characters	The character is copied to the result string unchanged. More information: Character literals.	68 ("# °") -> 68 °

The following sections provide detailed information about each of the custom numeric format specifiers.

> **NOTE**
>
> The 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 "0" custom specifier

The "0" custom format specifier serves as a zero-placeholder symbol. If the value that is being formatted has a digit in the position where the zero appears in the format string, that digit is copied to the result string; otherwise, a zero appears in the result string. The position of the leftmost zero before the decimal point and the rightmost zero after the decimal point determines the range of digits that are always present in the result string.

The "00" specifier causes the value to be rounded to the nearest digit preceding the decimal, where rounding away from zero is always used. For example, formatting 34.5 with "00" would result in the value 35.

The following example displays several values that are formatted by using custom format strings that include zero placeholders.

```
double value;

value = 123;
Console::WriteLine(value.ToString("00000"));
Console::WriteLine(String::Format("{0:00000}", value));
// Displays 00123

value = 1.2;
Console::WriteLine(value.ToString("0.00", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                  "{0:0.00}", value));
// Displays 1.20

Console::WriteLine(value.ToString("00.00", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                "{0:00.00}", value));
// Displays 01.20

CultureInfo^ daDK = CultureInfo::CreateSpecificCulture("da-DK");
Console::WriteLine(value.ToString("00.00", daDK));
Console::WriteLine(String::Format(daDK, "{0:00.00}", value));
// Displays 01,20

value = .56;
Console::WriteLine(value.ToString("0.0", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                "{0:0.0}", value));
// Displays 0.6

value = 1234567890;
Console::WriteLine(value.ToString("0,0", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                "{0:0,0}", value));
// Displays 1,234,567,890

CultureInfo^ elGR = CultureInfo::CreateSpecificCulture("el-GR");
Console::WriteLine(value.ToString("0,0", elGR));
Console::WriteLine(String::Format(elGR, "{0:0,0}", value));
// Displays 1.234.567.890

value = 1234567890.123456;
Console::WriteLine(value.ToString("0,0.0", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                "{0:0,0.0}", value));
// Displays 1,234,567,890.1

value = 1234.567890;
Console::WriteLine(value.ToString("0,0.00", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                "{0:0,0.00}", value));
// Displays 1,234.57
```

```
double value;

value = 123;
Console.WriteLine(value.ToString("00000"));
Console.WriteLine(String.Format("{0:00000}", value));
// Displays 00123

value = 1.2;
Console.WriteLine(value.ToString("0.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                  "{0:0.00}", value));
// Displays 1.20

Console.WriteLine(value.ToString("00.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:00.00}", value));
// Displays 01.20

CultureInfo daDK = CultureInfo.CreateSpecificCulture("da-DK");
Console.WriteLine(value.ToString("00.00", daDK));
Console.WriteLine(String.Format(daDK, "{0:00.00}", value));
// Displays 01,20

value = .56;
Console.WriteLine(value.ToString("0.0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.0}", value));
// Displays 0.6

value = 1234567890;
Console.WriteLine(value.ToString("0,0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0,0}", value));
// Displays 1,234,567,890

CultureInfo elGR = CultureInfo.CreateSpecificCulture("el-GR");
Console.WriteLine(value.ToString("0,0", elGR));
Console.WriteLine(String.Format(elGR, "{0:0,0}", value));
// Displays 1.234.567.890

value = 1234567890.123456;
Console.WriteLine(value.ToString("0,0.0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0,0.0}", value));
// Displays 1,234,567,890.1

value = 1234.567890;
Console.WriteLine(value.ToString("0,0.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0,0.00}", value));
// Displays 1,234.57
```

```
Dim value As Double

value = 123
Console.WriteLine(value.ToString("00000"))
Console.WriteLine(String.Format("{0:00000}", value))
' Displays 00123

value = 1.2
Console.Writeline(value.ToString("0.00", CultureInfo.InvariantCulture))
Console.Writeline(String.Format(CultureInfo.InvariantCulture,
                  "{0:0.00}", value))
' Displays 1.20
Console.WriteLine(value.ToString("00.00", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:00.00}", value))
' Displays 01.20
Dim daDK As CultureInfo = CultureInfo.CreateSpecificCulture("da-DK")
Console.WriteLine(value.ToString("00.00", daDK))
Console.WriteLine(String.Format(daDK, "{0:00.00}", value))
' Displays 01,20

value = .56
Console.WriteLine(value.ToString("0.0", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.0}", value))
' Displays 0.6

value = 1234567890
Console.WriteLine(value.ToString("0,0", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0,0}", value))
' Displays 1,234,567,890
Dim elGR As CultureInfo = CultureInfo.CreateSpecificCulture("el-GR")
Console.WriteLine(value.ToString("0,0", elGR))
Console.WriteLine(String.Format(elGR, "{0:0,0}", value))
' Displays 1.234.567.890

value = 1234567890.123456
Console.WriteLine(value.ToString("0,0.0", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0,0.0}", value))
' Displays 1,234,567,890.1

value = 1234.567890
Console.WriteLine(value.ToString("0,0.00", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0,0.00}", value))
' Displays 1,234.57
```

Back to table

The "#" custom specifier

The "#" custom format specifier serves as a digit-placeholder symbol. If the value that is being formatted has a digit in the position where the "#" symbol appears in the format string, that digit is copied to the result string. Otherwise, nothing is stored in that position in the result string.

Note that this specifier never displays a zero that is not a significant digit, even if zero is the only digit in the string. It will display zero only if it is a significant digit in the number that is being displayed.

The "##" format string causes the value to be rounded to the nearest digit preceding the decimal, where rounding away from zero is always used. For example, formatting 34.5 with "##" would result in the value 35.

The following example displays several values that are formatted by using custom format strings that include digit placeholders.

```
double value;

value = 1.2;
Console::WriteLine(value.ToString("#.##", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:#.##}", value));
// Displays 1.2

value = 123;
Console::WriteLine(value.ToString("#####"));
Console::WriteLine(String::Format("{0:#####}", value));
// Displays 123

value = 123456;
Console::WriteLine(value.ToString("[##-##-##]"));
Console::WriteLine(String::Format("{0:[##-##-##]}", value));
// Displays [12-34-56]

value = 1234567890;
Console::WriteLine(value.ToString("#"));
Console::WriteLine(String::Format("{0:#}", value));
// Displays 1234567890

Console::WriteLine(value.ToString("(###) ###-####"));
Console::WriteLine(String::Format("{0:(###) ###-####}", value));
// Displays (123) 456-7890
```

```
double value;

value = 1.2;
Console.WriteLine(value.ToString("#.##", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#.##}", value));
// Displays 1.2

value = 123;
Console.WriteLine(value.ToString("#####"));
Console.WriteLine(String.Format("{0:#####}", value));
// Displays 123

value = 123456;
Console.WriteLine(value.ToString("[##-##-##]"));
Console.WriteLine(String.Format("{0:[##-##-##]}", value));
// Displays [12-34-56]

value = 1234567890;
Console.WriteLine(value.ToString("#"));
Console.WriteLine(String.Format("{0:#}", value));
// Displays 1234567890

Console.WriteLine(value.ToString("(###) ###-####"));
Console.WriteLine(String.Format("{0:(###) ###-####}", value));
// Displays (123) 456-7890
```

```
  Dim value As Double

  value = 1.2
  Console.WriteLine(value.ToString("#.##", CultureInfo.InvariantCulture))
  Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                  "{0:#.##}", value))
  ' Displays 1.2

  value = 123
  Console.WriteLine(value.ToString("#####"))
  Console.WriteLine(String.Format("{0:#####}", value))
  ' Displays 123

  value = 123456
  Console.WriteLine(value.ToString("[##-##-##]"))
  Console.WriteLine(String.Format("{0:[##-##-##]}", value))
' Displays [12-34-56]

  value = 1234567890
  Console.WriteLine(value.ToString("#"))
  Console.WriteLine(String.Format("{0:#}", value))
  ' Displays 1234567890

  Console.WriteLine(value.ToString("(###) ###-####"))
  Console.WriteLine(String.Format("{0:(###) ###-####}", value))
  ' Displays (123) 456-7890
```

To return a result string in which absent digits or leading zeroes are replaced by spaces, use the composite formatting feature and specify a field width, as the following example illustrates.

```
using namespace System;

void main()
{
    Double value = .324;
    Console::WriteLine("The value is: '{0,5:#.###}'", value);
}
// The example displays the following output if the current culture
// is en-US:
//      The value is: ' .324'
```

```
using System;

public class Example
{
   public static void Main()
   {
      Double value = .324;
      Console.WriteLine("The value is: '{0,5:#.###}'", value);
   }
}
// The example displays the following output if the current culture
// is en-US:
//      The value is: ' .324'
```

```
Module Example
   Public Sub Main()
      Dim value As Double = .324
      Console.WriteLine("The value is: '{0,5:#.###}'", value)
   End Sub
End Module
' The example displays the following output if the current culture
' is en-US:
'      The value is: ' .324'
```

Back to table

The "." custom specifier

The "." custom format specifier inserts a localized decimal separator into the result string. The first period in the format string determines the location of the decimal separator in the formatted value; any additional periods are ignored.

The character that is used as the decimal separator in the result string is not always a period; it is determined by the NumberDecimalSeparator property of the NumberFormatInfo object that controls formatting.

The following example uses the "." format specifier to define the location of the decimal point in several result strings.

```
 double value;

 value = 1.2;
 Console::WriteLine(value.ToString("0.00", CultureInfo::InvariantCulture));
 Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                 "{0:0.00}", value));
 // Displays 1.20

 Console::WriteLine(value.ToString("00.00", CultureInfo::InvariantCulture));
 Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                 "{0:00.00}", value));
 // Displays 01.20

 Console::WriteLine(value.ToString("00.00",
                   CultureInfo::CreateSpecificCulture("da-DK")));
 Console::WriteLine(String::Format(CultureInfo::CreateSpecificCulture("da-DK"),
                   "{0:00.00}", value));
 // Displays 01,20

 value = .086;
 Console::WriteLine(value.ToString("#0.##%", CultureInfo::InvariantCulture));
 Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                 "{0:#0.##%}", value));
 // Displays 8.6%

 value = 86000;
 Console::WriteLine(value.ToString("0.###E+0", CultureInfo::InvariantCulture));
 Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                   "{0:0.###E+0}", value));
// Displays 8.6E+4
```

```csharp
double value;

value = 1.2;
Console.WriteLine(value.ToString("0.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.00}", value));
// Displays 1.20

Console.WriteLine(value.ToString("00.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:00.00}", value));
// Displays 01.20

Console.WriteLine(value.ToString("00.00",
                  CultureInfo.CreateSpecificCulture("da-DK")));
Console.WriteLine(String.Format(CultureInfo.CreateSpecificCulture("da-DK"),
                  "{0:00.00}", value));
// Displays 01,20

value = .086;
Console.WriteLine(value.ToString("#0.##%", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#0.##%}", value));
// Displays 8.6%

value = 86000;
Console.WriteLine(value.ToString("0.###E+0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                  "{0:0.###E+0}", value));
// Displays 8.6E+4
```

```vb
Dim value As Double

value = 1.2
Console.Writeline(value.ToString("0.00", CultureInfo.InvariantCulture))
Console.Writeline(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.00}", value))
' Displays 1.20

Console.WriteLine(value.ToString("00.00", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:00.00}", value))
' Displays 01.20

Console.WriteLine(value.ToString("00.00", _
                  CultureInfo.CreateSpecificCulture("da-DK")))
Console.WriteLine(String.Format(CultureInfo.CreateSpecificCulture("da-DK"),
                  "{0:00.00}", value))
' Displays 01,20

value = .086
Console.WriteLine(value.ToString("#0.##%", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#0.##%}", value))
' Displays 8.6%

value = 86000
Console.WriteLine(value.ToString("0.###E+0", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                  "{0:0.###E+0}", value))
' Displays 8.6E+4
```

Back to table

The "," custom specifier

The "," character serves as both a group separator and a number scaling specifier.

- Group separator: If one or more commas are specified between two digit placeholders (0 or #) that format the integral digits of a number, a group separator character is inserted between each number group in the integral part of the output.

  The NumberGroupSeparator and NumberGroupSizes properties of the current NumberFormatInfo object determine the character used as the number group separator and the size of each number group. For example, if the string "#,#" and the invariant culture are used to format the number 1000, the output is "1,000".

- Number scaling specifier: If one or more commas are specified immediately to the left of the explicit or implicit decimal point, the number to be formatted is divided by 1000 for each comma. For example, if the string "0,," is used to format the number 100 million, the output is "100".

You can use group separator and number scaling specifiers in the same format string. For example, if the string "#,0,," and the invariant culture are used to format the number one billion, the output is "1,000".

The following example illustrates the use of the comma as a group separator.

```
double value = 1234567890;
Console::WriteLine(value.ToString("#,#", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                "{0:#,#}", value));
// Displays 1,234,567,890

Console::WriteLine(value.ToString("#,##0,,", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                "{0:#,##0,,}", value));
// Displays 1,235
```

```
double value = 1234567890;
Console.WriteLine(value.ToString("#,#", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,#}", value));
// Displays 1,234,567,890

Console.WriteLine(value.ToString("#,##0,,", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,##0,,}", value));
// Displays 1,235
```

```
Dim value As Double = 1234567890
Console.WriteLine(value.ToString("#,#", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,#}", value))
' Displays 1,234,567,890

Console.WriteLine(value.ToString("#,##0,,", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,##0,,}", value))
' Displays 1,235
```

The following example illustrates the use of the comma as a specifier for number scaling.

```cpp
    double value = 1234567890;
    Console::WriteLine(value.ToString("#,,", CultureInfo::InvariantCulture));
    Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                    "{0:#,,}", value));
    // Displays 1235

    Console::WriteLine(value.ToString("#,,,", CultureInfo::InvariantCulture));
    Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                    "{0:#,,,}", value));
  // Displays 1

    Console::WriteLine(value.ToString("#,##0,,", CultureInfo::InvariantCulture));
    Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                    "{0:#,##0,,}", value));
  // Displays 1,235
```

```csharp
     double value = 1234567890;
     Console.WriteLine(value.ToString("#,,", CultureInfo.InvariantCulture));
     Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                     "{0:#,,}", value));
     // Displays 1235

     Console.WriteLine(value.ToString("#,,,", CultureInfo.InvariantCulture));
     Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                     "{0:#,,,}", value));
  // Displays 1

     Console.WriteLine(value.ToString("#,##0,,", CultureInfo.InvariantCulture));
     Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                     "{0:#,##0,,}", value));
  // Displays 1,235
```

```vb
    Dim value As Double = 1234567890
    Console.WriteLine(value.ToString("#,,", CultureInfo.InvariantCulture))
    Console.WriteLine(String.Format(CultureInfo.InvariantCulture, "{0:#,,}", value))
    ' Displays 1235

    Console.WriteLine(value.ToString("#,,,", CultureInfo.InvariantCulture))
    Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                    "{0:#,,,}", value))
  ' Displays 1

    Console.WriteLine(value.ToString("#,##0,,", CultureInfo.InvariantCulture))
    Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                    "{0:#,##0,,}", value))
  ' Displays 1,235
```

Back to table

The "%" custom specifier

A percent sign (%) in a format string causes a number to be multiplied by 100 before it is formatted. The localized percent symbol is inserted in the number at the location where the % appears in the format string. The percent character used is defined by the PercentSymbol property of the current NumberFormatInfo object.

The following example defines several custom format strings that include the "%" custom specifier.

```
double value = .086;
Console::WriteLine(value.ToString("#0.##%", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                "{0:#0.##%}", value));
// Displays 8.6%
```

```
double value = .086;
Console.WriteLine(value.ToString("#0.##%", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#0.##%}", value));
// Displays 8.6%
```

```
Dim value As Double = .086
Console.WriteLine(value.ToString("#0.##%", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#0.##%}", value))
' Displays 8.6%
```

Back to table

The "‰" custom specifier

A per mille character (‰ or \u2030) in a format string causes a number to be multiplied by 1000 before it is formatted. The appropriate per mille symbol is inserted in the returned string at the location where the ‰ symbol appears in the format string. The per mille character used is defined by the NumberFormatInfo.PerMilleSymbol property of the object that provides culture-specific formatting information.

The following example defines a custom format string that includes the "‰" custom specifier.

```
double value = .00354;
String^ perMilleFmt = "#0.## " + '\u2030';
Console::WriteLine(value.ToString(perMilleFmt, CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                "{0:" + perMilleFmt + "}", value));
// Displays 3.54‰
```

```
double value = .00354;
string perMilleFmt = "#0.## " + '\u2030';
Console.WriteLine(value.ToString(perMilleFmt, CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:" + perMilleFmt + "}", value));
// Displays 3.54‰
```

```
Dim value As Double = .00354
Dim perMilleFmt As String = "#0.## " & ChrW(&h2030)
Console.WriteLine(value.ToString(perMilleFmt, CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:" + perMilleFmt + "}", value))
' Displays 3.54 ‰
```

Back to table

The "E" and "e" custom specifiers

If any of the strings "E", "E+", "E-", "e", "e+", or "e-" are present in the format string and are followed immediately by at least one zero, the number is formatted by using scientific notation with an "E" or "e" inserted between the number and the exponent. The number of zeros following the scientific notation indicator determines the minimum number of digits to output for the exponent. The "E+" and "e+" formats indicate that a plus sign or minus sign should always precede the exponent. The "E", "E-", "e", or "e-" formats indicate that a sign character should precede only negative exponents.

The following example formats several numeric values using the specifiers for scientific notation.

```
double value = 86000;
Console::WriteLine(value.ToString("0.###E+0", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                "{0:0.###E+0}", value));
// Displays 8.6E+4

Console::WriteLine(value.ToString("0.###E+000", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                "{0:0.###E+000}", value));
// Displays 8.6E+004

Console::WriteLine(value.ToString("0.###E-000", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                "{0:0.###E-000}", value));
// Displays 8.6E004
```

```
double value = 86000;
Console.WriteLine(value.ToString("0.###E+0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E+0}", value));
// Displays 8.6E+4

Console.WriteLine(value.ToString("0.###E+000", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E+000}", value));
// Displays 8.6E+004

Console.WriteLine(value.ToString("0.###E-000", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E-000}", value));
// Displays 8.6E004
```

```
Dim value As Double = 86000
Console.WriteLine(value.ToString("0.###E+0", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E+0}", value))
' Displays 8.6E+4

Console.WriteLine(value.ToString("0.###E+000", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E+000}", value))
' Displays 8.6E+004

Console.WriteLine(value.ToString("0.###E-000", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E-000}", value))
' Displays 8.6E004
```

Back to table

The "\" escape character

The "#", "0", ".", ",", "%", and "‰" symbols in a format string are interpreted as format specifiers rather than as literal characters. Depending on their position in a custom format string, the uppercase and lowercase "E" as well as the + and - symbols may also be interpreted as format specifiers.

To prevent a character from being interpreted as a format specifier, you can precede it with a backslash, which is the escape character. The escape character signifies that the following character is a character literal that should be included in the result string unchanged.

To include a backslash in a result string, you must escape it with another backslash (`\\`).

> **NOTE**
>
> Some compilers, such as the C++ and C# compilers, may also interpret a single backslash character as an escape character. To ensure that a string is interpreted correctly when formatting, you can use the verbatim string literal character (the @ character) before the string in C#, or add another backslash character before each backslash in C# and C++. The following C# example illustrates both approaches.

The following example uses the escape character to prevent the formatting operation from interpreting the "#", "0", and "\" characters as either escape characters or format specifiers. The C# examples uses an additional backslash to ensure that a backslash is interpreted as a literal character.

```
int value = 123;
Console::WriteLine(value.ToString("\\#\\#\\# ##0 dollars and \\0\\0 cents \\#\\#\\#"));
Console::WriteLine(String::Format("{0:\\#\\#\\# ##0 dollars and \\0\\0 cents \\#\\#\\#}",
                                  value));
// Displays ### 123 dollars and 00 cents ###

Console::WriteLine(value.ToString("\\#\\#\\# ##0 dollars and \0\0 cents \\#\\#\\#"));
Console::WriteLine(String::Format("{0:\\#\\#\\# ##0 dollars and \0\0 cents \\#\\#\\#}",
                                value));
// Displays ### 123 dollars and 00 cents ###

Console::WriteLine(value.ToString("\\\\\\\\\\\\ ##0 dollars and \\0\\0 cents \\\\\\\\\\\\"));
Console::WriteLine(String::Format("{0:\\\\\\\\\\\\ ##0 dollars and \\0\\0 cents \\\\\\\\\\\\}",
                                value));
// Displays \\\ 123 dollars and 00 cents \\\

Console::WriteLine(value.ToString("\\\\\\ ##0 dollars and \0\0 cents \\\\\\"));
Console::WriteLine(String::Format("{0:\\\\\\ ##0 dollars and \0\0 cents \\\\\\}",
                                value));
// Displays \\\ 123 dollars and 00 cents \\\
```

```
int value = 123;
Console.WriteLine(value.ToString("\\#\\#\\# ##0 dollars and \\0\\0 cents \\#\\#\\#"));
Console.WriteLine(String.Format("{0:\\#\\#\\# ##0 dollars and \\0\\0 cents \\#\\#\\#}",
                                value));
// Displays ### 123 dollars and 00 cents ###

Console.WriteLine(value.ToString(@"\#\#\# ##0 dollars and \0\0 cents \#\#\#"));
Console.WriteLine(String.Format(@"{0:\#\#\# ##0 dollars and \0\0 cents \#\#\#}",
                                value));
// Displays ### 123 dollars and 00 cents ###

Console.WriteLine(value.ToString("\\\\\\\\\\\\ ##0 dollars and \\0\\0 cents \\\\\\\\\\\\"));
Console.WriteLine(String.Format("{0:\\\\\\\\\\\\ ##0 dollars and \\0\\0 cents \\\\\\\\\\\\}",
                                value));
// Displays \\\ 123 dollars and 00 cents \\\

Console.WriteLine(value.ToString(@"\\\\\\ ##0 dollars and \0\0 cents \\\\\\"));
Console.WriteLine(String.Format(@"{0:\\\\\\ ##0 dollars and \0\0 cents \\\\\\}",
                                value));
// Displays \\\ 123 dollars and 00 cents \\\
```

```
Dim value As Integer = 123
Console.WriteLine(value.ToString("\#\#\# ##0 dollars and \0\0 cents \#\#\#"))
Console.WriteLine(String.Format("{0:\#\#\# ##0 dollars and \0\0 cents \#\#\#}",
                                value))
' Displays ### 123 dollars and 00 cents ###

Console.WriteLine(value.ToString("\\\\\\ ##0 dollars and \0\0 cents \\\\\\"))
Console.WriteLine(String.Format("{0:\\\\\\ ##0 dollars and \0\0 cents \\\\\\}",
                                value))
' Displays \\\ 123 dollars and 00 cents \\\
```

Back to table

The ";" section separator

The semicolon (;) is a conditional format specifier that applies different formatting to a number depending on whether its value is positive, negative, or zero. To produce this behavior, a custom format string can contain up to three sections separated by semicolons. These sections are described in the following table.

NUMBER OF SECTIONS	DESCRIPTION
One section	The format string applies to all values.
Two sections	The first section applies to positive values and zeros, and the second section applies to negative values. If the number to be formatted is negative, but becomes zero after rounding according to the format in the second section, the resulting zero is formatted according to the first section.

NUMBER OF SECTIONS	DESCRIPTION
Three sections	The first section applies to positive values, the second section applies to negative values, and the third section applies to zeros. The second section can be left empty (by having nothing between the semicolons), in which case the first section applies to all nonzero values. If the number to be formatted is nonzero, but becomes zero after rounding according to the format in the first or second section, the resulting zero is formatted according to the third section.

Section separators ignore any preexisting formatting associated with a number when the final value is formatted. For example, negative values are always displayed without a minus sign when section separators are used. If you want the final formatted value to have a minus sign, you should explicitly include the minus sign as part of the custom format specifier.

The following example uses the ";" format specifier to format positive, negative, and zero numbers differently.

```
double posValue = 1234;
double negValue = -1234;
double zeroValue = 0;

String^ fmt2 = "##;(##)";
String^ fmt3 = "##;(##);**Zero**";

Console::WriteLine(posValue.ToString(fmt2));
Console::WriteLine(String::Format("{0:" + fmt2 + "}", posValue));
// Displays 1234

Console::WriteLine(negValue.ToString(fmt2));
Console::WriteLine(String::Format("{0:" + fmt2 + "}", negValue));
// Displays (1234)

Console::WriteLine(zeroValue.ToString(fmt3));
Console::WriteLine(String::Format("{0:" + fmt3 + "}", zeroValue));
// Displays **Zero**
```

```
double posValue = 1234;
double negValue = -1234;
double zeroValue = 0;

string fmt2 = "##;(##)";
string fmt3 = "##;(##);**Zero**";

Console.WriteLine(posValue.ToString(fmt2));
Console.WriteLine(String.Format("{0:" + fmt2 + "}", posValue));
// Displays 1234

Console.WriteLine(negValue.ToString(fmt2));
Console.WriteLine(String.Format("{0:" + fmt2 + "}", negValue));
// Displays (1234)

Console.WriteLine(zeroValue.ToString(fmt3));
Console.WriteLine(String.Format("{0:" + fmt3 + "}", zeroValue));
// Displays **Zero**
```

```
Dim posValue As Double = 1234
Dim negValue As Double = -1234
Dim zeroValue As Double = 0

Dim fmt2 As String = "##;(##)"
Dim fmt3 As String = "##;(##);**Zero**"

Console.WriteLine(posValue.ToString(fmt2))
Console.WriteLine(String.Format("{0:" + fmt2 + "}", posValue))
' Displays 1234

Console.WriteLine(negValue.ToString(fmt2))
Console.WriteLine(String.Format("{0:" + fmt2 + "}", negValue))
' Displays (1234)

Console.WriteLine(zeroValue.ToString(fmt3))
Console.WriteLine(String.Format("{0:" + fmt3 + "}", zeroValue))
' Displays **Zero**
```

Back to table

Character literals

Format specifiers that appear in a custom numeric format string are always interpreted as formatting characters and never as literal characters. This includes the following characters:

- 0
- #
- %
- ‰
- '
- \
- .
- ,
- E or e, depending on its position in the format string.

All other characters are always interpreted as character literals and, in a formatting operation, are included in the result string unchanged. In a parsing operation, they must match the characters in the input string exactly; the comparison is case-sensitive.

The following example illustrates one common use of literal character units (in this case, thousands):

```
double n = 123.8;
Console.WriteLine($"{n:#,##0.0K}");
// The example displays the following output:
//      123.8K
```

```
Dim n As Double = 123.8
Console.WriteLine($"{n:#,##0.0K}")
' The example displays the following output:
'       123.8K
```

There are two ways to indicate that characters are to be interpreted as literal characters and not as formatting characters, so that they can be included in a result string or successfully parsed in an input string:

- By escaping a formatting character. For more information, see The "\" escape character.

- By enclosing the entire literal string in quotation apostrophes.

The following example uses both approaches to include reserved characters in a custom numeric format string.

```
double n = 9.3;
Console.WriteLine($@"{n:##.0\%}");
Console.WriteLine($@"{n:\'##\'}");
Console.WriteLine($@"{n:\\##\\}");
Console.WriteLine();
Console.WriteLine($"{n:##.0'%'}");
Console.WriteLine($@"{n:'\'##'\'}");
// The example displays the following output:
//      9.3%
//      '9'
//      \9\
//
//      9.3%
//      \9\
```

```
Dim n As Double = 9.3
Console.WriteLine($"{n:##.0\%}")
Console.WriteLine($"{n:\'##\'}")
Console.WriteLine($"{n:\\##\\}")
Console.WriteLine()
Console.WriteLine($"{n:##.0'%'}")
Console.WriteLine($"{n:'\'##'\'}")
' The example displays the following output:
'      9.3%
'      '9'
'      \9\
'
'      9.3%
'      \9\
```

Notes

Floating-Point infinities and NaN

Regardless of the format string, if the value of a Single or Double floating-point type is positive infinity, negative infinity, or not a number (NaN), the formatted string is the value of the respective PositiveInfinitySymbol, NegativeInfinitySymbol, or NaNSymbol property specified by the currently applicable NumberFormatInfo object.

Control Panel settings

The settings in the **Regional and Language Options** item in Control Panel influence the result string produced by a formatting operation. Those settings are used to initialize the NumberFormatInfo object associated with the current thread culture, and the current thread culture provides values used to govern formatting. Computers that use different settings generate different result strings.

In addition, if you use the CultureInfo.CultureInfo(String) constructor to instantiate a new CultureInfo object that represents the same culture as the current system culture, any customizations established by the **Regional and Language Options** item in Control Panel will be applied to the new CultureInfo object. You can use the CultureInfo.CultureInfo(String, Boolean) constructor to create a CultureInfo object that does not reflect a system's customizations.

Rounding and fixed-point format strings

For fixed-point format strings (that is, format strings that do not contain scientific notation format characters), numbers are rounded to as many decimal places as there are digit placeholders to the right of the decimal point. If the format string does not contain a decimal point, the number is rounded to the nearest integer. If the number has more digits than there are digit placeholders to the left of the decimal point, the extra digits are copied to the result string immediately before the first digit placeholder.

Back to table

Example

The following example demonstrates two custom numeric format strings. In both cases, the digit placeholder (`#`) displays the numeric data, and all other characters are copied to the result string.

```
double number1 = 1234567890;
String^ value1 = number1.ToString("(###) ###-####");
Console::WriteLine(value1);

int number2 = 42;
String^ value2 = number2.ToString("My Number = #");
Console::WriteLine(value2);
// The example displays the following output:
//       (123) 456-7890
//       My Number = 42
```

```
double number1 = 1234567890;
string value1 = number1.ToString("(###) ###-####");
Console.WriteLine(value1);

int number2 = 42;
string value2 = number2.ToString("My Number = #");
Console.WriteLine(value2);
// The example displays the following output:
//       (123) 456-7890
//       My Number = 42
```

```
Dim number1 As Double = 1234567890
Dim value1 As String = number1.ToString("(###) ###-####")
Console.WriteLine(value1)

Dim number2 As Integer = 42
Dim value2 As String = number2.ToString("My Number = #")
Console.WriteLine(value2)
' The example displays the following output:
'       (123) 456-7890
'       My Number = 42
```

Back to table

See also

- System.Globalization.NumberFormatInfo
- Formatting Types
- Standard Numeric Format Strings
- How to: Pad a Number with Leading Zeros
- Sample: .NET Framework 4 Formatting Utility

Standard Date and Time Format Strings

9/6/2018 • 28 minutes to read • Edit Online

A standard date and time format string uses a single format specifier to define the text representation of a date and time value. Any date and time format string that contains more than one character, including white space, is interpreted as a custom date and time format string; for more information, see Custom Date and Time Format Strings. A standard or custom format string can be used in two ways:

- To define the string that results from a formatting operation.
- To define the text representation of a date and time value that can be converted to a DateTime or DateTimeOffset value by a parsing operation.

> **TIP**
>
> You can download the Formatting Utility, an application that enables you to apply format strings to either numeric or date and time values and displays the result string.

Standard date and time format strings can be used with both DateTime and DateTimeOffset values.

> **NOTE**
>
> The 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.

The following table describes the standard date and time format specifiers. Unless otherwise noted, a particular standard date and time format specifier produces an identical string representation regardless of whether it is used with a DateTime or a DateTimeOffset value. See the Notes section for additional information about using standard date and time format strings.

FORMAT SPECIFIER	DESCRIPTION	EXAMPLES
"d"	Short date pattern. More information:The Short Date ("d") Format Specifier.	2009-06-15T13:45:30 -> 6/15/2009 (en-US) 2009-06-15T13:45:30 -> 15/06/2009 (fr-FR) 2009-06-15T13:45:30 -> 2009/06/15 (ja-JP)

FORMAT SPECIFIER	DESCRIPTION	EXAMPLES
"D"	Long date pattern. More information:The Long Date ("D") Format Specifier.	2009-06-15T13:45:30 -> Monday, June 15, 2009 (en-US) 2009-06-15T13:45:30 -> 15 июня 2009 г. (ru-RU) 2009-06-15T13:45:30 -> Montag, 15. Juni 2009 (de-DE)
"f"	Full date/time pattern (short time). More information: The Full Date Short Time ("f") Format Specifier.	2009-06-15T13:45:30 -> Monday, June 15, 2009 1:45 PM (en-US) 2009-06-15T13:45:30 -> den 15 juni 2009 13:45 (sv-SE) 2009-06-15T13:45:30 -> Δευτέρα, 15 Ιουνίου 2009 1:45 μμ (el-GR)
"F"	Full date/time pattern (long time). More information: The Full Date Long Time ("F") Format Specifier.	2009-06-15T13:45:30 -> Monday, June 15, 2009 1:45:30 PM (en-US) 2009-06-15T13:45:30 -> den 15 juni 2009 13:45:30 (sv-SE) 2009-06-15T13:45:30 -> Δευτέρα, 15 Ιουνίου 2009 1:45:30 μμ (el-GR)
"g"	General date/time pattern (short time). More information: The General Date Short Time ("g") Format Specifier.	2009-06-15T13:45:30 -> 6/15/2009 1:45 PM (en-US) 2009-06-15T13:45:30 -> 15/06/2009 13:45 (es-ES) 2009-06-15T13:45:30 -> 2009/6/15 13:45 (zh-CN)
"G"	General date/time pattern (long time). More information: The General Date Long Time ("G") Format Specifier.	2009-06-15T13:45:30 -> 6/15/2009 1:45:30 PM (en-US) 2009-06-15T13:45:30 -> 15/06/2009 13:45:30 (es-ES) 2009-06-15T13:45:30 -> 2009/6/15 13:45:30 (zh-CN)
"M", "m"	Month/day pattern. More information: The Month ("M", "m") Format Specifier.	2009-06-15T13:45:30 -> June 15 (en-US) 2009-06-15T13:45:30 -> 15. juni (da-DK) 2009-06-15T13:45:30 -> 15 Juni (id-ID)

FORMAT SPECIFIER	DESCRIPTION	EXAMPLES
"O", "o"	Round-trip date/time pattern. More information: The Round-trip ("O", "o") Format Specifier.	DateTime values: 2009-06-15T13:45:30 (DateTimeKind.Local) --> 2009-06-15T13:45:30.0000000-07:00 2009-06-15T13:45:30 (DateTimeKind.Utc) --> 2009-06-15T13:45:30.0000000Z 2009-06-15T13:45:30 (DateTimeKind.Unspecified) --> 2009-06-15T13:45:30.0000000 DateTimeOffset values: 2009-06-15T13:45:30-07:00 --> 2009-06-15T13:45:30.0000000-07:00
"R", "r"	RFC1123 pattern. More information: The RFC1123 ("R", "r") Format Specifier.	2009-06-15T13:45:30 -> Mon, 15 Jun 2009 20:45:30 GMT
"s"	Sortable date/time pattern. More information: The Sortable ("s") Format Specifier.	2009-06-15T13:45:30 (DateTimeKind.Local) -> 2009-06-15T13:45:30 2009-06-15T13:45:30 (DateTimeKind.Utc) -> 2009-06-15T13:45:30
"t"	Short time pattern. More information: The Short Time ("t") Format Specifier.	2009-06-15T13:45:30 -> 1:45 PM (en-US) 2009-06-15T13:45:30 -> 13:45 (hr-HR) 2009-06-15T13:45:30 -> 01:45 م (ar-EG)
"T"	Long time pattern. More information: The Long Time ("T") Format Specifier.	2009-06-15T13:45:30 -> 1:45:30 PM (en-US) 2009-06-15T13:45:30 -> 13:45:30 (hr-HR) 2009-06-15T13:45:30 -> 01:45:30 م (ar-EG)
"u"	Universal sortable date/time pattern. More information: The Universal Sortable ("u") Format Specifier.	With a DateTime value: 2009-06-15T13:45:30 -> 2009-06-15 13:45:30Z With a DateTimeOffset value: 2009-06-15T13:45:30 -> 2009-06-15 20:45:30Z

FORMAT SPECIFIER	DESCRIPTION	EXAMPLES
"U"	Universal full date/time pattern. More information: The Universal Full ("U") Format Specifier.	2009-06-15T13:45:30 -> Monday, June 15, 2009 8:45:30 PM (en-US) 2009-06-15T13:45:30 -> den 15 juni 2009 20:45:30 (sv-SE) 2009-06-15T13:45:30 -> Δευτέρα, 15 Ιουνίου 2009 8:45:30 μμ (el-GR)
"Y", "y"	Year month pattern. More information: The Year Month ("Y") Format Specifier.	2009-06-15T13:45:30 -> June, 2009 (en-US) 2009-06-15T13:45:30 -> juni 2009 (da-DK) 2009-06-15T13:45:30 -> Juni 2009 (id-ID)
Any other single character	Unknown specifier.	Throws a run-time FormatException.

How Standard Format Strings Work

In a formatting operation, a standard format string is simply an alias for a custom format string. The advantage of using an alias to refer to a custom format string is that, although the alias remains invariant, the custom format string itself can vary. This is important because the string representations of date and time values typically vary by culture. For example, the "d" standard format string indicates that a date and time value is to be displayed using a short date pattern. For the invariant culture, this pattern is "MM/dd/yyyy". For the fr-FR culture, it is "dd/MM/yyyy". For the ja-JP culture, it is "yyyy/MM/dd".

If a standard format string in a formatting operation maps to a particular culture's custom format string, your application can define the specific culture whose custom format strings are used in one of these ways:

- You can use the default (or current) culture. The following example displays a date using the current culture's short date format. In this case, the current culture is en-US.

```
// Display using current (en-us) culture's short date format
DateTime thisDate = new DateTime(2008, 3, 15);
Console.WriteLine(thisDate.ToString("d"));           // Displays 3/15/2008
```

```
' Display using current (en-us) culture's short date format
Dim thisDate As Date = #03/15/2008#
Console.WriteLine(thisDate.ToString("d"))     ' Displays 3/15/2008
```

- You can pass a CultureInfo object representing the culture whose formatting is to be used to a method that has an IFormatProvider parameter. The following example displays a date using the short date format of the pt-BR culture.

```
// Display using pt-BR culture's short date format
DateTime thisDate = new DateTime(2008, 3, 15);
CultureInfo culture = new CultureInfo("pt-BR");
Console.WriteLine(thisDate.ToString("d", culture));  // Displays 15/3/2008
```

```
' Display using pt-BR culture's short date format
Dim thisDate As Date = #03/15/2008#
Dim culture As New CultureInfo("pt-BR")
Console.WriteLine(thisDate.ToString("d", culture))   ' Displays 15/3/2008
```

- You can pass a DateTimeFormatInfo object that provides formatting information to a method that has an IFormatProvider parameter. The following example displays a date using the short date format from a DateTimeFormatInfo object for the hr-HR culture.

```
// Display using date format information from hr-HR culture
DateTime thisDate = new DateTime(2008, 3, 15);
DateTimeFormatInfo fmt = (new CultureInfo("hr-HR")).DateTimeFormat;
Console.WriteLine(thisDate.ToString("d", fmt));      // Displays 15.3.2008
```

```
' Display using date format information from hr-HR culture
Dim thisDate As Date = #03/15/2008#
Dim fmt As DateTimeFormatInfo = (New CultureInfo("hr-HR")).DateTimeFormat
Console.WriteLine(thisDate.ToString("d", fmt))   ' Displays 15.3.2008
```

NOTE

For information about customizing the patterns or strings used in formatting date and time values, see the NumberFormatInfo class topic.

In some cases, the standard format string serves as a convenient abbreviation for a longer custom format string that is invariant. Four standard format strings fall into this category: "O" (or "o"), "R" (or "r"), "s", and "u". These strings correspond to custom format strings defined by the invariant culture. They produce string representations of date and time values that are intended to be identical across cultures. The following table provides information on these four standard date and time format strings.

STANDARD FORMAT STRING	DEFINED BY DATETIMEFORMATINFO.INVARIANTINFO PROPERTY	CUSTOM FORMAT STRING
"O" or "o"	None	yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffzz
"R" or "r"	RFC1123Pattern	ddd, dd MMM yyyy HH':'mm':'ss 'GMT'
"s"	SortableDateTimePattern	yyyy'-'MM'-'dd'T'HH':'mm':'ss
"u"	UniversalSortableDateTimePattern	yyyy'-'MM'-'dd HH':'mm':'ss'Z'

Standard format strings can also be used in parsing operations with the DateTime.ParseExact or DateTimeOffset.ParseExact methods, which require an input string to exactly conform to a particular pattern for the parse operation to succeed. Many standard format strings map to multiple custom format strings, so a date and time value can be represented in a variety of formats and the parse operation will still succeed. You can determine the custom format string or strings that correspond to a standard format string by calling the DateTimeFormatInfo.GetAllDateTimePatterns(Char) method. The following example displays the custom format strings that map to the "d" (short date pattern) standard format string.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      Console.WriteLine("'d' standard format string:");
      foreach (var customString in DateTimeFormatInfo.CurrentInfo.GetAllDateTimePatterns('d'))
          Console.WriteLine("   {0}", customString);
   }
}
// The example displays the following output:
//       'd' standard format string:
//          M/d/yyyy
//          M/d/yy
//          MM/dd/yy
//          MM/dd/yyyy
//          yy/MM/dd
//          yyyy-MM-dd
//          dd-MMM-yy
```

```
Imports System.Globalization

Module Example
   Public Sub Main()
      Console.WriteLine("'d' standard format string:")
      For Each customString In DateTimeFormatInfo.CurrentInfo.GetAllDateTimePatterns("d"c)
          Console.WriteLine("   {0}", customString)
      Next
   End Sub
End Module
' The example displays the following output:
'     'd' standard format string:
'        M/d/yyyy
'        M/d/yy
'        MM/dd/yy
'        MM/dd/yyyy
'        yy/MM/dd
'        yyyy-MM-dd
'        dd-MMM-yy
```

The following sections describe the standard format specifiers for DateTime and DateTimeOffset values.

The Short Date ("d") Format Specifier

The "d" standard format specifier represents a custom date and time format string that is defined by a specific culture's DateTimeFormatInfo.ShortDatePattern property. For example, the custom format string that is returned by the ShortDatePattern property of the invariant culture is "MM/dd/yyyy".

The following table lists the DateTimeFormatInfo object properties that control the formatting of the returned string.

PROPERTY	DESCRIPTION
ShortDatePattern	Defines the overall format of the result string.
DateSeparator	Defines the string that separates the year, month, and day components of a date.

The following example uses the "d" format specifier to display a date and time value.

```
DateTime date1 = new DateTime(2008,4, 10);
Console.WriteLine(date1.ToString("d", DateTimeFormatInfo.InvariantInfo));
// Displays 04/10/2008
Console.WriteLine(date1.ToString("d",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays 4/10/2008
Console.WriteLine(date1.ToString("d",
                  CultureInfo.CreateSpecificCulture("en-NZ")));
// Displays 10/04/2008
Console.WriteLine(date1.ToString("d",
                  CultureInfo.CreateSpecificCulture("de-DE")));
// Displays 10.04.2008
```

```
Dim date1 As Date = #4/10/2008#
Console.WriteLine(date1.ToString("d", DateTimeFormatInfo.InvariantInfo))
' Displays 04/10/2008
Console.WriteLine(date1.ToString("d", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays 4/10/2008
Console.WriteLine(date1.ToString("d", _
                  CultureInfo.CreateSpecificCulture("en-NZ")))
' Displays 10/04/2008
Console.WriteLine(date1.ToString("d", _
                  CultureInfo.CreateSpecificCulture("de-DE")))
' Displays 10.04.2008
```

Back to table

The Long Date ("D") Format Specifier

The "D" standard format specifier represents a custom date and time format string that is defined by the current DateTimeFormatInfo.LongDatePattern property. For example, the custom format string for the invariant culture is "dddd, dd MMMM yyyy".

The following table lists the properties of the DateTimeFormatInfo object that control the formatting of the returned string.

PROPERTY	DESCRIPTION
LongDatePattern	Defines the overall format of the result string.
DayNames	Defines the localized day names that can appear in the result string.
MonthNames	Defines the localized month names that can appear in the result string.

The following example uses the "D" format specifier to display a date and time value.

```
DateTime date1 = new DateTime(2008, 4, 10);
Console.WriteLine(date1.ToString("D",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008
Console.WriteLine(date1.ToString("D",
                  CultureInfo.CreateSpecificCulture("pt-BR")));
// Displays quinta-feira, 10 de abril de 2008
Console.WriteLine(date1.ToString("D",
                  CultureInfo.CreateSpecificCulture("es-MX")));
// Displays jueves, 10 de abril de 2008
```

```
Dim date1 As Date = #4/10/2008#
Console.WriteLine(date1.ToString("D", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Thursday, April 10, 2008
Console.WriteLine(date1.ToString("D", _
                  CultureInfo.CreateSpecificCulture("pt-BR")))
' Displays quinta-feira, 10 de abril de 2008
Console.WriteLine(date1.ToString("D", _
                  CultureInfo.CreateSpecificCulture("es-MX")))
' Displays jueves, 10 de abril de 2008
```

Back to table

The Full Date Short Time ("f") Format Specifier

The "f" standard format specifier represents a combination of the long date ("D") and short time ("t") patterns, separated by a space.

The result string is affected by the formatting information of a specific DateTimeFormatInfo object. The following table lists the DateTimeFormatInfo object properties that may control the formatting of the returned string. The custom format specifier returned by the DateTimeFormatInfo.LongDatePattern and DateTimeFormatInfo.ShortTimePattern properties of some cultures may not make use of all properties.

PROPERTY	DESCRIPTION
LongDatePattern	Defines the format of the date component of the result string.
ShortTimePattern	Defines the format of the time component of the result string.
DayNames	Defines the localized day names that can appear in the result string.
MonthNames	Defines the localized month names that can appear in the result string.
TimeSeparator	Defines the string that separates the hour, minute, and second components of a time.
AMDesignator	Defines the string that indicates times from midnight to before noon in a 12-hour clock.
PMDesignator	Defines the string that indicates times from noon to before midnight in a 12-hour clock.

The following example uses the "f" format specifier to display a date and time value.

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("f",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008 6:30 AM
Console.WriteLine(date1.ToString("f",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays jeudi 10 avril 2008 06:30
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("f", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Thursday, April 10, 2008 6:30 AM
Console.WriteLine(date1.ToString("f", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays jeudi 10 avril 2008 06:30
```

Back to table

The Full Date Long Time ("F") Format Specifier

The "F" standard format specifier represents a custom date and time format string that is defined by the current DateTimeFormatInfo.FullDateTimePattern property. For example, the custom format string for the invariant culture is "dddd, dd MMMM yyyy HH:mm:ss".

The following table lists the DateTimeFormatInfo object properties that may control the formatting of the returned string. The custom format specifier that is returned by the FullDateTimePattern property of some cultures may not make use of all properties.

PROPERTY	DESCRIPTION
FullDateTimePattern	Defines the overall format of the result string.
DayNames	Defines the localized day names that can appear in the result string.
MonthNames	Defines the localized month names that can appear in the result string.
TimeSeparator	Defines the string that separates the hour, minute, and second components of a time.
AMDesignator	Defines the string that indicates times from midnight to before noon in a 12-hour clock.
PMDesignator	Defines the string that indicates times from noon to before midnight in a 12-hour clock.

The following example uses the "F" format specifier to display a date and time value.

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("F",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008 6:30:00 AM
Console.WriteLine(date1.ToString("F",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays jeudi 10 avril 2008 06:30:00
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("F", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Thursday, April 10, 2008 6:30:00 AM
Console.WriteLine(date1.ToString("F", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays jeudi 10 avril 2008 06:30:00
```

Back to table

The General Date Short Time ("g") Format Specifier

The "g" standard format specifier represents a combination of the short date ("d") and short time ("t") patterns, separated by a space.

The result string is affected by the formatting information of a specific DateTimeFormatInfo object. The following table lists the DateTimeFormatInfo object properties that may control the formatting of the returned string. The custom format specifier that is returned by the DateTimeFormatInfo.ShortDatePattern and DateTimeFormatInfo.ShortTimePattern properties of some cultures may not make use of all properties.

PROPERTY	DESCRIPTION
ShortDatePattern	Defines the format of the date component of the result string.
ShortTimePattern	Defines the format of the time component of the result string.
DateSeparator	Defines the string that separates the year, month, and day components of a date.
TimeSeparator	Defines the string that separates the hour, minute, and second components of a time.
AMDesignator	Defines the string that indicates times from midnight to before noon in a 12-hour clock.
PMDesignator	Defines the string that indicates times from noon to before midnight in a 12-hour clock.

The following example uses the "g" format specifier to display a date and time value.

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("g",
                  DateTimeFormatInfo.InvariantInfo));
// Displays 04/10/2008 06:30
Console.WriteLine(date1.ToString("g",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays 4/10/2008 6:30 AM
Console.WriteLine(date1.ToString("g",
                  CultureInfo.CreateSpecificCulture("fr-BE")));
// Displays 10/04/2008 6:30
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("g", _
                  DateTimeFormatInfo.InvariantInfo))
' Displays 04/10/2008 06:30
Console.WriteLine(date1.ToString("g", _
                  CultureInfo.CreateSpecificCulture("en-us")))
' Displays 4/10/2008 6:30 AM
Console.WriteLine(date1.ToString("g", _
                  CultureInfo.CreateSpecificCulture("fr-BE")))
' Displays 10/04/2008 6:30
```

Back to table

The General Date Long Time ("G") Format Specifier

The "G" standard format specifier represents a combination of the short date ("d") and long time ("T") patterns, separated by a space.

The result string is affected by the formatting information of a specific DateTimeFormatInfo object. The following table lists the DateTimeFormatInfo object properties that may control the formatting of the returned string. The custom format specifier that is returned by the DateTimeFormatInfo.ShortDatePattern and DateTimeFormatInfo.LongTimePattern properties of some cultures may not make use of all properties.

PROPERTY	DESCRIPTION
ShortDatePattern	Defines the format of the date component of the result string.
LongTimePattern	Defines the format of the time component of the result string.
DateSeparator	Defines the string that separates the year, month, and day components of a date.
TimeSeparator	Defines the string that separates the hour, minute, and second components of a time.
AMDesignator	Defines the string that indicates times from midnight to before noon in a 12-hour clock.
PMDesignator	Defines the string that indicates times from noon to before midnight in a 12-hour clock.

The following example uses the "G" format specifier to display a date and time value.

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("G",
                  DateTimeFormatInfo.InvariantInfo));
// Displays 04/10/2008 06:30:00
Console.WriteLine(date1.ToString("G",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays 4/10/2008 6:30:00 AM
Console.WriteLine(date1.ToString("G",
                  CultureInfo.CreateSpecificCulture("nl-BE")));
// Displays 10/04/2008 6:30:00
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("G", _
                  DateTimeFormatInfo.InvariantInfo))
' Displays 04/10/2008 06:30:00
Console.WriteLine(date1.ToString("G", _
                  CultureInfo.CreateSpecificCulture("en-us")))
' Displays 4/10/2008 6:30:00 AM
Console.WriteLine(date1.ToString("G", _
                  CultureInfo.CreateSpecificCulture("nl-BE")))
' Displays 10/04/2008 6:30:00
```

Back to table

The Month ("M", "m") Format Specifier

The "M" or "m" standard format specifier represents a custom date and time format string that is defined by the current DateTimeFormatInfo.MonthDayPattern property. For example, the custom format string for the invariant culture is "MMMM dd".

The following table lists the DateTimeFormatInfo object properties that control the formatting of the returned string.

PROPERTY	DESCRIPTION
MonthDayPattern	Defines the overall format of the result string.
MonthNames	Defines the localized month names that can appear in the result string.

The following example uses the "m" format specifier to display a date and time value.

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("m",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays April 10
Console.WriteLine(date1.ToString("m",
                  CultureInfo.CreateSpecificCulture("ms-MY")));
// Displays 10 April
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("m", _
                  CultureInfo.CreateSpecificCulture("en-us")))
' Displays April 10
Console.WriteLine(date1.ToString("m", _
                  CultureInfo.CreateSpecificCulture("ms-MY")))
' Displays 10 April
```

Back to table

The Round-trip ("O", "o") Format Specifier

The "O" or "o" standard format specifier represents a custom date and time format string using a pattern that preserves time zone information and emits a result string that complies with ISO 8601. For DateTime values, this format specifier is designed to preserve date and time values along with the DateTime.Kind property in text. The formatted string can be parsed back by using the DateTime.Parse(String, IFormatProvider, DateTimeStyles) or DateTime.ParseExact method if the `styles` parameter is set to DateTimeStyles.RoundtripKind.

The "O" or "o" standard format specifier corresponds to the "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffK" custom format string for DateTime values and to the "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffzzz" custom format string for DateTimeOffset values. In this string, the pairs of single quotation marks that delimit individual characters, such as the hyphens, the colons, and the letter "T", indicate that the individual character is a literal that cannot be changed. The apostrophes do not appear in the output string.

The "O" or "o" standard format specifier (and the "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffK" custom format string) takes advantage of the three ways that ISO 8601 represents time zone information to preserve the Kind property of DateTime values:

- The time zone component of DateTimeKind.Local date and time values is an offset from UTC (for example, +01:00, -07:00). All DateTimeOffset values are also represented in this format.
- The time zone component of DateTimeKind.Utc date and time values uses "Z" (which stands for zero offset) to represent UTC.
- DateTimeKind.Unspecified date and time values have no time zone information.

Because the O" or "o" standard format specifier conforms to an international standard, the formatting or parsing operation that uses the specifier always uses the invariant culture and the Gregorian calendar.

Strings that are passed to the `Parse`, `TryParse`, `ParseExact`, and `TryParseExact` methods of DateTime and DateTimeOffset can be parsed by using the "O" or "o" format specifier if they are in one of these formats. In the case of DateTime objects, the parsing overload that you call should also include a `styles` parameter with a value of DateTimeStyles.RoundtripKind. Note that if you call a parsing method with the custom format string that corresponds to the "O" or "o" format specifier, you won't get the same results as "O" or "o". This is because parsing methods that use a custom format string can't parse the string representation of date and time values that lack a time zone component or use "Z" to indicate UTC.

The following example uses the "o" format specifier to display a series of DateTime values and a DateTimeOffset value on a system in the U.S. Pacific Time zone.

```
using System;

public class Example
{
   public static void Main()
   {
       DateTime dat = new DateTime(2009, 6, 15, 13, 45, 30,
                                   DateTimeKind.Unspecified);
       Console.WriteLine("{0} ({1}) --> {0:O}", dat, dat.Kind);

       DateTime uDat = new DateTime(2009, 6, 15, 13, 45, 30,
                                    DateTimeKind.Utc);
       Console.WriteLine("{0} ({1}) --> {0:O}", uDat, uDat.Kind);

       DateTime lDat = new DateTime(2009, 6, 15, 13, 45, 30,
                                    DateTimeKind.Local);
       Console.WriteLine("{0} ({1}) --> {0:O}\n", lDat, lDat.Kind);

       DateTimeOffset dto = new DateTimeOffset(lDat);
       Console.WriteLine("{0} --> {0:O}", dto);
   }
}
// The example displays the following output:
//    6/15/2009 1:45:30 PM (Unspecified) --> 2009-06-15T13:45:30.0000000
//    6/15/2009 1:45:30 PM (Utc) --> 2009-06-15T13:45:30.0000000Z
//    6/15/2009 1:45:30 PM (Local) --> 2009-06-15T13:45:30.0000000-07:00
//
//    6/15/2009 1:45:30 PM -07:00 --> 2009-06-15T13:45:30.0000000-07:00
```

```
Module Example
   Public Sub Main()
       Dim dat As New Date(2009, 6, 15, 13, 45, 30,
                           DateTimeKind.Unspecified)
       Console.WriteLine("{0} ({1}) --> {0:O}", dat, dat.Kind)

       Dim uDat As New Date(2009, 6, 15, 13, 45, 30, DateTimeKind.Utc)
       Console.WriteLine("{0} ({1}) --> {0:O}", uDat, uDat.Kind)

       Dim lDat As New Date(2009, 6, 15, 13, 45, 30, DateTimeKind.Local)
       Console.WriteLine("{0} ({1}) --> {0:O}", lDat, lDat.Kind)
       Console.WriteLine()

       Dim dto As New DateTimeOffset(lDat)
       Console.WriteLine("{0} --> {0:O}", dto)
   End Sub
End Module
' The example displays the following output:
'    6/15/2009 1:45:30 PM (Unspecified) --> 2009-06-15T13:45:30.0000000
'    6/15/2009 1:45:30 PM (Utc) --> 2009-06-15T13:45:30.0000000Z
'    6/15/2009 1:45:30 PM (Local) --> 2009-06-15T13:45:30.0000000-07:00
'
'    6/15/2009 1:45:30 PM -07:00 --> 2009-06-15T13:45:30.0000000-07:00
```

The following example uses the "o" format specifier to create a formatted string, and then restores the original date and time value by calling a date and time `Parse` method.

```
// Round-trip DateTime values.
DateTime originalDate, newDate;
string dateString;
// Round-trip a local time.
originalDate = DateTime.SpecifyKind(new DateTime(2008, 4, 10, 6, 30, 0), DateTimeKind.Local);
dateString = originalDate.ToString("o");
newDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate, originalDate.Kind,
                  newDate, newDate.Kind);
// Round-trip a UTC time.
originalDate = DateTime.SpecifyKind(new DateTime(2008, 4, 12, 9, 30, 0), DateTimeKind.Utc);
dateString = originalDate.ToString("o");
newDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate, originalDate.Kind,
                  newDate, newDate.Kind);
// Round-trip time in an unspecified time zone.
originalDate = DateTime.SpecifyKind(new DateTime(2008, 4, 13, 12, 30, 0), DateTimeKind.Unspecified);
dateString = originalDate.ToString("o");
newDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate, originalDate.Kind,
                  newDate, newDate.Kind);

// Round-trip a DateTimeOffset value.
DateTimeOffset originalDTO = new DateTimeOffset(2008, 4, 12, 9, 30, 0, new TimeSpan(-8, 0, 0));
dateString = originalDTO.ToString("o");
DateTimeOffset newDTO = DateTimeOffset.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Round-tripped {0} to {1}.", originalDTO, newDTO);
// The example displays the following output:
//    Round-tripped 4/10/2008 6:30:00 AM Local to 4/10/2008 6:30:00 AM Local.
//    Round-tripped 4/12/2008 9:30:00 AM Utc to 4/12/2008 9:30:00 AM Utc.
//    Round-tripped 4/13/2008 12:30:00 PM Unspecified to 4/13/2008 12:30:00 PM Unspecified.
//    Round-tripped 4/12/2008 9:30:00 AM -08:00 to 4/12/2008 9:30:00 AM -08:00.
```

```
' Round-trip DateTime values.
Dim originalDate, newDate As Date
Dim dateString As String
' Round-trip a local time.
originalDate = Date.SpecifyKind(#4/10/2008 6:30AM#, DateTimeKind.Local)
dateString = originalDate.ToString("o")
newDate = Date.Parse(dateString, Nothing, DateTimeStyles.RoundtripKind)
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate, originalDate.Kind, _
                  newDate, newDate.Kind)
' Round-trip a UTC time.
originalDate = Date.SpecifyKind(#4/12/2008 9:30AM#, DateTimeKind.Utc)
dateString = originalDate.ToString("o")
newDate = Date.Parse(dateString, Nothing, DateTimeStyles.RoundtripKind)
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate, originalDate.Kind, _
                  newDate, newDate.Kind)
' Round-trip time in an unspecified time zone.
originalDate = Date.SpecifyKind(#4/13/2008 12:30PM#, DateTimeKind.Unspecified)
dateString = originalDate.ToString("o")
newDate = Date.Parse(dateString, Nothing, DateTimeStyles.RoundtripKind)
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate, originalDate.Kind, _
                  newDate, newDate.Kind)

' Round-trip a DateTimeOffset value.
Dim originalDTO As New DateTimeOffset(#4/12/2008 9:30AM#, New TimeSpan(-8, 0, 0))
dateString = originalDTO.ToString("o")
Dim newDTO As DateTimeOffset = DateTimeOffset.Parse(dateString, Nothing, DateTimeStyles.RoundtripKind)
Console.WriteLine("Round-tripped {0} to {1}.", originalDTO, newDTO)
' The example displays the following output:
'    Round-tripped 4/10/2008 6:30:00 AM Local to 4/10/2008 6:30:00 AM Local.
'    Round-tripped 4/12/2008 9:30:00 AM Utc to 4/12/2008 9:30:00 AM Utc.
'    Round-tripped 4/13/2008 12:30:00 PM Unspecified to 4/13/2008 12:30:00 PM Unspecified.
'    Round-tripped 4/12/2008 9:30:00 AM -08:00 to 4/12/2008 9:30:00 AM -08:00.
```

Back to table

The RFC1123 ("R", "r") Format Specifier

The "R" or "r" standard format specifier represents a custom date and time format string that is defined by the DateTimeFormatInfo.RFC1123Pattern property. The pattern reflects a defined standard, and the property is read-only. Therefore, it is always the same, regardless of the culture used or the format provider supplied. The custom format string is "ddd, dd MMM yyyy HH':'mm':'ss 'GMT'". When this standard format specifier is used, the formatting or parsing operation always uses the invariant culture.

The result string is affected by the following properties of the DateTimeFormatInfo object returned by the DateTimeFormatInfo.InvariantInfo property that represents the invariant culture.

PROPERTY	DESCRIPTION
RFC1123Pattern	Defines the format of the result string.
AbbreviatedDayNames	Defines the abbreviated day names that can appear in the result string.
AbbreviatedMonthNames	Defines the abbreviated month names that can appear in the result string.

Although the RFC 1123 standard expresses a time as Coordinated Universal Time (UTC), the formatting operation does not modify the value of the DateTime object that is being formatted. Therefore, you must convert the DateTime value to UTC by calling the DateTime.ToUniversalTime method before you perform the formatting operation. In contrast, DateTimeOffset values perform this conversion automatically; there is no need to call the

DateTimeOffset.ToUniversalTime method before the formatting operation.

The following example uses the "r" format specifier to display a DateTime and a DateTimeOffset value on a system in the U.S. Pacific Time zone.

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
DateTimeOffset dateOffset = new DateTimeOffset(date1,
                            TimeZoneInfo.Local.GetUtcOffset(date1));
Console.WriteLine(date1.ToUniversalTime().ToString("r"));
// Displays Thu, 10 Apr 2008 13:30:00 GMT
Console.WriteLine(dateOffset.ToUniversalTime().ToString("r"));
// Displays Thu, 10 Apr 2008 13:30:00 GMT
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Dim dateOffset As New DateTimeOffset(date1, TimeZoneInfo.Local.GetUtcOFfset(date1))
Console.WriteLine(date1.ToUniversalTime.ToString("r"))
' Displays Thu, 10 Apr 2008 13:30:00 GMT
Console.WriteLine(dateOffset.ToUniversalTime.ToString("r"))
' Displays Thu, 10 Apr 2008 13:30:00 GMT
```

Back to table

The Sortable ("s") Format Specifier

The "s" standard format specifier represents a custom date and time format string that is defined by the DateTimeFormatInfo.SortableDateTimePattern property. The pattern reflects a defined standard (ISO 8601), and the property is read-only. Therefore, it is always the same, regardless of the culture used or the format provider supplied. The custom format string is "yyyy'-'MM'-'dd'T'HH':'mm':'ss".

The purpose of the "s" format specifier is to produce result strings that sort consistently in ascending or descending order based on date and time values. As a result, although the "s" standard format specifier represents a date and time value in a consistent format, the formatting operation does not modify the value of the date and time object that is being formatted to reflect its DateTime.Kind property or its DateTimeOffset.Offset value. For example, the result strings produced by formatting the date and time values 2014-11-15T18:32:17+00:00 and 2014-11-15T18:32:17+08:00 are identical.

When this standard format specifier is used, the formatting or parsing operation always uses the invariant culture.

The following example uses the "s" format specifier to display a DateTime and a DateTimeOffset value on a system in the U.S. Pacific Time zone.

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("s"));
// Displays 2008-04-10T06:30:00
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("s"))
' Displays 2008-04-10T06:30:00
```

Back to table

The Short Time ("t") Format Specifier

The "t" standard format specifier represents a custom date and time format string that is defined by the current

DateTimeFormatInfo.ShortTimePattern property. For example, the custom format string for the invariant culture is "HH:mm".

The result string is affected by the formatting information of a specific DateTimeFormatInfo object. The following table lists the DateTimeFormatInfo object properties that may control the formatting of the returned string. The custom format specifier that is returned by the DateTimeFormatInfo.ShortTimePattern property of some cultures may not make use of all properties.

PROPERTY	DESCRIPTION
ShortTimePattern	Defines the format of the time component of the result string.
TimeSeparator	Defines the string that separates the hour, minute, and second components of a time.
AMDesignator	Defines the string that indicates times from midnight to before noon in a 12-hour clock.
PMDesignator	Defines the string that indicates times from noon to before midnight in a 12-hour clock.

The following example uses the "t" format specifier to display a date and time value.

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("t",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays 6:30 AM
Console.WriteLine(date1.ToString("t",
                  CultureInfo.CreateSpecificCulture("es-ES")));
// Displays 6:30
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("t", _
                  CultureInfo.CreateSpecificCulture("en-us")))
' Displays 6:30 AM
Console.WriteLine(date1.ToString("t", _
                  CultureInfo.CreateSpecificCulture("es-ES")))
' Displays 6:30
```

Back to table

The Long Time ("T") Format Specifier

The "T" standard format specifier represents a custom date and time format string that is defined by a specific culture's DateTimeFormatInfo.LongTimePattern property. For example, the custom format string for the invariant culture is "HH:mm:ss".

The following table lists the DateTimeFormatInfo object properties that may control the formatting of the returned string. The custom format specifier that is returned by the DateTimeFormatInfo.LongTimePattern property of some cultures may not make use of all properties.

PROPERTY	DESCRIPTION
LongTimePattern	Defines the format of the time component of the result string.

PROPERTY	DESCRIPTION
TimeSeparator	Defines the string that separates the hour, minute, and second components of a time.
AMDesignator	Defines the string that indicates times from midnight to before noon in a 12-hour clock.
PMDesignator	Defines the string that indicates times from noon to before midnight in a 12-hour clock.

The following example uses the "T" format specifier to display a date and time value.

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("T",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays 6:30:00 AM
Console.WriteLine(date1.ToString("T",
                  CultureInfo.CreateSpecificCulture("es-ES")));
// Displays 6:30:00
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("T", _
                  CultureInfo.CreateSpecificCulture("en-us")))
' Displays 6:30:00 AM
Console.WriteLine(date1.ToString("T", _
                  CultureInfo.CreateSpecificCulture("es-ES")))
' Displays 6:30:00
```

Back to table

The Universal Sortable ("u") Format Specifier

The "u" standard format specifier represents a custom date and time format string that is defined by the DateTimeFormatInfo.UniversalSortableDateTimePattern property. The pattern reflects a defined standard, and the property is read-only. Therefore, it is always the same, regardless of the culture used or the format provider supplied. The custom format string is "yyyy'-'MM'-'dd HH':'mm':'ss'Z'". When this standard format specifier is used, the formatting or parsing operation always uses the invariant culture.

Although the result string should express a time as Coordinated Universal Time (UTC), no conversion of the original DateTime value is performed during the formatting operation. Therefore, you must convert a DateTime value to UTC by calling the DateTime.ToUniversalTime method before formatting it. In contrast, DateTimeOffset values perform this conversion automatically; there is no need to call the DateTimeOffset.ToUniversalTime method before the formatting operation.

The following example uses the "u" format specifier to display a date and time value.

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToUniversalTime().ToString("u"));
// Displays 2008-04-10 13:30:00Z
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToUniversalTime.ToString("u"))
' Displays 2008-04-10 13:30:00Z
```

[Back to table](#)

The Universal Full ("U") Format Specifier

The "U" standard format specifier represents a custom date and time format string that is defined by a specified culture's DateTimeFormatInfo.FullDateTimePattern property. The pattern is the same as the "F" pattern. However, the DateTime value is automatically converted to UTC before it is formatted.

The following table lists the DateTimeFormatInfo object properties that may control the formatting of the returned string. The custom format specifier that is returned by the FullDateTimePattern property of some cultures may not make use of all properties.

PROPERTY	DESCRIPTION
FullDateTimePattern	Defines the overall format of the result string.
DayNames	Defines the localized day names that can appear in the result string.
MonthNames	Defines the localized month names that can appear in the result string.
TimeSeparator	Defines the string that separates the hour, minute, and second components of a time.
AMDesignator	Defines the string that indicates times from midnight to before noon in a 12-hour clock.
PMDesignator	Defines the string that indicates times from noon to before midnight in a 12-hour clock.

The "U" format specifier is not supported by the DateTimeOffset type and throws a FormatException if it is used to format a DateTimeOffset value.

The following example uses the "U" format specifier to display a date and time value.

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("U",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008 1:30:00 PM
Console.WriteLine(date1.ToString("U",
                  CultureInfo.CreateSpecificCulture("sv-FI")));
// Displays den 10 april 2008 13:30:00
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("U", CultureInfo.CreateSpecificCulture("en-US")))
' Displays Thursday, April 10, 2008 1:30:00 PM
Console.WriteLine(date1.ToString("U", CultureInfo.CreateSpecificCulture("sv-FI")))
' Displays den 10 april 2008 13:30:00
```

[Back to table](#)

The Year Month ("Y", "y") Format Specifier

The "Y" or "y" standard format specifier represents a custom date and time format string that is defined by the DateTimeFormatInfo.YearMonthPattern property of a specified culture. For example, the custom format string for the invariant culture is "yyyy MMMM".

The following table lists the DateTimeFormatInfo object properties that control the formatting of the returned string.

PROPERTY	DESCRIPTION
YearMonthPattern	Defines the overall format of the result string.
MonthNames	Defines the localized month names that can appear in the result string.

The following example uses the "y" format specifier to display a date and time value.

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("Y",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays April, 2008
Console.WriteLine(date1.ToString("y",
                  CultureInfo.CreateSpecificCulture("af-ZA")));
// Displays April 2008
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("Y", CultureInfo.CreateSpecificCulture("en-US")))
' Displays April, 2008
Console.WriteLine(date1.ToString("y", CultureInfo.CreateSpecificCulture("af-ZA")))
' Displays April 2008
```

Back to table

Notes

Control Panel Settings

The settings in the **Regional and Language Options** item in Control Panel influence the result string produced by a formatting operation. These settings are used to initialize the DateTimeFormatInfo object associated with the current thread culture, which provides values used to govern formatting. Computers that use different settings generate different result strings.

In addition, if you use the CultureInfo.CultureInfo(String) constructor to instantiate a new CultureInfo object that represents the same culture as the current system culture, any customizations established by the **Regional and Language Options** item in Control Panel will be applied to the new CultureInfo object. You can use the CultureInfo.CultureInfo(String, Boolean) constructor to create a CultureInfo object that does not reflect a system's customizations.

DateTimeFormatInfo Properties

Formatting is influenced by properties of the current DateTimeFormatInfo object, which is provided implicitly by the current thread culture or explicitly by the IFormatProvider parameter of the method that invokes formatting. For the IFormatProvider parameter, your application should specify a CultureInfo object, which represents a culture, or a DateTimeFormatInfo object, which represents a particular culture's date and time formatting conventions. Many of the standard date and time format specifiers are aliases for formatting patterns defined by properties of the current DateTimeFormatInfo object. Your application can change the result produced by some standard date and time format specifiers by changing the corresponding date and time format patterns of the corresponding DateTimeFormatInfo property.

See also

- System.DateTime
- System.DateTimeOffset
- Formatting Types
- Custom Date and Time Format Strings
- Sample: .NET Framework 4 Formatting Utility

Custom Date and Time Format Strings

9/6/2018 • 54 minutes to read • Edit Online

A date and time format string defines the text representation of a DateTime or DateTimeOffset value that results from a formatting operation . It can also define the representation of a date and time value that is required in a parsing operation in order to successfully convert the string to a date and time. A custom format string consists of one or more custom date and time format specifiers. Any string that is not a standard date and time format string is interpreted as a custom date and time format string.

TIP

You can download the Formatting Utility, an application that enables you to apply format strings to either date and time or numeric values and displays the result string.

Custom date and time format strings can be used with both DateTime and DateTimeOffset values.

NOTE

The 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.

In formatting operations, custom date and time format strings can be used either with the `ToString` method of a date and time instance or with a method that supports composite formatting. The following example illustrates both uses.

```
DateTime thisDate1 = new DateTime(2011, 6, 10);
Console.WriteLine("Today is " + thisDate1.ToString("MMMM dd, yyyy") + ".");

DateTimeOffset thisDate2 = new DateTimeOffset(2011, 6, 10, 15, 24, 16,
                                              TimeSpan.Zero);
Console.WriteLine("The current date and time: {0:MM/dd/yy H:mm:ss zzz}",
                   thisDate2);
// The example displays the following output:
//    Today is June 10, 2011.
//    The current date and time: 06/10/11 15:24:16 +00:00
```

```
 Dim thisDate1 As Date = #6/10/2011#
 Console.WriteLine("Today is " + thisDate1.ToString("MMMM dd, yyyy") + ".")

 Dim thisDate2 As New DateTimeOffset(2011, 6, 10, 15, 24, 16, TimeSpan.Zero)
 Console.WriteLine("The current date and time: {0:MM/dd/yy H:mm:ss zzz}",
                   thisDate2)
 ' The example displays the following output:
'    Today is June 10, 2011.
'    The current date and time: 06/10/11 15:24:16 +00:00
```

In parsing operations, custom date and time format strings can be used with the DateTime.ParseExact, DateTime.TryParseExact, DateTimeOffset.ParseExact, and DateTimeOffset.TryParseExact methods. These methods require that an input string conform exactly to a particular pattern for the parse operation to succeed. The following example illustrates a call to the DateTimeOffset.ParseExact(String, String, IFormatProvider) method to parse a date that must include a day, a month, and a two-digit year.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string[] dateValues = { "30-12-2011", "12-30-2011",
                              "30-12-11", "12-30-11" };
      string pattern = "MM-dd-yy";
      DateTime parsedDate;

      foreach (var dateValue in dateValues) {
         if (DateTime.TryParseExact(dateValue, pattern, null,
                                   DateTimeStyles.None, out parsedDate))
            Console.WriteLine("Converted '{0}' to {1:d}.",
                              dateValue, parsedDate);
         else
            Console.WriteLine("Unable to convert '{0}' to a date and time.",
                              dateValue);
      }
   }
}
// The example displays the following output:
//    Unable to convert '30-12-2011' to a date and time.
//    Unable to convert '12-30-2011' to a date and time.
//    Unable to convert '30-12-11' to a date and time.
//    Converted '12-30-11' to 12/30/2011.
```

```
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim dateValues() As String = { "30-12-2011", "12-30-2011",
                                     "30-12-11", "12-30-11" }
      Dim pattern As String = "MM-dd-yy"
      Dim parsedDate As Date

      For Each dateValue As String In dateValues
         If DateTime.TryParseExact(dateValue, pattern, Nothing,
                                   DateTimeStyles.None, parsedDate) Then
            Console.WriteLine("Converted '{0}' to {1:d}.",
                              dateValue, parsedDate)
         Else
            Console.WriteLine("Unable to convert '{0}' to a date and time.",
                              dateValue)
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'    Unable to convert '30-12-2011' to a date and time.
'    Unable to convert '12-30-2011' to a date and time.
'    Unable to convert '30-12-11' to a date and time.
'    Converted '12-30-11' to 12/30/2011.
```

The following table describes the custom date and time format specifiers and displays a result string produced by each format specifier. By default, result strings reflect the formatting conventions of the en-US culture. If a particular format specifier produces a localized result string, the example also notes the culture to which the result string applies. See the Notes section for additional information about using custom date and time format strings.

FORMAT SPECIFIER	DESCRIPTION	EXAMPLES
"d"	The day of the month, from 1 through 31. More information: The "d" Custom Format Specifier.	2009-06-01T13:45:30 -> 1 2009-06-15T13:45:30 -> 15
"dd"	The day of the month, from 01 through 31. More information: The "dd" Custom Format Specifier.	2009-06-01T13:45:30 -> 01 2009-06-15T13:45:30 -> 15
"ddd"	The abbreviated name of the day of the week. More information: The "ddd" Custom Format Specifier.	2009-06-15T13:45:30 -> Mon (en-US) 2009-06-15T13:45:30 -> Пн (ru-RU) 2009-06-15T13:45:30 -> lun. (fr-FR)
"dddd"	The full name of the day of the week. More information: The "dddd" Custom Format Specifier.	2009-06-15T13:45:30 -> Monday (en-US) 2009-06-15T13:45:30 -> понедельник (ru-RU) 2009-06-15T13:45:30 -> lundi (fr-FR)
"f"	The tenths of a second in a date and time value. More information: The "f" Custom Format Specifier.	2009-06-15T13:45:30.6170000 -> 6 2009-06-15T13:45:30.05 -> 0
"ff"	The hundredths of a second in a date and time value. More information: The "ff" Custom Format Specifier.	2009-06-15T13:45:30.6170000 -> 61 2009-06-15T13:45:30.0050000 -> 00
"fff"	The milliseconds in a date and time value. More information: The "fff" Custom Format Specifier.	6/15/2009 13:45:30.617 -> 617 6/15/2009 13:45:30.0005 -> 000
"ffff"	The ten thousandths of a second in a date and time value. More information: The "ffff" Custom Format Specifier.	2009-06-15T13:45:30.6175000 -> 6175 2009-06-15T13:45:30.0000500 -> 0000

FORMAT SPECIFIER	DESCRIPTION	EXAMPLES
"fffff"	The hundred thousandths of a second in a date and time value. More information: The "fffff" Custom Format Specifier.	2009-06-15T13:45:30.6175400 -> 61754 6/15/2009 13:45:30.000005 -> 00000
"ffffff"	The millionths of a second in a date and time value. More information: The "ffffff" Custom Format Specifier.	2009-06-15T13:45:30.6175420 -> 617542 2009-06-15T13:45:30.0000005 -> 000000
"fffffff"	The ten millionths of a second in a date and time value. More information: The "fffffff" Custom Format Specifier.	2009-06-15T13:45:30.6175425 -> 6175425 2009-06-15T13:45:30.0001150 -> 0001150
"F"	If non-zero, the tenths of a second in a date and time value. More information: The "F" Custom Format Specifier.	2009-06-15T13:45:30.6170000 -> 6 2009-06-15T13:45:30.0500000 -> (no output)
"FF"	If non-zero, the hundredths of a second in a date and time value. More information: The "FF" Custom Format Specifier.	2009-06-15T13:45:30.6170000 -> 61 2009-06-15T13:45:30.0050000 -> (no output)
"FFF"	If non-zero, the milliseconds in a date and time value. More information: The "FFF" Custom Format Specifier.	2009-06-15T13:45:30.6170000 -> 617 2009-06-15T13:45:30.0005000 -> (no output)
"FFFF"	If non-zero, the ten thousandths of a second in a date and time value. More information: The "FFFF" Custom Format Specifier.	2009-06-15T13:45:30.5275000 -> 5275 2009-06-15T13:45:30.0000500 -> (no output)
"FFFFF"	If non-zero, the hundred thousandths of a second in a date and time value. More information: The "FFFFF" Custom Format Specifier.	2009-06-15T13:45:30.6175400 -> 61754 2009-06-15T13:45:30.0000050 -> (no output)
"FFFFFF"	If non-zero, the millionths of a second in a date and time value. More information: The "FFFFFF" Custom Format Specifier.	2009-06-15T13:45:30.6175420 -> 617542 2009-06-15T13:45:30.0000005 -> (no output)

FORMAT SPECIFIER	DESCRIPTION	EXAMPLES
"FFFFFFF"	If non-zero, the ten millionths of a second in a date and time value. More information: The "FFFFFFF" Custom Format Specifier.	2009-06-15T13:45:30.6175425 -> 6175425 2009-06-15T13:45:30.0001150 -> 000115
"g", "gg"	The period or era. More information: The "g" or "gg" Custom Format Specifier.	2009-06-15T13:45:30.6170000 -> A.D.
"h"	The hour, using a 12-hour clock from 1 to 12. More information: The "h" Custom Format Specifier.	2009-06-15T01:45:30 -> 1 2009-06-15T13:45:30 -> 1
"hh"	The hour, using a 12-hour clock from 01 to 12. More information: The "hh" Custom Format Specifier.	2009-06-15T01:45:30 -> 01 2009-06-15T13:45:30 -> 01
"H"	The hour, using a 24-hour clock from 0 to 23. More information: The "H" Custom Format Specifier.	2009-06-15T01:45:30 -> 1 2009-06-15T13:45:30 -> 13
"HH"	The hour, using a 24-hour clock from 00 to 23. More information: The "HH" Custom Format Specifier.	2009-06-15T01:45:30 -> 01 2009-06-15T13:45:30 -> 13
"K"	Time zone information. More information: The "K" Custom Format Specifier.	With DateTime values: 2009-06-15T13:45:30, Kind Unspecified -> 2009-06-15T13:45:30, Kind Utc -> Z 2009-06-15T13:45:30, Kind Local -> -07:00 (depends on local computer settings) With DateTimeOffset values: 2009-06-15T01:45:30-07:00 --> -07:00 2009-06-15T08:45:30+00:00 --> +00:00
"m"	The minute, from 0 through 59. More information: The "m" Custom Format Specifier.	2009-06-15T01:09:30 -> 9 2009-06-15T13:29:30 -> 29

FORMAT SPECIFIER	DESCRIPTION	EXAMPLES
"mm"	The minute, from 00 through 59. More information: The "mm" Custom Format Specifier.	2009-06-15T01:09:30 -> 09 2009-06-15T01:45:30 -> 45
"M"	The month, from 1 through 12. More information: The "M" Custom Format Specifier.	2009-06-15T13:45:30 -> 6
"MM"	The month, from 01 through 12. More information: The "MM" Custom Format Specifier.	2009-06-15T13:45:30 -> 06
"MMM"	The abbreviated name of the month. More information: The "MMM" Custom Format Specifier.	2009-06-15T13:45:30 -> Jun (en-US) 2009-06-15T13:45:30 -> juin (fr-FR) 2009-06-15T13:45:30 -> Jun (zu-ZA)
"MMMM"	The full name of the month. More information: The "MMMM" Custom Format Specifier.	2009-06-15T13:45:30 -> June (en-US) 2009-06-15T13:45:30 -> juni (da-DK) 2009-06-15T13:45:30 -> uJuni (zu-ZA)
"s"	The second, from 0 through 59. More information: The "s" Custom Format Specifier.	2009-06-15T13:45:09 -> 9
"ss"	The second, from 00 through 59. More information: The "ss" Custom Format Specifier.	2009-06-15T13:45:09 -> 09
"t"	The first character of the AM/PM designator. More information: The "t" Custom Format Specifier.	2009-06-15T13:45:30 -> P (en-US) 2009-06-15T13:45:30 -> 午 (ja-JP) 2009-06-15T13:45:30 -> (fr-FR)
"tt"	The AM/PM designator. More information: The "tt" Custom Format Specifier.	2009-06-15T13:45:30 -> PM (en-US) 2009-06-15T13:45:30 -> 午後 (ja-JP) 2009-06-15T13:45:30 -> (fr-FR)

FORMAT SPECIFIER	DESCRIPTION	EXAMPLES
"y"	The year, from 0 to 99. More information: The "y" Custom Format Specifier.	0001-01-01T00:00:00 -> 1 0900-01-01T00:00:00 -> 0 1900-01-01T00:00:00 -> 0 2009-06-15T13:45:30 -> 9 2019-06-15T13:45:30 -> 19
"yy"	The year, from 00 to 99. More information: The "yy" Custom Format Specifier.	0001-01-01T00:00:00 -> 01 0900-01-01T00:00:00 -> 00 1900-01-01T00:00:00 -> 00 2019-06-15T13:45:30 -> 19
"yyy"	The year, with a minimum of three digits. More information: The "yyy" Custom Format Specifier.	0001-01-01T00:00:00 -> 001 0900-01-01T00:00:00 -> 900 1900-01-01T00:00:00 -> 1900 2009-06-15T13:45:30 -> 2009
"yyyy"	The year as a four-digit number. More information: The "yyyy" Custom Format Specifier.	0001-01-01T00:00:00 -> 0001 0900-01-01T00:00:00 -> 0900 1900-01-01T00:00:00 -> 1900 2009-06-15T13:45:30 -> 2009
"yyyyy"	The year as a five-digit number. More information: The "yyyyy" Custom Format Specifier.	0001-01-01T00:00:00 -> 00001 2009-06-15T13:45:30 -> 02009
"z"	Hours offset from UTC, with no leading zeros. More information: The "z" Custom Format Specifier.	2009-06-15T13:45:30-07:00 -> -7
"zz"	Hours offset from UTC, with a leading zero for a single-digit value. More information: The "zz" Custom Format Specifier.	2009-06-15T13:45:30-07:00 -> -07
"zzz"	Hours and minutes offset from UTC. More information: The "zzz" Custom Format Specifier.	2009-06-15T13:45:30-07:00 -> -07:00

FORMAT SPECIFIER	DESCRIPTION	EXAMPLES
":"	The time separator. More information: The ":" Custom Format Specifier.	2009-06-15T13:45:30 -> : (en-US) 2009-06-15T13:45:30 -> . (it-IT) 2009-06-15T13:45:30 -> : (ja-JP)
"/"	The date separator. More Information: The "/" Custom Format Specifier.	2009-06-15T13:45:30 -> / (en-US) 2009-06-15T13:45:30 -> - (ar-DZ) 2009-06-15T13:45:30 -> . (tr-TR)
"*string*" '*string*'	Literal string delimiter. More information: Character literals.	2009-06-15T13:45:30 ("arr:" h:m t) -> arr: 1:45 P 2009-06-15T13:45:30 ('arr:' h:m t) -> arr: 1:45 P
%	Defines the following character as a custom format specifier. More information:Using Single Custom Format Specifiers.	2009-06-15T13:45:30 (%h) -> 1
|The escape character. More information: Character literals and Using the Escape Character.	2009-06-15T13:45:30 (h \h) -> 1 h	
Any other character	The character is copied to the result string unchanged. More information: Character literals.	2009-06-15T01:45:30 (arr hh:mm t) -> arr 01:45 A

The following sections provide additional information about each custom date and time format specifier. Unless otherwise noted, each specifier produces an identical string representation regardless of whether it is used with a DateTime value or a DateTimeOffset value.

The "d" custom format specifier

The "d" custom format specifier represents the day of the month as a number from 1 through 31. A single-digit day is formatted without a leading zero.

If the "d" format specifier is used without other custom format specifiers, it is interpreted as the "d" standard date and time format specifier. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

The following example includes the "d" custom format specifier in several format strings.

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("d, M",
                  CultureInfo.InvariantCulture));
// Displays 29, 8

Console.WriteLine(date1.ToString("d MMMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays 29 August
Console.WriteLine(date1.ToString("d MMMM",
                  CultureInfo.CreateSpecificCulture("es-MX")));
// Displays 29 agosto
```

```
Dim date1 As Date = #08/29/2008 7:27:15PM#

Console.WriteLine(date1.ToString("d, M", _
                  CultureInfo.InvariantCulture))
' Displays 29, 8

Console.WriteLine(date1.ToString("d MMMM", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays 29 August
Console.WriteLine(date1.ToString("d MMMM", _
                  CultureInfo.CreateSpecificCulture("es-MX")))
' Displays 29 agosto
```

Back to table

The "dd" custom format specifier

The "dd" custom format string represents the day of the month as a number from 01 through 31. A single-digit day is formatted with a leading zero.

The following example includes the "dd" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 1, 2, 6, 30, 15);

Console.WriteLine(date1.ToString("dd, MM",
                  CultureInfo.InvariantCulture));
// 02, 01
```

```
Dim date1 As Date = #1/2/2008 6:30:15AM#

Console.WriteLine(date1.ToString("dd, MM", _
                  CultureInfo.InvariantCulture))
' 02, 01
```

Back to table

The "ddd" custom format specifier

The "ddd" custom format specifier represents the abbreviated name of the day of the week. The localized abbreviated name of the day of the week is retrieved from the DateTimeFormatInfo.AbbreviatedDayNames property of the current or specified culture.

The following example includes the "ddd" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("ddd d MMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Fri 29 Aug
Console.WriteLine(date1.ToString("ddd d MMM",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays ven. 29 août
```

```
Dim date1 As Date = #08/29/2008 7:27:15PM#

Console.WriteLine(date1.ToString("ddd d MMM", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Fri 29 Aug
Console.WriteLine(date1.ToString("ddd d MMM", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays ven. 29 août
```

Back to table

The "dddd" custom format specifier

The "dddd" custom format specifier (plus any number of additional "d" specifiers) represents the full name of the day of the week. The localized name of the day of the week is retrieved from the DateTimeFormatInfo.DayNames property of the current or specified culture.

The following example includes the "dddd" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("dddd dd MMMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Friday 29 August
Console.WriteLine(date1.ToString("dddd dd MMMM",
                  CultureInfo.CreateSpecificCulture("it-IT")));
// Displays venerdì 29 agosto
```

```
Dim date1 As Date = #08/29/2008 7:27:15PM#

Console.WriteLine(date1.ToString("dddd dd MMMM", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Friday 29 August
Console.WriteLine(date1.ToString("dddd dd MMMM", _
                  CultureInfo.CreateSpecificCulture("it-IT")))
' Displays venerdì 29 agosto
```

Back to table

The "f" custom format specifier

The "f" custom format specifier represents the most significant digit of the seconds fraction; that is, it represents the tenths of a second in a date and time value.

If the "f" format specifier is used without other format specifiers, it is interpreted as the "f" standard date and time format specifier. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

When you use "f" format specifiers as part of a format string supplied to the ParseExact, TryParseExact, ParseExact, or TryParseExact method, the number of "f" format specifiers indicates the number of most significant digits of the seconds fraction that must be present to successfully parse the string.

The following example includes the "f" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

```
Dim date1 As New Date(2008, 8, 29, 19, 27, 15, 018)
Dim ci As CultureInfo = CultureInfo.InvariantCulture

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci))
' Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci))
' Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci))
' Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci))
' Displays 07:27:15.018
```

Back to table

The "ff" custom format specifier

The "ff" custom format specifier represents the two most significant digits of the seconds fraction; that is, it represents the hundredths of a second in a date and time value.

following example includes the "ff" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

```
Dim date1 As New Date(2008, 8, 29, 19, 27, 15, 018)
Dim ci As CultureInfo = CultureInfo.InvariantCulture

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci))
' Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci))
' Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci))
' Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci))
' Displays 07:27:15.018
```

Back to table

The "fff" custom format specifier

The "fff" custom format specifier represents the three most significant digits of the seconds fraction; that is, it represents the milliseconds in a date and time value.

The following example includes the "fff" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

```
Dim date1 As New Date(2008, 8, 29, 19, 27, 15, 018)
Dim ci As CultureInfo = CultureInfo.InvariantCulture

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci))
' Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci))
' Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci))
' Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci))
' Displays 07:27:15.018
```

Back to table

The "ffff" custom format specifier

The "ffff" custom format specifier represents the four most significant digits of the seconds fraction; that is, it represents the ten thousandths of a second in a date and time value.

Although it is possible to display the ten thousandths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT version 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

Back to table

The "fffff" custom format specifier

The "fffff" custom format specifier represents the five most significant digits of the seconds fraction; that is, it represents the hundred thousandths of a second in a date and time value.

Although it is possible to display the hundred thousandths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

Back to table

The "ffffff" custom format specifier

The "ffffff" custom format specifier represents the six most significant digits of the seconds fraction; that is, it represents the millionths of a second in a date and time value.

Although it is possible to display the millionths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

Back to table

The "fffffff" custom format specifier

The "fffffff" custom format specifier represents the seven most significant digits of the seconds fraction; that is, it represents the ten millionths of a second in a date and time value.

Although it is possible to display the ten millionths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

Back to table

The "F" custom format specifier

The "F" custom format specifier represents the most significant digit of the seconds fraction; that is, it represents the tenths of a second in a date and time value. Nothing is displayed if the digit is zero.

If the "F" format specifier is used without other format specifiers, it is interpreted as the "F" standard date and time format specifier. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

The number of "F" format specifiers used with the ParseExact, TryParseExact, ParseExact, or TryParseExact method indicates the maximum number of most significant digits of the seconds fraction that can be present to successfully parse the string.

The following example includes the "F" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

```
Dim date1 As New Date(2008, 8, 29, 19, 27, 15, 018)
Dim ci As CultureInfo = CultureInfo.InvariantCulture

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci))
' Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci))
' Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci))
' Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci))
' Displays 07:27:15.018
```

Back to table

The "FF" custom format specifier

The "FF" custom format specifier represents the two most significant digits of the seconds fraction; that is, it represents the hundredths of a second in a date and time value. However, trailing zeros or two zero digits are not displayed.

The following example includes the "FF" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

```
Dim date1 As New Date(2008, 8, 29, 19, 27, 15, 018)
Dim ci As CultureInfo = CultureInfo.InvariantCulture

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci))
' Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci))
' Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci))
' Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci))
' Displays 07:27:15.018
```

Back to table

The "FFF" custom format specifier

The "FFF" custom format specifier represents the three most significant digits of the seconds fraction; that is, it represents the milliseconds in a date and time value. However, trailing zeros or three zero digits are not displayed.

The following example includes the "FFF" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

```
Dim date1 As New Date(2008, 8, 29, 19, 27, 15, 018)
Dim ci As CultureInfo = CultureInfo.InvariantCulture

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci))
' Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci))
' Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci))
' Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci))
' Displays 07:27:15.018
```

Back to table

The "FFFF" custom format specifier

The "FFFF" custom format specifier represents the four most significant digits of the seconds fraction; that is, it represents the ten thousandths of a second in a date and time value. However, trailing zeros or four zero digits are not displayed.

Although it is possible to display the ten thousandths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

Back to table

The "FFFFF" custom format specifier

The "FFFFF" custom format specifier represents the five most significant digits of the seconds fraction; that is, it represents the hundred thousandths of a second in a date and time value. However, trailing zeros or five zero digits are not displayed.

Although it is possible to display the hundred thousandths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

Back to table

The "FFFFFF" custom format specifier

The "FFFFFF" custom format specifier represents the six most significant digits of the seconds fraction; that is, it represents the millionths of a second in a date and time value. However, trailing zeros or six zero digits are not displayed.

Although it is possible to display the millionths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On tfhe Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

Back to table

The "FFFFFFF" custom format specifier

The "FFFFFFF" custom format specifier represents the seven most significant digits of the seconds fraction; that is, it represents the ten millionths of a second in a date and time value. However, trailing zeros or seven zero digits are not displayed.

Although it is possible to display the ten millionths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

Back to table

The "g" or "gg" custom format specifier

The "g" or "gg" custom format specifiers (plus any number of additional "g" specifiers) represents the period or era, such as A.D. The formatting operation ignores this specifier if the date to be formatted does not have an associated period or era string.

If the "g" format specifier is used without other custom format specifiers, it is interpreted as the "g" standard date and time format specifier. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

The following example includes the "g" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(70, 08, 04);

Console.WriteLine(date1.ToString("MM/dd/yyyy g",
                  CultureInfo.InvariantCulture));
// Displays 08/04/0070 A.D.
Console.WriteLine(date1.ToString("MM/dd/yyyy g",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 08/04/0070 ap. J.-C.
```

```
Dim date1 As Date = #08/04/0070#

Console.WriteLine(date1.ToString("MM/dd/yyyy g", _
                  CultureInfo.InvariantCulture))
' Displays 08/04/0070 A.D.
Console.WriteLine(date1.ToString("MM/dd/yyyy g", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays 08/04/0070 ap. J.-C.
```

Back to table

The "h" custom format specifier

The "h" custom format specifier represents the hour as a number from 1 through 12; that is, the hour is represented by a 12-hour clock that counts the whole hours since midnight or noon. A particular hour after midnight is indistinguishable from the same hour after noon. The hour is not rounded, and a single-digit hour is formatted without a leading zero. For example, given a time of 5:43 in the morning or afternoon, this custom format specifier displays "5".

If the "h" format specifier is used without other custom format specifiers, it is interpreted as a standard date and time format specifier and throws a FormatException. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

The following example includes the "h" custom format specifier in a custom format string.

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 µ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 µ
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1 µ
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1.5 µ
```

Back to table

The "hh" custom format specifier

The "hh" custom format specifier (plus any number of additional "h" specifiers) represents the hour as a number from 01 through 12; that is, the hour is represented by a 12-hour clock that counts the whole hours since midnight or noon. A particular hour after midnight is indistinguishable from the same hour after noon. The hour is not rounded, and a single-digit hour is formatted with a leading zero. For example, given a time of 5:43 in the morning or afternoon, this format specifier displays "05".

The following example includes the "hh" custom format specifier in a custom format string.

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01 du.
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01.50 du.
```

Back to table

The "H" custom format specifier

The "H" custom format specifier represents the hour as a number from 0 through 23; that is, the hour is represented by a zero-based 24-hour clock that counts the hours since midnight. A single-digit hour is formatted without a leading zero.

If the "H" format specifier is used without other custom format specifiers, it is interpreted as a standard date and time format specifier and throws a FormatException. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

The following example includes the "H" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 1, 1, 6, 9, 1);
Console.WriteLine(date1.ToString("H:mm:ss",
                  CultureInfo.InvariantCulture));
// Displays 6:09:01
```

```
Dim date1 As Date = #6:09:01AM#
Console.WriteLine(date1.ToString("H:mm:ss", _
                  CultureInfo.InvariantCulture))
' Displays 6:09:01
```

Back to table

The "HH" custom format specifier

The "HH" custom format specifier (plus any number of additional "H" specifiers) represents the hour as a number from 00 through 23; that is, the hour is represented by a zero-based 24-hour clock that counts the hours since midnight. A single-digit hour is formatted with a leading zero.

The following example includes the "HH" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 1, 1, 6, 9, 1);
Console.WriteLine(date1.ToString("HH:mm:ss",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01
```

```
Dim date1 As Date = #6:09:01AM#
Console.WriteLine(date1.ToString("HH:mm:ss", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01
```

Back to table

The "K" custom format specifier

The "K" custom format specifier represents the time zone information of a date and time value. When this format specifier is used with DateTime values, the result string is defined by the value of the DateTime.Kind property:

- For the local time zone (a DateTime.Kind property value of DateTimeKind.Local), this specifier is equivalent to the "zzz" specifier and produces a result string containing the local offset from Coordinated Universal Time (UTC); for example, "-07:00".
- For a UTC time (a DateTime.Kind property value of DateTimeKind.Utc), the result string includes a "Z" character to represent a UTC date.
- For a time from an unspecified time zone (a time whose DateTime.Kind property equals DateTimeKind.Unspecified), the result is equivalent to String.Empty.

For DateTimeOffset values, the "K" format specifier is equivalent to the "zzz" format specifier, and produces a result string containing the DateTimeOffset value's offset from UTC.

If the "K" format specifier is used without other custom format specifiers, it is interpreted as a standard date and time format specifier and throws a FormatException. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

The following example displays the string that results from using the "K" custom format specifier with various DateTime and DateTimeOffset values on a system in the U.S. Pacific Time zone.

```
Console.WriteLine(DateTime.Now.ToString("%K"));
// Displays -07:00
Console.WriteLine(DateTime.UtcNow.ToString("%K"));
// Displays Z
Console.WriteLine("'{0}'",
                  DateTime.SpecifyKind(DateTime.Now,
                       DateTimeKind.Unspecified).ToString("%K"));
// Displays ''
Console.WriteLine(DateTimeOffset.Now.ToString("%K"));
// Displays -07:00
Console.WriteLine(DateTimeOffset.UtcNow.ToString("%K"));
// Displays +00:00
Console.WriteLine(new DateTimeOffset(2008, 5, 1, 6, 30, 0,
                      new TimeSpan(5, 0, 0)).ToString("%K"));
// Displays +05:00
```

```
Console.WriteLine(Date.Now.ToString("%K"))
' Displays -07:00
Console.WriteLine(Date.UtcNow.ToString("%K"))
' Displays Z
Console.WriteLine("'{0}'", _
                  Date.SpecifyKind(Date.Now, _
                                   DateTimeKind.Unspecified). _
                  ToString("%K"))
' Displays ''
Console.WriteLine(DateTimeOffset.Now.ToString("%K"))
' Displays -07:00
Console.WriteLine(DateTimeOffset.UtcNow.ToString("%K"))
' Displays +00:00
Console.WriteLine(New DateTimeOffset(2008, 5, 1, 6, 30, 0, _
                                     New TimeSpan(5, 0, 0)). _
                  ToString("%K"))
' Displays +05:00
```

Back to table

The "m" custom format specifier

The "m" custom format specifier represents the minute as a number from 0 through 59. The minute represents whole minutes that have passed since the last hour. A single-digit minute is formatted without a leading zero.

If the "m" format specifier is used without other custom format specifiers, it is interpreted as the "m" standard date and time format specifier. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

The following example includes the "m" custom format specifier in a custom format string.

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 µ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 µ
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1 µ
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1.5 µ
```

Back to table

The "mm" custom format specifier

The "mm" custom format specifier (plus any number of additional "m" specifiers) represents the minute as a number from 00 through 59. The minute represents whole minutes that have passed since the last hour. A single-digit minute is formatted with a leading zero.

The following example includes the "mm" custom format specifier in a custom format string.

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01 du.
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01.50 du.
```

Back to table

The "M" custom format specifier

The "M" custom format specifier represents the month as a number from 1 through 12 (or from 1 through 13 for calendars that have 13 months). A single-digit month is formatted without a leading zero.

If the "M" format specifier is used without other custom format specifiers, it is interpreted as the "M" standard date and time format specifier. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

The following example includes the "M" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 8, 18);
Console.WriteLine(date1.ToString("(M) MMM, MMMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays (8) Aug, August
Console.WriteLine(date1.ToString("(M) MMM, MMMM",
                  CultureInfo.CreateSpecificCulture("nl-NL")));
// Displays (8) aug, augustus
Console.WriteLine(date1.ToString("(M) MMM, MMMM",
                  CultureInfo.CreateSpecificCulture("lv-LV")));
// Displays (8) Aug, augusts
```

```
Dim date1 As Date = #8/18/2008#
Console.WriteLine(date1.ToString("(M) MMM, MMMM", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays (8) Aug, August
Console.WriteLine(date1.ToString("(M) MMM, MMMM", _
                  CultureInfo.CreateSpecificCulture("nl-NL")))
' Displays (8) aug, augustus
Console.WriteLine(date1.ToString("(M) MMM, MMMM", _
                  CultureInfo.CreateSpecificCulture("lv-LV")))
' Displays (8) Aug, augusts
```

Back to table

The "MM" custom format specifier

The "MM" custom format specifier represents the month as a number from 01 through 12 (or from 1 through 13 for calendars that have 13 months). A single-digit month is formatted with a leading zero.

The following example includes the "MM" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 1, 2, 6, 30, 15);

Console.WriteLine(date1.ToString("dd, MM",
                  CultureInfo.InvariantCulture));
// 02, 01
```

```
Dim date1 As Date = #1/2/2008 6:30:15AM#

Console.WriteLine(date1.ToString("dd, MM", _
                  CultureInfo.InvariantCulture))
' 02, 01
```

Back to table

The "MMM" custom format specifier

The "MMM" custom format specifier represents the abbreviated name of the month. The localized abbreviated name of the month is retrieved from the DateTimeFormatInfo.AbbreviatedMonthNames property of the current or specified culture.

The following example includes the "MMM" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("ddd d MMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Fri 29 Aug
Console.WriteLine(date1.ToString("ddd d MMM",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays ven. 29 août
```

```
Dim date1 As Date = #08/29/2008 7:27:15PM#

Console.WriteLine(date1.ToString("ddd d MMM", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Fri 29 Aug
Console.WriteLine(date1.ToString("ddd d MMM", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays ven. 29 août
```

Back to table

The "MMMM" custom format specifier

The "MMMM" custom format specifier represents the full name of the month. The localized name of the month is retrieved from the DateTimeFormatInfo.MonthNames property of the current or specified culture.

The following example includes the "MMMM" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("dddd dd MMMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Friday 29 August
Console.WriteLine(date1.ToString("dddd dd MMMM",
                  CultureInfo.CreateSpecificCulture("it-IT")));
// Displays venerdì 29 agosto
```

```
Dim date1 As Date = #08/29/2008 7:27:15PM#

Console.WriteLine(date1.ToString("dddd dd MMMM", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Friday 29 August
Console.WriteLine(date1.ToString("dddd dd MMMM", _
                  CultureInfo.CreateSpecificCulture("it-IT")))
' Displays venerdì 29 agosto
```

Back to table

The "s" custom format specifier

The "s" custom format specifier represents the seconds as a number from 0 through 59. The result represents whole seconds that have passed since the last minute. A single-digit second is formatted without a leading zero.

If the "s" format specifier is used without other custom format specifiers, it is interpreted as the "s" standard date and time format specifier. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

The following example includes the "s" custom format specifier in a custom format string.

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 µ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 µ
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1 µ
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1.5 µ
```

Back to table

The "ss" custom format specifier

The "ss" custom format specifier (plus any number of additional "s" specifiers) represents the seconds as a number from 00 through 59. The result represents whole seconds that have passed since the last minute. A single-digit second is formatted with a leading zero.

The following example includes the "ss" custom format specifier in a custom format string.

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01 du.
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01.50 du.
```

Back to table

The "t" custom format specifier

The "t" custom format specifier represents the first character of the AM/PM designator. The appropriate localized designator is retrieved from the DateTimeFormatInfo.AMDesignator or DateTimeFormatInfo.PMDesignator property of the current or specific culture. The AM designator is used for all times from 0:00:00 (midnight) to 11:59:59.999. The PM designator is used for all times from 12:00:00 (noon) to 23:59:59.999.

If the "t" format specifier is used without other custom format specifiers, it is interpreted as the "t" standard date and time format specifier. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

The following example includes the "t" custom format specifier in a custom format string.

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 μ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 μ
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1 μ
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1.5 μ
```

Back to table

The "tt" custom format specifier

The "tt" custom format specifier (plus any number of additional "t" specifiers) represents the entire AM/PM designator. The appropriate localized designator is retrieved from the DateTimeFormatInfo.AMDesignator or DateTimeFormatInfo.PMDesignator property of the current or specific culture. The AM designator is used for all times from 0:00:00 (midnight) to 11:59:59.999. The PM designator is used for all times from 12:00:00 (noon) to 23:59:59.999.

Make sure to use the "tt" specifier for languages for which it is necessary to maintain the distinction between AM and PM. An example is Japanese, for which the AM and PM designators differ in the second character instead of the first character.

The following example includes the "tt" custom format specifier in a custom format string.

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01 du.
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01.50 du.
```

Back to table

The "y" custom format specifier

The "y" custom format specifier represents the year as a one-digit or two-digit number. If the year has more than two digits, only the two low-order digits appear in the result. If the first digit of a two-digit year begins with a zero (for example, 2008), the number is formatted without a leading zero.

If the "y" format specifier is used without other custom format specifiers, it is interpreted as the "y" standard date and time format specifier. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

The following example includes the "y" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

```
Dim date1 As Date = #12/1/0001#
Dim date2 As Date = #1/1/2010#
Console.WriteLine(date1.ToString("%y"))
' Displays 1
Console.WriteLine(date1.ToString("yy"))
' Displays 01
Console.WriteLine(date1.ToString("yyy"))
' Displays 001
Console.WriteLine(date1.ToString("yyyy"))
' Displays 0001
Console.WriteLine(date1.ToString("yyyyy"))
' Displays 00001
Console.WriteLine(date2.ToString("%y"))
' Displays 10
Console.WriteLine(date2.ToString("yy"))
' Displays 10
Console.WriteLine(date2.ToString("yyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyyy"))
' Displays 02010
```

Back to table

The "yy" custom format specifier

The "yy" custom format specifier represents the year as a two-digit number. If the year has more than two digits, only the two low-order digits appear in the result. If the two-digit year has fewer than two significant digits, the number is padded with leading zeros to produce two digits.

In a parsing operation, a two-digit year that is parsed using the "yy" custom format specifier is interpreted based on the Calendar.TwoDigitYearMax property of the format provider's current calendar. The following example parses the string representation of a date that has a two-digit year by using the default Gregorian calendar of the en-US culture, which, in this case, is the current culture. It then changes the current culture's CultureInfo object to use a GregorianCalendar object whose TwoDigitYearMax property has been modified.

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string fmt = "dd-MMM-yy";
      string value = "24-Jan-49";

      Calendar cal = (Calendar) CultureInfo.CurrentCulture.Calendar.Clone();
      Console.WriteLine("Two Digit Year Range: {0} - {1}",
                        cal.TwoDigitYearMax - 99, cal.TwoDigitYearMax);

      Console.WriteLine("{0:d}", DateTime.ParseExact(value, fmt, null));
      Console.WriteLine();

      cal.TwoDigitYearMax = 2099;
      CultureInfo culture = (CultureInfo) CultureInfo.CurrentCulture.Clone();
      culture.DateTimeFormat.Calendar = cal;
      Thread.CurrentThread.CurrentCulture = culture;

      Console.WriteLine("Two Digit Year Range: {0} - {1}",
                        cal.TwoDigitYearMax - 99, cal.TwoDigitYearMax);
      Console.WriteLine("{0:d}", DateTime.ParseExact(value, fmt, null));
   }
}
// The example displays the following output:
//       Two Digit Year Range: 1930 - 2029
//       1/24/1949
//
//       Two Digit Year Range: 2000 - 2099
//       1/24/2049
```

```
Imports System.Globalization
Imports System.Threading

Module Example
   Public Sub Main()
      Dim fmt As String = "dd-MMM-yy"
      Dim value As String = "24-Jan-49"

      Dim cal As Calendar = CType(CultureInfo.CurrentCulture.Calendar.Clone(), Calendar)
      Console.WriteLine("Two Digit Year Range: {0} - {1}",
                        cal.TwoDigitYearMax - 99, cal.TwoDigitYearMax)

      Console.WriteLine("{0:d}", DateTime.ParseExact(value, fmt, Nothing))
      Console.WriteLine()

      cal.TwoDigitYearMax = 2099
      Dim culture As CultureInfo = CType(CultureInfo.CurrentCulture.Clone(), CultureInfo)
      culture.DateTimeFormat.Calendar = cal
      Thread.CurrentThread.CurrentCulture = culture

      Console.WriteLine("Two Digit Year Range: {0} - {1}",
                        cal.TwoDigitYearMax - 99, cal.TwoDigitYearMax)
      Console.WriteLine("{0:d}", DateTime.ParseExact(value, fmt, Nothing))
   End Sub
End Module
' The example displays the following output:
'       Two Digit Year Range: 1930 - 2029
'       1/24/1949
'
'       Two Digit Year Range: 2000 - 2099
'       1/24/2049
```

The following example includes the "yy" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

```
Dim date1 As Date = #12/1/0001#
Dim date2 As Date = #1/1/2010#
Console.WriteLine(date1.ToString("%y"))
' Displays 1
Console.WriteLine(date1.ToString("yy"))
' Displays 01
Console.WriteLine(date1.ToString("yyy"))
' Displays 001
Console.WriteLine(date1.ToString("yyyy"))
' Displays 0001
Console.WriteLine(date1.ToString("yyyyy"))
' Displays 00001
Console.WriteLine(date2.ToString("%y"))
' Displays 10
Console.WriteLine(date2.ToString("yy"))
' Displays 10
Console.WriteLine(date2.ToString("yyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyyy"))
' Displays 02010
```

Back to table

The "yyy" custom format specifier

The "yyy" custom format specifier represents the year with a minimum of three digits. If the year has more than three significant digits, they are included in the result string. If the year has fewer than three digits, the number is padded with leading zeros to produce three digits.

NOTE

For the Thai Buddhist calendar, which can have five-digit years, this format specifier displays all significant digits.

The following example includes the "yyy" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

```
Dim date1 As Date = #12/1/0001#
Dim date2 As Date = #1/1/2010#
Console.WriteLine(date1.ToString("%y"))
' Displays 1
Console.WriteLine(date1.ToString("yy"))
' Displays 01
Console.WriteLine(date1.ToString("yyy"))
' Displays 001
Console.WriteLine(date1.ToString("yyyy"))
' Displays 0001
Console.WriteLine(date1.ToString("yyyyy"))
' Displays 00001
Console.WriteLine(date2.ToString("%y"))
' Displays 10
Console.WriteLine(date2.ToString("yy"))
' Displays 10
Console.WriteLine(date2.ToString("yyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyyy"))
' Displays 02010
```

Back to table

The "yyyy" custom format specifier

The "yyyy" custom format specifier represents the year with a minimum of four digits. If the year has more than four significant digits, they are included in the result string. If the year has fewer than four digits, the number is padded with leading zeros to produce four digits.

NOTE

For the Thai Buddhist calendar, which can have five-digit years, this format specifier displays a minimum of four digits.

The following example includes the "yyyy" custom format specifier in a custom format string.

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

```vb
Dim date1 As Date = #12/1/0001#
Dim date2 As Date = #1/1/2010#
Console.WriteLine(date1.ToString("%y"))
' Displays 1
Console.WriteLine(date1.ToString("yy"))
' Displays 01
Console.WriteLine(date1.ToString("yyy"))
' Displays 001
Console.WriteLine(date1.ToString("yyyy"))
' Displays 0001
Console.WriteLine(date1.ToString("yyyyy"))
' Displays 00001
Console.WriteLine(date2.ToString("%y"))
' Displays 10
Console.WriteLine(date2.ToString("yy"))
' Displays 10
Console.WriteLine(date2.ToString("yyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyyy"))
' Displays 02010
```

Back to table

The "yyyyy" custom format specifier

The "yyyyy" custom format specifier (plus any number of additional "y" specifiers) represents the year with a minimum of five digits. If the year has more than five significant digits, they are included in the result string. If the year has fewer than five digits, the number is padded with leading zeros to produce five digits.

If there are additional "y" specifiers, the number is padded with as many leading zeros as necessary to produce the number of "y" specifiers.

The following example includes the "yyyyy" custom format specifier in a custom format string.

```csharp
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

```
Dim date1 As Date = #12/1/0001#
Dim date2 As Date = #1/1/2010#
Console.WriteLine(date1.ToString("%y"))
' Displays 1
Console.WriteLine(date1.ToString("yy"))
' Displays 01
Console.WriteLine(date1.ToString("yyy"))
' Displays 001
Console.WriteLine(date1.ToString("yyyy"))
' Displays 0001
Console.WriteLine(date1.ToString("yyyyy"))
' Displays 00001
Console.WriteLine(date2.ToString("%y"))
' Displays 10
Console.WriteLine(date2.ToString("yy"))
' Displays 10
Console.WriteLine(date2.ToString("yyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyyy"))
' Displays 02010
```

Back to table

The "z" custom format specifier

With DateTime values, the "z" custom format specifier represents the signed offset of the local operating system's time zone from Coordinated Universal Time (UTC), measured in hours. It does not reflect the value of an instance's DateTime.Kind property. For this reason, the "z" format specifier is not recommended for use with DateTime values.

With DateTimeOffset values, this format specifier represents the DateTimeOffset value's offset from UTC in hours.

The offset is always displayed with a leading sign. A plus sign (+) indicates hours ahead of UTC, and a minus sign (-) indicates hours behind UTC. A single-digit offset is formatted without a leading zero.

If the "z" format specifier is used without other custom format specifiers, it is interpreted as a standard date and time format specifier and throws a FormatException. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

The following example includes the "z" custom format specifier in a custom format string.

```
DateTime date1 = DateTime.UtcNow;
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date1));
// Displays -7, -07, -07:00

DateTimeOffset date2 = new DateTimeOffset(2008, 8, 1, 0, 0, 0,
                                          new TimeSpan(6, 0, 0));
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date2));
// Displays +6, +06, +06:00
```

```
Dim date1 As Date = Date.UtcNow
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}", _
                  date1))
' Displays -7, -07, -07:00

Dim date2 As New DateTimeOffset(2008, 8, 1, 0, 0, 0, _
                                New Timespan(6, 0, 0))
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}", _
                  date2))
' Displays +6, +06, +06:00
```

Back to table

The "zz" custom format specifier

With DateTime values, the "zz" custom format specifier represents the signed offset of the local operating system's time zone from UTC, measured in hours. It does not reflect the value of an instance's DateTime.Kind property. For this reason, the "zz" format specifier is not recommended for use with DateTime values.

With DateTimeOffset values, this format specifier represents the DateTimeOffset value's offset from UTC in hours.

The offset is always displayed with a leading sign. A plus sign (+) indicates hours ahead of UTC, and a minus sign (-) indicates hours behind UTC. A single-digit offset is formatted with a leading zero.

The following example includes the "zz" custom format specifier in a custom format string.

```
DateTime date1 = DateTime.UtcNow;
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date1));
// Displays -7, -07, -07:00

DateTimeOffset date2 = new DateTimeOffset(2008, 8, 1, 0, 0, 0,
                                          new TimeSpan(6, 0, 0));
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date2));
// Displays +6, +06, +06:00
```

```
Dim date1 As Date = Date.UtcNow
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}", _
                  date1))
' Displays -7, -07, -07:00

Dim date2 As New DateTimeOffset(2008, 8, 1, 0, 0, 0, _
                                New Timespan(6, 0, 0))
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}", _
                  date2))
' Displays +6, +06, +06:00
```

Back to table

The "zzz" custom format specifier

With DateTime values, the "zzz" custom format specifier represents the signed offset of the local operating system's time zone from UTC, measured in hours and minutes. It does not reflect the value of an instance's DateTime.Kind property. For this reason, the "zzz" format specifier is not recommended for use with DateTime values.

With DateTimeOffset values, this format specifier represents the DateTimeOffset value's offset from UTC in hours and minutes.

The offset is always displayed with a leading sign. A plus sign (+) indicates hours ahead of UTC, and a minus sign (-) indicates hours behind UTC. A single-digit offset is formatted with a leading zero.

The following example includes the "zzz" custom format specifier in a custom format string.

```
DateTime date1 = DateTime.UtcNow;
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date1));
// Displays -7, -07, -07:00

DateTimeOffset date2 = new DateTimeOffset(2008, 8, 1, 0, 0, 0,
                                          new TimeSpan(6, 0, 0));
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date2));
// Displays +6, +06, +06:00
```

```
Dim date1 As Date = Date.UtcNow
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}", _
                  date1))
' Displays -7, -07, -07:00

Dim date2 As New DateTimeOffset(2008, 8, 1, 0, 0, 0, _
                                New Timespan(6, 0, 0))
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}", _
                  date2))
' Displays +6, +06, +06:00
```

Back to table

The ":" custom format specifier

The ":" custom format specifier represents the time separator, which is used to differentiate hours, minutes, and seconds. The appropriate localized time separator is retrieved from the DateTimeFormatInfo.TimeSeparator property of the current or specified culture.

NOTE

To change the time separator for a particular date and time string, specify the separator character within a literal string delimiter. For example, the custom format string `hh'_'dd'_'ss` produces a result string in which "_" (an underscore) is always used as the time separator. To change the time separator for all dates for a culture, either change the value of the DateTimeFormatInfo.TimeSeparator property of the current culture, or instantiate a DateTimeFormatInfo object, assign the character to its TimeSeparator property, and call an overload of the formatting method that includes an IFormatProvider parameter.

If the ":" format specifier is used without other custom format specifiers, it is interpreted as a standard date and time format specifier and throws a FormatException. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

Back to table

The "/" custom format specifier

The "/" custom format specifier represents the date separator, which is used to differentiate years, months, and days. The appropriate localized date separator is retrieved from the DateTimeFormatInfo.DateSeparator property of the current or specified culture.

> **NOTE**
>
> To change the date separator for a particular date and time string, specify the separator character within a literal string delimiter. For example, the custom format string `mm'/'dd'/'yyyy` produces a result string in which "/" is always used as the date separator. To change the date separator for all dates for a culture, either change the value of the DateTimeFormatInfo.DateSeparator property of the current culture, or instantiate a DateTimeFormatInfo object, assign the character to its DateSeparator property, and call an overload of the formatting method that includes an IFormatProvider parameter.

If the "/" format specifier is used without other custom format specifiers, it is interpreted as a standard date and time format specifier and throws a FormatException. For more information about using a single format specifier, see Using Single Custom Format Specifiers later in this topic.

Back to table

Character literals

The following characters in a custom date and time format string are reserved and are always interpreted as formatting characters or, in the case of ", ', /, and \, as special characters.

F	H	K	M	d
f	g	h	m	s
t	y	z	%	:
/	"	'	|	

All other characters are always interpreted as character literals and, in a formatting operation, are included in the result string unchanged. In a parsing operation, they must match the characters in the input string exactly; the comparison is case-sensitive.

The following example includes the literal characters "PST" (for Pacific Standard Time) and "PDT" (for Pacific Daylight Time) to represent the local time zone in a format string. Note that the string is included in the result string, and that a string that includes the local time zone string also parses successfully.

```csharp
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      String[] formats = { "dd MMM yyyy hh:mm tt PST",
                           "dd MMM yyyy hh:mm tt PDT" };
      var dat = new DateTime(2016, 8, 18, 16, 50, 0);
      // Display the result string.
      Console.WriteLine(dat.ToString(formats[1]));

      // Parse a string.
      String value = "25 Dec 2016 12:00 pm PST";
      DateTime newDate;
      if (DateTime.TryParseExact(value, formats, null,
                                 DateTimeStyles.None, out newDate))
         Console.WriteLine(newDate);
      else
         Console.WriteLine("Unable to parse '{0}'", value);
   }
}
// The example displays the following output:
//       18 Aug 2016 04:50 PM PDT
//       12/25/2016 12:00:00 PM
```

```vb
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim formats() As String = { "dd MMM yyyy hh:mm tt PST",
                                  "dd MMM yyyy hh:mm tt PDT" }
      Dim dat As New Date(2016, 8, 18, 16, 50, 0)
      ' Display the result string.
      Console.WriteLine(dat.ToString(formats(1)))

      ' Parse a string.
      Dim value As String = "25 Dec 2016 12:00 pm PST"
      Dim newDate As Date
      If Date.TryParseExact(value, formats, Nothing,
                            DateTimeStyles.None, newDate) Then
         Console.WriteLine(newDate)
      Else
         Console.WriteLine("Unable to parse '{0}'", value)
      End If
   End Sub
End Module
' The example displays the following output:
'       18 Aug 2016 04:50 PM PDT
'       12/25/2016 12:00:00 PM
```

There are two ways to indicate that characters are to be interpreted as literal characters and not as reserve characters, so that they can be included in a result string or successfully parsed in an input string:

- By escaping each reserved character. For more information, see Using the Escape Character.

  The following example includes the literal characters "pst" (for Pacific Standard time) to represent the local time zone in a format string. Because both "s" and "t" are custom format strings, both characters must be escaped to be interpreted as character literals.

```csharp
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      String format = "dd MMM yyyy hh:mm tt p\\s\\t";
      var dat = new DateTime(2016, 8, 18, 16, 50, 0);
      // Display the result string.
      Console.WriteLine(dat.ToString(format));

      // Parse a string.
      String value = "25 Dec 2016 12:00 pm pst";
      DateTime newDate;
      if (DateTime.TryParseExact(value, format, null,
                                 DateTimeStyles.None, out newDate))
         Console.WriteLine(newDate);
      else
         Console.WriteLine("Unable to parse '{0}'", value);
   }
}
// The example displays the following output:
//       18 Aug 2016 04:50 PM PDT
//       12/25/2016 12:00:00 PM
```

```vb
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim fmt As String = "dd MMM yyyy hh:mm tt p\s\t"
      Dim dat As New Date(2016, 8, 18, 16, 50, 0)
      ' Display the result string.
      Console.WriteLine(dat.ToString(fmt))

      ' Parse a string.
      Dim value As String = "25 Dec 2016 12:00 pm pst"
      Dim newDate As Date
      If Date.TryParseExact(value, fmt, Nothing,
                            DateTimeStyles.None, newDate) Then
         Console.WriteLine(newDate)
      Else
         Console.WriteLine("Unable to parse '{0}'", value)
      End If
   End Sub
End Module
' The example displays the following output:
'       18 Aug 2016 04:50 PM pst
'       12/25/2016 12:00:00 PM
```

- By enclosing the entire literal string in quotation marks or apostrophes. The following example is like the previous one, except that "pst" is enclosed in quotation marks to indicate that the entire delimited string should be interpreted as character literals.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      String format = "dd MMM yyyy hh:mm tt \"pst\"";
      var dat = new DateTime(2016, 8, 18, 16, 50, 0);
      // Display the result string.
      Console.WriteLine(dat.ToString(format));

      // Parse a string.
      String value = "25 Dec 2016 12:00 pm pst";
      DateTime newDate;
      if (DateTime.TryParseExact(value, format, null,
                                 DateTimeStyles.None, out newDate))
         Console.WriteLine(newDate);
      else
         Console.WriteLine("Unable to parse '{0}'", value);
   }
}
// The example displays the following output:
//       18 Aug 2016 04:50 PM PDT
//       12/25/2016 12:00:00 PM
```

```
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim fmt As String = "dd MMM yyyy hh:mm tt ""pst"""
      Dim dat As New Date(2016, 8, 18, 16, 50, 0)
      ' Display the result string.
      Console.WriteLine(dat.ToString(fmt))

      ' Parse a string.
      Dim value As String = "25 Dec 2016 12:00 pm pst"
      Dim newDate As Date
      If Date.TryParseExact(value, fmt, Nothing,
                            DateTimeStyles.None, newDate) Then
         Console.WriteLine(newDate)
      Else
         Console.WriteLine("Unable to parse '{0}'", value)
      End If
   End Sub
End Module
' The example displays the following output:
'       18 Aug 2016 04:50 PM pst
'       12/25/2016 12:00:00 PM
```

Notes

Using single custom format specifiers

A custom date and time format string consists of two or more characters. Date and time formatting methods interpret any single-character string as a standard date and time format string. If they do not recognize the character as a valid format specifier, they throw a FormatException. For example, a format string that consists only of the specifier "h" is interpreted as a standard date and time format string. However, in this particular case, an exception is thrown because there is no "h" standard date and timeformat specifier.

To use any of the custom date and time format specifiers as the only specifier in a format string (that is, to use the "d", "f", "F", "g", "h", "H", "K", "m", "M", "s", "t", "y", "z", ":", or "/" custom format specifier by itself), include a space before or after the specifier, or include a percent ("%") format specifier before the single custom date and time specifier.

For example, "`%h`" is interpreted as a custom date and time format string that displays the hour represented by the current date and time value. You can also use the " h" or "h " format string, although this includes a space in the result string along with the hour. The following example illustrates these three format strings.

```
DateTime dat1 = new DateTime(2009, 6, 15, 13, 45, 0);

Console.WriteLine("'{0:%h}'", dat1);
Console.WriteLine("'{0: h}'", dat1);
Console.WriteLine("'{0:h }'", dat1);
// The example displays the following output:
//       '1'
//       ' 1'
//       '1 '
```

```
Dim dat1 As Date = #6/15/2009 1:45PM#

Console.WriteLine("'{0:%h}'", dat1)
Console.WriteLine("'{0: h}'", dat1)
Console.WriteLine("'{0:h }'", dat1)
' The example displays the following output:
'       '1'
'       ' 1'
'       '1 '
```

Using the Escape Character

The "d", "f", "F", "g", "h", "H", "K", "m", "M", "s", "t", "y", "z", ":", or "/" characters in a format string are interpreted as custom format specifiers rather than as literal characters. To prevent a character from being interpreted as a format specifier, you can precede it with a backslash (\), which is the escape character. The escape character signifies that the following character is a character literal that should be included in the result string unchanged.

To include a backslash in a result string, you must escape it with another backslash (`\\`).

> **NOTE**
>
> Some compilers, such as the C++ and C# compilers, may also interpret a single backslash character as an escape character. To ensure that a string is interpreted correctly when formatting, you can use the verbatim string literal character (the @ character) before the string in C#, or add another backslash character before each backslash in C# and C++. The following C# example illustrates both approaches.

The following example uses the escape character to prevent the formatting operation from interpreting the "h" and "m" characters as format specifiers.

```
DateTime date = new DateTime(2009, 06, 15, 13, 45, 30, 90);
string fmt1 = "h \\h m \\m";
string fmt2 = @"h \h m \m";

Console.WriteLine("{0} ({1}) -> {2}", date, fmt1, date.ToString(fmt1));
Console.WriteLine("{0} ({1}) -> {2}", date, fmt2, date.ToString(fmt2));
// The example displays the following output:
//       6/15/2009 1:45:30 PM (h \h m \m) -> 1 h 45 m
//       6/15/2009 1:45:30 PM (h \h m \m) -> 1 h 45 m
```

```
Dim date1 As Date = #6/15/2009 13:45#
Dim fmt As String = "h \h m \m"

Console.WriteLine("{0} ({1}) -> {2}", date1, fmt, date1.ToString(fmt))
' The example displays the following output:
'       6/15/2009 1:45:00 PM (h \h m \m) -> 1 h 45 m
```

Control Panel settings

The **Regional and Language Options** settings in Control Panel influence the result string produced by a formatting operation that includes many of the custom date and time format specifiers. These settings are used to initialize the DateTimeFormatInfo object associated with the current thread culture, which provides values used to govern formatting. Computers that use different settings generate different result strings.

In addition, if you use the CultureInfo.CultureInfo(String) constructor to instantiate a new CultureInfo object that represents the same culture as the current system culture, any customizations established by the **Regional and Language Options** item in Control Panel will be applied to the new CultureInfo object. You can use the CultureInfo.CultureInfo(String, Boolean) constructor to create a CultureInfo object that does not reflect a system's customizations.

DateTimeFormatInfo properties

Formatting is influenced by properties of the current DateTimeFormatInfo object, which is provided implicitly by the current thread culture or explicitly by the IFormatProvider parameter of the method that invokes formatting. For the IFormatProvider parameter, you should specify a CultureInfo object, which represents a culture, or a DateTimeFormatInfo object.

The result string produced by many of the custom date and time format specifiers also depends on properties of the current DateTimeFormatInfo object. Your application can change the result produced by some custom date and time format specifiers by changing the corresponding DateTimeFormatInfo property. For example, the "ddd" format specifier adds an abbreviated weekday name found in the AbbreviatedDayNames string array to the result string. Similarly, the "MMMM" format specifier adds a full month name found in the MonthNames string array to the result string.

See also

- System.DateTime
- System.IFormatProvider
- Formatting Types
- Standard Date and Time Format Strings
- Sample: .NET Framework 4 Formatting Utility

Standard TimeSpan Format Strings

9/6/2018 • 9 minutes to read • Edit Online

A standard TimeSpan format string uses a single format specifier to define the text representation of a TimeSpan value that results from a formatting operation. Any format string that contains more than one character, including white space, is interpreted as a custom TimeSpan format string. For more information, see Custom TimeSpan Format Strings .

The string representations of TimeSpan values are produced by calls to the overloads of the TimeSpan.ToString method, as well as by methods that support composite formatting, such as String.Format. For more information, see Formatting Types and Composite Formatting. The following example illustrates the use of standard format strings in formatting operations.

```
using System;

public class Example
{
   public static void Main()
   {
      TimeSpan duration = new TimeSpan(1, 12, 23, 62);
      string output = "Time of Travel: " + duration.ToString("c");
      Console.WriteLine(output);

      Console.WriteLine("Time of Travel: {0:c}", duration);
   }
}
// The example displays the following output:
//       Time of Travel: 1.12:24:02
//       Time of Travel: 1.12:24:02
```

```
Module Example
   Public Sub Main()
      Dim duration As New TimeSpan(1, 12, 23, 62)
      Dim output As String = "Time of Travel: " + duration.ToString("c")
      Console.WriteLine(output)

      Console.WriteLine("Time of Travel: {0:c}", duration)
   End Sub
End Module
' The example displays the following output:
'       Time of Travel: 1.12:24:02
'       Time of Travel: 1.12:24:02
```

Standard TimeSpan format strings are also used by the TimeSpan.ParseExact and TimeSpan.TryParseExact methods to define the required format of input strings for parsing operations. (Parsing converts the string representation of a value to that value.) The following example illustrates the use of standard format strings in parsing operations.

```
using System;

public class Example
{
   public static void Main()
   {
      string value = "1.03:14:56.1667";
      TimeSpan interval;
      try {
         interval = TimeSpan.ParseExact(value, "c", null);
         Console.WriteLine("Converted '{0}' to {1}", value, interval);
      }
      catch (FormatException) {
         Console.WriteLine("{0}: Bad Format", value);
      }
      catch (OverflowException) {
         Console.WriteLine("{0}: Out of Range", value);
      }

      if (TimeSpan.TryParseExact(value, "c", null, out interval))
         Console.WriteLine("Converted '{0}' to {1}", value, interval);
      else
         Console.WriteLine("Unable to convert {0} to a time interval.",
                           value);
   }
}
// The example displays the following output:
//       Converted '1.03:14:56.1667' to 1.03:14:56.1667000
//       Converted '1.03:14:56.1667' to 1.03:14:56.1667000
```

```
Module Example
   Public Sub Main()
      Dim value As String = "1.03:14:56.1667"
      Dim interval As TimeSpan
      Try
         interval = TimeSpan.ParseExact(value, "c", Nothing)
         Console.WriteLine("Converted '{0}' to {1}", value, interval)
      Catch e As FormatException
         Console.WriteLine("{0}: Bad Format", value)
      Catch e As OverflowException
         Console.WriteLine("{0}: Out of Range", value)
      End Try

      If TimeSpan.TryParseExact(value, "c", Nothing, interval) Then
         Console.WriteLine("Converted '{0}' to {1}", value, interval)
      Else
         Console.WriteLine("Unable to convert {0} to a time interval.",
                           value)
      End If
   End Sub
End Module
' The example displays the following output:
'       Converted '1.03:14:56.1667' to 1.03:14:56.1667000
'       Converted '1.03:14:56.1667' to 1.03:14:56.1667000
```

The following table lists the standard time interval format specifiers.

FORMAT SPECIFIER	NAME	DESCRIPTION	EXAMPLES

FORMAT SPECIFIER	NAME	DESCRIPTION	EXAMPLES
"c"	Constant (invariant) format	This specifier is not culture-sensitive. It takes the form `[-][d’.’]hh’:’mm’:’ss[‘.’ffffff`. (The "t" and "T" format strings produce the same results.) More information: The Constant ("c") Format Specifier.	`TimeSpan.Zero` -> 00:00:00 `New TimeSpan(0, 0, 30, 0)` -> 00:30:00 `New TimeSpan(3, 17, 25, 30, 500)` -> 3.17:25:30.5000000
"g"	General short format	This specifier outputs only what is needed. It is culture-sensitive and takes the form `[-][d’:’]h’:’mm’:’ss[.FFFFFFF]`. More information: The General Short ("g") Format Specifier.	`New TimeSpan(1, 3, 16, 50, 500)` -> 1:3:16:50.5 (en-US) `New TimeSpan(1, 3, 16, 50, 500)` -> 1:3:16:50,5 (fr-FR) `New TimeSpan(1, 3, 16, 50, 599)` -> 1:3:16:50.599 (en-US) `New TimeSpan(1, 3, 16, 50, 599)` -> 1:3:16:50,599 (fr-FR)
"G"	General long format	This specifier always outputs days and seven fractional digits. It is culture-sensitive and takes the form `[-]d’:’hh’:’mm’:’ss.fffffff`. More information: The General Long ("G") Format Specifier.	`New TimeSpan(18, 30, 0)` -> 0:18:30:00.0000000 (en-US) `New TimeSpan(18, 30, 0)` -> 0:18:30:00,0000000 (fr-FR)

The Constant ("c") Format Specifier

The "c" format specifier returns the string representation of a TimeSpan value in the following form:

[-][*d*.]*hh*:*mm*:*ss*[.*fffffff*]

Elements in square brackets ([and]) are optional. The period (.) and colon (:) are literal symbols. The following table describes the remaining elements.

ELEMENT	DESCRIPTION
-	An optional negative sign, which indicates a negative time interval.
d	The optional number of days, with no leading zeros.

ELEMENT	DESCRIPTION
hh	The number of hours, which ranges from "00" to "23".
mm	The number of minutes, which ranges from "00" to "59".
ss	The number of seconds, which ranges from "0" to "59".
fffffff	The optional fractional portion of a second. Its value can range from "0000001" (one tick, or one ten-millionth of a second) to "9999999" (9,999,999 ten-millionths of a second, or one second less one tick).

Unlike the "g" and "G" format specifiers, the "c" format specifier is not culture-sensitive. It produces the string representation of a TimeSpan value that is invariant and that is common to all previous versions of the .NET Framework before the .NET Framework 4. "c" is the default TimeSpan format string; the TimeSpan.ToString() method formats a time interval value by using the "c" format string.

NOTE

TimeSpan also supports the "t" and "T" standard format strings, which are identical in behavior to the "c" standard format string.

The following example instantiates two TimeSpan objects, uses them to perform arithmetic operations, and displays the result. In each case, it uses composite formatting to display the TimeSpan value by using the "c" format specifier.

```
using System;

public class Example
{
   public static void Main()
   {
      TimeSpan interval1, interval2;
      interval1 = new TimeSpan(7, 45, 16);
      interval2 = new TimeSpan(18, 12, 38);

      Console.WriteLine("{0:c} - {1:c} = {2:c}", interval1,
                        interval2, interval1 - interval2);
      Console.WriteLine("{0:c} + {1:c} = {2:c}", interval1,
                        interval2, interval1 + interval2);

      interval1 = new TimeSpan(0, 0, 1, 14, 365);
      interval2 = TimeSpan.FromTicks(2143756);
      Console.WriteLine("{0:c} + {1:c} = {2:c}", interval1,
                        interval2, interval1 + interval2);
   }
}
// The example displays the following output:
//       07:45:16 - 18:12:38 = -10:27:22
//       07:45:16 + 18:12:38 = 1.01:57:54
//       00:01:14.3650000 + 00:00:00.2143756 = 00:01:14.5793756
```

```
Module Example
   Public Sub Main()
      Dim interval1, interval2 As TimeSpan
      interval1 = New TimeSpan(7, 45, 16)
      interval2 = New TimeSpan(18, 12, 38)

      Console.WriteLine("{0:c} - {1:c} = {2:c}", interval1,
                        interval2, interval1 - interval2)
      Console.WriteLine("{0:c} + {1:c} = {2:c}", interval1,
                        interval2, interval1 + interval2)

      interval1 = New TimeSpan(0, 0, 1, 14, 365)
      interval2 = TimeSpan.FromTicks(2143756)
      Console.WriteLine("{0:c} + {1:c} = {2:c}", interval1,
                        interval2, interval1 + interval2)
   End Sub
End Module
' The example displays the following output:
'       07:45:16 - 18:12:38 = -10:27:22
'       07:45:16 + 18:12:38 = 1.01:57:54
'       00:01:14.3650000 + 00:00:00.2143756 = 00:01:14.5793756
```

Back to table

The General Short ("g") Format Specifier

The "g" TimeSpan format specifier returns the string representation of a TimeSpan value in a compact form by including only the elements that are necessary. It has the following form:

[-][*d*:]*h*:*mm*:*ss*[.*FFFFFFF*]

Elements in square brackets ([and]) are optional. The colon (:) is a literal symbol. The following table describes the remaining elements.

ELEMENT	DESCRIPTION
-	An optional negative sign, which indicates a negative time interval.
d	The optional number of days, with no leading zeros.
h	The number of hours, which ranges from "0" to "23", with no leading zeros.
mm	The number of minutes, which ranges from "00" to "59"..
ss	The number of seconds, which ranges from "00" to "59"..
.	The fractional seconds separator. It is equivalent to the specified culture's NumberDecimalSeparator property without user overrides.
FFFFFFF	The fractional seconds. As few digits as possible are displayed.

Like the "G" format specifier, the "g" format specifier is localized. Its fractional seconds separator is based on either the current culture or a specified culture's NumberDecimalSeparator property.

The following example instantiates two TimeSpan objects, uses them to perform arithmetic operations, and displays the result. In each case, it uses composite formatting to display the TimeSpan value by using the "g" format specifier. In addition, it formats the TimeSpan value by using the formatting conventions of the current system culture (which, in this case, is English - United States or en-US) and the French - France (fr-FR) culture.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      TimeSpan interval1, interval2;
      interval1 = new TimeSpan(7, 45, 16);
      interval2 = new TimeSpan(18, 12, 38);

      Console.WriteLine("{0:g} - {1:g} = {2:g}", interval1,
                        interval2, interval1 - interval2);
      Console.WriteLine(String.Format(new CultureInfo("fr-FR"),
                        "{0:g} + {1:g} = {2:g}", interval1,
                        interval2, interval1 + interval2));

      interval1 = new TimeSpan(0, 0, 1, 14, 36);
      interval2 = TimeSpan.FromTicks(2143756);
      Console.WriteLine("{0:g} + {1:g} = {2:g}", interval1,
                        interval2, interval1 + interval2);
   }
}
// The example displays the following output:
//       7:45:16 - 18:12:38 = -10:27:22
//       7:45:16 + 18:12:38 = 1:1:57:54
//       0:01:14.036 + 0:00:00.2143756 = 0:01:14.2503756
```

```
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim interval1, interval2 As TimeSpan
      interval1 = New TimeSpan(7, 45, 16)
      interval2 = New TimeSpan(18, 12, 38)

      Console.WriteLine("{0:g} - {1:g} = {2:g}", interval1,
                        interval2, interval1 - interval2)
      Console.WriteLine(String.Format(New CultureInfo("fr-FR"),
                        "{0:g} + {1:g} = {2:g}", interval1,
                        interval2, interval1 + interval2))

      interval1 = New TimeSpan(0, 0, 1, 14, 36)
      interval2 = TimeSpan.FromTicks(2143756)
      Console.WriteLine("{0:g} + {1:g} = {2:g}", interval1,
                        interval2, interval1 + interval2)
   End Sub
End Module
' The example displays the following output:
'       7:45:16 - 18:12:38 = -10:27:22
'       7:45:16 + 18:12:38 = 1:1:57:54
'       0:01:14.036 + 0:00:00.2143756 = 0:01:14.2503756
```

Back to table

The General Long ("G") Format Specifier

The "G" TimeSpan format specifier returns the string representation of a TimeSpan value in a long form that always includes both days and fractional seconds. The string that results from the "G" standard format specifier has the following form:

[-]*d*:*hh*:*mm*:*ss*.*fffffff*

Elements in square brackets ([and]) are optional. The colon (:) is a literal symbol. The following table describes the remaining elements.

ELEMENT	DESCRIPTION
-	An optional negative sign, which indicates a negative time interval.
d	The number of days, with no leading zeros.
hh	The number of hours, which ranges from "00" to "23".
mm	The number of minutes, which ranges from "00" to "59".
ss	The number of seconds, which ranges from "00" to "59".
.	The fractional seconds separator. It is equivalent to the specified culture's NumberDecimalSeparator property without user overrides.
fffffff	The fractional seconds.

Like the "G" format specifier, the "g" format specifier is localized. Its fractional seconds separator is based on either the current culture or a specified culture's NumberDecimalSeparator property.

The following example instantiates two TimeSpan objects, uses them to perform arithmetic operations, and displays the result. In each case, it uses composite formatting to display the TimeSpan value by using the "G" format specifier. In addition, it formats the TimeSpan value by using the formatting conventions of the current system culture (which, in this case, is English - United States or en-US) and the French - France (fr-FR) culture.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      TimeSpan interval1, interval2;
      interval1 = new TimeSpan(7, 45, 16);
      interval2 = new TimeSpan(18, 12, 38);

      Console.WriteLine("{0:G} - {1:G} = {2:G}", interval1,
                        interval2, interval1 - interval2);
      Console.WriteLine(String.Format(new CultureInfo("fr-FR"),
                        "{0:G} + {1:G} = {2:G}", interval1,
                        interval2, interval1 + interval2));

      interval1 = new TimeSpan(0, 0, 1, 14, 36);
      interval2 = TimeSpan.FromTicks(2143756);
      Console.WriteLine("{0:G} + {1:G} = {2:G}", interval1,
                        interval2, interval1 + interval2);
   }
}
// The example displays the following output:
//       0:07:45:16.0000000 - 0:18:12:38.0000000 = -0:10:27:22.0000000
//       0:07:45:16,0000000 + 0:18:12:38,0000000 = 1:01:57:54,0000000
//       0:00:01:14.0360000 + 0:00:00:00.2143756 = 0:00:01:14.2503756
```

```
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim interval1, interval2 As TimeSpan
      interval1 = New TimeSpan(7, 45, 16)
      interval2 = New TimeSpan(18, 12, 38)

      Console.WriteLine("{0:G} - {1:G} = {2:G}", interval1,
                        interval2, interval1 - interval2)
      Console.WriteLine(String.Format(New CultureInfo("fr-FR"),
                        "{0:G} + {1:G} = {2:G}", interval1,
                        interval2, interval1 + interval2))

      interval1 = New TimeSpan(0, 0, 1, 14, 36)
      interval2 = TimeSpan.FromTicks(2143756)
      Console.WriteLine("{0:G} + {1:G} = {2:G}", interval1,
                        interval2, interval1 + interval2)
   End Sub
End Module
' The example displays the following output:
'       0:07:45:16.0000000 - 0:18:12:38.0000000 = -0:10:27:22.0000000
'       0:07:45:16,0000000 + 0:18:12:38,0000000 = 1:01:57:54,0000000
'       0:00:01:14.0360000 + 0:00:00:00.2143756 = 0:00:01:14.2503756
```

Back to table

See also

- Formatting Types
- Custom TimeSpan Format Strings
- Parsing Strings

Custom TimeSpan Format Strings

9/6/2018 • 43 minutes to read • Edit Online

A TimeSpan format string defines the string representation of a TimeSpan value that results from a formatting operation. A custom format string consists of one or more custom TimeSpan format specifiers along with any number of literal characters. Any string that is not a standard TimeSpan format string is interpreted as a custom TimeSpan format string.

> **IMPORTANT**
>
> The custom TimeSpan format specifiers do not include placeholder separator symbols, such as the symbols that separate days from hours, hours from minutes, or seconds from fractional seconds. Instead, these symbols must be included in the custom format string as string literals. For example, `"dd\.hh\:mm"` defines a period (.) as the separator between days and hours, and a colon (:) as the separator between hours and minutes.
>
> Custom TimeSpan format specifiers also do not include a sign symbol that enables you to differentiate between negative and positive time intervals. To include a sign symbol, you have to construct a format string by using conditional logic. The Other Characters section includes an example.

The string representations of TimeSpan values are produced by calls to the overloads of the TimeSpan.ToString method, and by methods that support composite formatting, such as String.Format. For more information, see Formatting Types and Composite Formatting. The following example illustrates the use of custom format strings in formatting operations.

```
using System;

public class Example
{
   public static void Main()
   {
      TimeSpan duration = new TimeSpan(1, 12, 23, 62);

      string output = null;
      output = "Time of Travel: " + duration.ToString("%d") + " days";
      Console.WriteLine(output);
      output = "Time of Travel: " + duration.ToString(@"dd\.hh\:mm\:ss");
      Console.WriteLine(output);

      Console.WriteLine("Time of Travel: {0:%d} day(s)", duration);
      Console.WriteLine("Time of Travel: {0:dd\\.hh\\:mm\\:ss} days", duration);
   }
}
// The example displays the following output:
//       Time of Travel: 1 days
//       Time of Travel: 01.12:24:02
//       Time of Travel: 1 day(s)
//       Time of Travel: 01.12:24:02 days
```

```
Module Example
   Public Sub Main()
      Dim duration As New TimeSpan(1, 12, 23, 62)

      Dim output As String = Nothing
      output = "Time of Travel: " + duration.ToString("%d") + " days"
      Console.WriteLine(output)
      output = "Time of Travel: " + duration.ToString("dd\.hh\:mm\:ss")
      Console.WriteLine(output)

      Console.WriteLine("Time of Travel: {0:%d} day(s)", duration)
      Console.WriteLine("Time of Travel: {0:dd\.hh\:mm\:ss} days", duration)
   End Sub
End Module
' The example displays the following output:
'       Time of Travel: 1 days
'       Time of Travel: 01.12:24:02
'       Time of Travel: 1 day(s)
'       Time of Travel: 01.12:24:02 days
```

Custom TimeSpan format strings are also used by the TimeSpan.ParseExact and TimeSpan.TryParseExact methods to define the required format of input strings for parsing operations. (Parsing converts the string representation of a value to that value.) The following example illustrates the use of standard format strings in parsing operations.

```
using System;

public class Example
{
   public static void Main()
   {
      string value = null;
      TimeSpan interval;

      value = "6";
      if (TimeSpan.TryParseExact(value, "%d", null, out interval))
         Console.WriteLine("{0} --> {1}", value, interval.ToString("c"));
      else
         Console.WriteLine("Unable to parse '{0}'", value);

      value = "16:32.05";
      if (TimeSpan.TryParseExact(value, @"mm\:ss\.ff", null, out interval))
         Console.WriteLine("{0} --> {1}", value, interval.ToString("c"));
      else
         Console.WriteLine("Unable to parse '{0}'", value);

      value= "12.035";
      if (TimeSpan.TryParseExact(value, "ss\\.fff", null, out interval))
         Console.WriteLine("{0} --> {1}", value, interval.ToString("c"));
      else
         Console.WriteLine("Unable to parse '{0}'", value);
   }
}
// The example displays the following output:
//       6 --> 6.00:00:00
//       16:32.05 --> 00:16:32.0500000
//       12.035 --> 00:00:12.0350000
```

```
Module Example
   Public Sub Main()
      Dim value As String = Nothing
      Dim interval As TimeSpan

      value = "6"
      If TimeSpan.TryParseExact(value, "%d", Nothing, interval) Then
         Console.WriteLine("{0} --> {1}", value, interval.ToString("c"))
      Else
         Console.WriteLine("Unable to parse '{0}'", value)
      End If

      value = "16:32.05"
      If TimeSpan.TryParseExact(value, "mm\:ss\.ff", Nothing, interval) Then
         Console.WriteLine("{0} --> {1}", value, interval.ToString("c"))
      Else
         Console.WriteLine("Unable to parse '{0}'", value)
      End If

      value= "12.035"
      If TimeSpan.TryParseExact(value, "ss\.fff", Nothing, interval) Then
         Console.WriteLine("{0} --> {1}", value, interval.ToString("c"))
      Else
         Console.WriteLine("Unable to parse '{0}'", value)
      End If
   End Sub
End Module
' The example displays the following output:
'       6 --> 6.00:00:00
'       16:32.05 --> 00:16:32.0500000
'       12.035 --> 00:00:12.0350000
```

The following table describes the custom date and time format specifiers.

FORMAT SPECIFIER	DESCRIPTION	EXAMPLE
"d", "%d"	The number of whole days in the time interval. More information: The "d" Custom Format Specifier.	`new TimeSpan(6, 14, 32, 17, 685):` `%d` --> "6" `d\.hh\:mm` --> "6.14:32"
"dd"-"dddddddd"	The number of whole days in the time interval, padded with leading zeros as needed. More information: The "dd"-"dddddddd" Custom Format Specifiers.	`new TimeSpan(6, 14, 32, 17, 685):` `ddd` --> "006" `dd\.hh\:mm` --> "06.14:32"
"h", "%h"	The number of whole hours in the time interval that are not counted as part of days. Single-digit hours do not have a leading zero. More information: The "h" Custom Format Specifier.	`new TimeSpan(6, 14, 32, 17, 685):` `%h` --> "14" `hh\:mm` --> "14:32"

FORMAT SPECIFIER	DESCRIPTION	EXAMPLE

"hh"	The number of whole hours in the time interval that are not counted as part of days. Single-digit hours have a leading zero.
	
	More information: The "hh" Custom Format Specifier.
		`new TimeSpan(6, 14, 32, 17, 685):`
		
		`hh` --> "14"
		
		`new TimeSpan(6, 8, 32, 17, 685):`
		
		`hh` --> 08

"m", "%m"	The number of whole minutes in the time interval that are not included as part of hours or days. Single-digit minutes do not have a leading zero. More information: The "m" Custom Format Specifier.	`new TimeSpan(6, 14, 8, 17, 685):` `%m` --> "8" `h\:m` --> "14:8"

"mm"	The number of whole minutes in the time interval that are not included as part of hours or days. Single-digit minutes have a leading zero.
	
	More information: The "mm" Custom Format Specifier.
		`new TimeSpan(6, 14, 8, 17, 685):`
		
		`mm` --> "08"
		
		`new TimeSpan(6, 8, 5, 17, 685):`
		
		`d\.hh\:mm\:ss` --> 6.08:05:17

"s", "%s"	The number of whole seconds in the time interval that are not included as part of hours, days, or minutes. Single-digit seconds do not have a leading zero. More information: The "s" Custom Format Specifier.	`TimeSpan.FromSeconds(12.965)`: `%s` --> 12 `s\.fff` --> 12.965
"ss"	The number of whole seconds in the time interval that are not included as part of hours, days, or minutes. Single-digit seconds have a leading zero. More information: The "ss" Custom Format Specifier.	`TimeSpan.FromSeconds(6.965)`: `ss` --> 06 `ss\.fff` --> 06.965
"f", "%f"	The tenths of a second in a time interval. More information: The "f" Custom Format Specifier.	`TimeSpan.FromSeconds(6.895)`: `f` --> 8 `ss\.f` --> 06.8
"ff"	The hundredths of a second in a time interval. More information:The "ff" Custom Format Specifier.	`TimeSpan.FromSeconds(6.895)`: `ff` --> 89 `ss\.ff` --> 06.89

FORMAT SPECIFIER	DESCRIPTION	EXAMPLE
"fff"	The milliseconds in a time interval. More information: The "fff" Custom Format Specifier.	`TimeSpan.FromSeconds(6.895)`: `fff` --> 895 `ss\.fff` --> 06.895
"ffff"	The ten-thousandths of a second in a time interval. More information: The "ffff" Custom Format Specifier.	`TimeSpan.Parse("0:0:6.8954321")`: `ffff` --> 8954 `ss\.ffff` --> 06.8954
"fffff"	The hundred-thousandths of a second in a time interval. More information: The "fffff" Custom Format Specifier.	`TimeSpan.Parse("0:0:6.8954321")`: `fffff` --> 89543 `ss\.fffff` --> 06.89543
"ffffff"	The millionths of a second in a time interval. More information: The "ffffff" Custom Format Specifier.	`TimeSpan.Parse("0:0:6.8954321")`: `ffffff` --> 895432 `ss\.ffffff` --> 06.895432
"fffffff"	The ten-millionths of a second (or the fractional ticks) in a time interval. More information: The "fffffff" Custom Format Specifier.	`TimeSpan.Parse("0:0:6.8954321")`: `fffffff` --> 8954321 `ss\.fffffff` --> 06.8954321
"F", "%F"	The tenths of a second in a time interval. Nothing is displayed if the digit is zero. More information: The "F" Custom Format Specifier.	`TimeSpan.Parse("00:00:06.32")`: `%F`: 3 `TimeSpan.Parse("0:0:3.091")`: `ss\.F`: 03.
"FF"	The hundredths of a second in a time interval. Any fractional trailing zeros or two zero digits are not included. More information: The "FF" Custom Format Specifier.	`TimeSpan.Parse("00:00:06.329")`: `FF`: 32 `TimeSpan.Parse("0:0:3.101")`: `ss\.FF`: 03.1
"FFF"	The milliseconds in a time interval. Any fractional trailing zeros are not included. More information:	`TimeSpan.Parse("00:00:06.3291")`: `FFF`: 329 `TimeSpan.Parse("0:0:3.1009")`: `ss\.FFF`: 03.1

FORMAT SPECIFIER	DESCRIPTION	EXAMPLE
"FFFF"	The ten-thousandths of a second in a time interval. Any fractional trailing zeros are not included. More information: The "FFFF" Custom Format Specifier.	`TimeSpan.Parse("00:00:06.32917")` : `FFFFF` : 3291 `TimeSpan.Parse("0:0:3.10009")` : `ss\.FFFF` : 03.1
"FFFFF"	The hundred-thousandths of a second in a time interval. Any fractional trailing zeros are not included. More information: The "FFFFF" Custom Format Specifier.	`TimeSpan.Parse("00:00:06.329179")` : `FFFFF` : 32917 `TimeSpan.Parse("0:0:3.100009")` : `ss\.FFFFF` : 03.1
"FFFFFF"	The millionths of a second in a time interval. Any fractional trailing zeros are not displayed. More information: The "FFFFFF" Custom Format Specifier.	`TimeSpan.Parse("00:00:06.3291791")` : `FFFFFF` : 329179 `TimeSpan.Parse("0:0:3.1000009")` : `ss\.FFFFFF` : 03.1
"FFFFFFF"	The ten-millions of a second in a time interval. Any fractional trailing zeros or seven zeros are not displayed. More information: The "FFFFFFF" Custom Format Specifier.	`TimeSpan.Parse("00:00:06.3291791")` : `FFFFFF` : 3291791 `TimeSpan.Parse("0:0:3.1900000")` : `ss\.FFFFFF` : 03.19
'*string*'	Literal string delimiter. More information: Other Characters.	`new TimeSpan(14, 32, 17):` `hh':'mm':'ss` --> "14:32:17"
|The escape character. More information:Other Characters.	`new TimeSpan(14, 32, 17):` `hh\:mm\:ss` --> "14:32:17"	
Any other character	Any other unescaped character is interpreted as a custom format specifier. More Information: Other Characters.	`new TimeSpan(14, 32, 17):` `hh\:mm\:ss` --> "14:32:17"

The "d" Custom Format Specifier

The "d" custom format specifier outputs the value of the TimeSpan.Days property, which represents the number of whole days in the time interval. It outputs the full number of days in a TimeSpan value, even if the value has more than one digit. If the value of the TimeSpan.Days property is zero, the specifier outputs "0".

If the "d" custom format specifier is used alone, specify "%d" so that it is not misinterpreted as a standard format string. The following example provides an illustration.

```
TimeSpan ts1 = new TimeSpan(16, 4, 3, 17, 250);
Console.WriteLine(ts1.ToString("%d"));
// Displays 16
```

```
Dim ts As New TimeSpan(16, 4, 3, 17, 250)
Console.WriteLine(ts.ToString("%d"))
' Displays 16
```

The following example illustrates the use of the "d" custom format specifier.

```
TimeSpan ts2 = new TimeSpan(4, 3, 17);
Console.WriteLine(ts2.ToString(@"d\.hh\:mm\:ss"));

TimeSpan ts3 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine(ts3.ToString(@"d\.hh\:mm\:ss"));
// The example displays the following output:
//       0.04:03:17
//       3.04:03:17
```

```
Dim ts2 As New TimeSpan(4, 3, 17)
Console.WriteLine(ts2.ToString("d\.hh\:mm\:ss"))

Dim ts3 As New TimeSpan(3, 4, 3, 17)
Console.WriteLine(ts3.ToString("d\.hh\:mm\:ss"))
' The example displays the following output:
'       0.04:03:17
'       3.04:03:17
```

Back to table

The "dd"-"dddddddd" Custom Format Specifiers

The "dd", "ddd", "dddd", "ddddd", "dddddd", "ddddddd", and "dddddddd" custom format specifiers output the value of the TimeSpan.Days property, which represents the number of whole days in the time interval.

The output string includes a minimum number of digits specified by the number of "d" characters in the format specifier, and it is padded with leading zeros as needed. If the digits in the number of days exceed the number of "d" characters in the format specifier, the full number of days is output in the result string.

The following example uses these format specifiers to display the string representation of two TimeSpan values. The value of the days component of the first time interval is zero; the value of the days component of the second is 365.

```csharp
TimeSpan ts1 = new TimeSpan(0, 23, 17, 47);
TimeSpan ts2 = new TimeSpan(365, 21, 19, 45);

for (int ctr = 2; ctr <= 8; ctr++)
{
   string fmt = new String('d', ctr) + @"\.hh\:mm\:ss";
   Console.WriteLine("{0} --> {1:" + fmt + "}", fmt, ts1);
   Console.WriteLine("{0} --> {1:" + fmt + "}", fmt, ts2);
   Console.WriteLine();
}
// The example displays the following output:
//       dd\.hh\:mm\:ss --> 00.23:17:47
//       dd\.hh\:mm\:ss --> 365.21:19:45
//
//       ddd\.hh\:mm\:ss --> 000.23:17:47
//       ddd\.hh\:mm\:ss --> 365.21:19:45
//
//       dddd\.hh\:mm\:ss --> 0000.23:17:47
//       dddd\.hh\:mm\:ss --> 0365.21:19:45
//
//       ddddd\.hh\:mm\:ss --> 00000.23:17:47
//       ddddd\.hh\:mm\:ss --> 00365.21:19:45
//
//       dddddd\.hh\:mm\:ss --> 000000.23:17:47
//       dddddd\.hh\:mm\:ss --> 000365.21:19:45
//
//       ddddddd\.hh\:mm\:ss --> 0000000.23:17:47
//       ddddddd\.hh\:mm\:ss --> 0000365.21:19:45
//
//       dddddddd\.hh\:mm\:ss --> 00000000.23:17:47
//       dddddddd\.hh\:mm\:ss --> 00000365.21:19:45
```

```vb
Dim ts1 As New TimeSpan(0, 23, 17, 47)
Dim ts2 As New TimeSpan(365, 21, 19, 45)

For ctr As Integer = 2 To 8
   Dim fmt As String = New String("d"c, ctr) + "\.hh\:mm\:ss"
   Console.WriteLine("{0} --> {1:" + fmt + "}", fmt, ts1)
   Console.WriteLine("{0} --> {1:" + fmt + "}", fmt, ts2)
   Console.WriteLine()
Next
' The example displays the following output:
'       dd\.hh\:mm\:ss --> 00.23:17:47
'       dd\.hh\:mm\:ss --> 365.21:19:45
'
'       ddd\.hh\:mm\:ss --> 000.23:17:47
'       ddd\.hh\:mm\:ss --> 365.21:19:45
'
'       dddd\.hh\:mm\:ss --> 0000.23:17:47
'       dddd\.hh\:mm\:ss --> 0365.21:19:45
'
'       ddddd\.hh\:mm\:ss --> 00000.23:17:47
'       ddddd\.hh\:mm\:ss --> 00365.21:19:45
'
'       dddddd\.hh\:mm\:ss --> 000000.23:17:47
'       dddddd\.hh\:mm\:ss --> 000365.21:19:45
'
'       ddddddd\.hh\:mm\:ss --> 0000000.23:17:47
'       ddddddd\.hh\:mm\:ss --> 0000365.21:19:45
'
'       dddddddd\.hh\:mm\:ss --> 00000000.23:17:47
'       dddddddd\.hh\:mm\:ss --> 00000365.21:19:45
```

Back to table

The "h" Custom Format Specifier

The "h" custom format specifier outputs the value of the TimeSpan.Hours property, which represents the number of whole hours in the time interval that is not counted as part of its day component. It returns a one-digit string value if the value of the TimeSpan.Hours property is 0 through 9, and it returns a two-digit string value if the value of the TimeSpan.Hours property ranges from 10 to 23.

If the "h" custom format specifier is used alone, specify "%h" so that it is not misinterpreted as a standard format string. The following example provides an illustration.

```
TimeSpan ts = new TimeSpan(3, 42, 0);
Console.WriteLine("{0:%h} hours {0:%m} minutes", ts);
// The example displays the following output:
//       3 hours 42 minutes
```

```
Dim ts As New TimeSpan(3, 42, 0)
Console.WriteLine("{0:%h} hours {0:%m} minutes", ts)
' The example displays the following output:
'       3 hours 42 minutes
```

Ordinarily, in a parsing operation, an input string that includes only a single number is interpreted as the number of days. You can use the "%h" custom format specifier instead to interpret the numeric string as the number of hours. The following example provides an illustration.

```
string value = "8";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "%h", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value);
// The example displays the following output:
//       08:00:00
```

```
Dim value As String = "8"
Dim interval As TimeSpan
If TimeSpan.TryParseExact(value, "%h", Nothing, interval) Then
   Console.WriteLine(interval.ToString("c"))
Else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value)
End If
' The example displays the following output:
'       08:00:00
```

The following example illustrates the use of the "h" custom format specifier.

```
TimeSpan ts1 = new TimeSpan(14, 3, 17);
Console.WriteLine(ts1.ToString(@"d\.h\:mm\:ss"));

TimeSpan ts2 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine(ts2.ToString(@"d\.h\:mm\:ss"));
// The example displays the following output:
//       0.14:03:17
//       3.4:03:17
```

```
Dim ts1 As New TimeSpan(14, 3, 17)
Console.WriteLine(ts1.ToString("d\.h\:mm\:ss"))

Dim ts2 As New TimeSpan(3, 4, 3, 17)
Console.WriteLine(ts2.ToString("d\.h\:mm\:ss"))
' The example displays the following output:
'       0.14:03:17
'       3.4:03:17
```

Back to table

The "hh" Custom Format Specifier

The "hh" custom format specifier outputs the value of the TimeSpan.Hours property, which represents the number of whole hours in the time interval that is not counted as part of its day component. For values from 0 through 9, the output string includes a leading zero.

Ordinarily, in a parsing operation, an input string that includes only a single number is interpreted as the number of days. You can use the "hh" custom format specifier instead to interpret the numeric string as the number of hours. The following example provides an illustration.

```
string value = "08";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "hh", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value);
// The example displays the following output:
//       08:00:00
```

```
Dim value As String = "08"
Dim interval As TimeSpan
If TimeSpan.TryParseExact(value, "hh", Nothing, interval) Then
   Console.WriteLine(interval.ToString("c"))
Else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value)
End If
' The example displays the following output:
'       08:00:00
```

The following example illustrates the use of the "hh" custom format specifier.

```
TimeSpan ts1 = new TimeSpan(14, 3, 17);
Console.WriteLine(ts1.ToString(@"d\.hh\:mm\:ss"));

TimeSpan ts2 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine(ts2.ToString(@"d\.hh\:mm\:ss"));
// The example displays the following output:
//       0.14:03:17
//       3.04:03:17
```

```
Dim ts1 As New TimeSpan(14, 3, 17)
Console.WriteLine(ts1.ToString("d\.hh\:mm\:ss"))

Dim ts2 As New TimeSpan(3, 4, 3, 17)
Console.WriteLine(ts2.ToString("d\.hh\:mm\:ss"))
' The example displays the following output:
'       0.14:03:17
'       3.04:03:17
```

Back to table

The "m" Custom Format Specifier

The "m" custom format specifier outputs the value of the TimeSpan.Minutes property, which represents the number of whole minutes in the time interval that is not counted as part of its day component. It returns a one-digit string value if the value of the TimeSpan.Minutes property is 0 through 9, and it returns a two-digit string value if the value of the TimeSpan.Minutes property ranges from 10 to 59.

If the "m" custom format specifier is used alone, specify "%m" so that it is not misinterpreted as a standard format string. The following example provides an illustration.

```
TimeSpan ts = new TimeSpan(3, 42, 0);
Console.WriteLine("{0:%h} hours {0:%m} minutes", ts);
// The example displays the following output:
//       3 hours 42 minutes
```

```
Dim ts As New TimeSpan(3, 42, 0)
Console.WriteLine("{0:%h} hours {0:%m} minutes", ts)
' The example displays the following output:
'       3 hours 42 minutes
```

Ordinarily, in a parsing operation, an input string that includes only a single number is interpreted as the number of days. You can use the "%m" custom format specifier instead to interpret the numeric string as the number of minutes. The following example provides an illustration.

```
string value = "3";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "%m", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value);
// The example displays the following output:
//       00:03:00
```

```
Dim value As String = "3"
Dim interval As TimeSpan
If TimeSpan.TryParseExact(value, "%m", Nothing, interval) Then
   Console.WriteLine(interval.ToString("c"))
Else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value)
End If
' The example displays the following output:
'       00:03:00
```

The following example illustrates the use of the "m" custom format specifier.

```
TimeSpan ts1 = new TimeSpan(0, 6, 32);
Console.WriteLine("{0:m\\:ss} minutes", ts1);

TimeSpan ts2 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine("Elapsed time: {0:m\\:ss}", ts2);
// The example displays the following output:
//       6:32 minutes
//       Elapsed time: 18:44
```

```
Dim ts1 As New TimeSpan(0, 6, 32)
Console.WriteLine("{0:m\:ss} minutes", ts1)

Dim ts2 As New TimeSpan(0, 18, 44)
Console.WriteLine("Elapsed time: {0:m\:ss}", ts2)
' The example displays the following output:
'       6:32 minutes
'       Elapsed time: 18:44
```

Back to table

The "mm" Custom Format Specifier

The "mm" custom format specifier outputs the value of the TimeSpan.Minutes property, which represents the number of whole minutes in the time interval that is not included as part of its hours or days component. For values from 0 through 9, the output string includes a leading zero.

Ordinarily, in a parsing operation, an input string that includes only a single number is interpreted as the number of days. You can use the "mm" custom format specifier instead to interpret the numeric string as the number of minutes. The following example provides an illustration.

```
string value = "07";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "mm", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value);
// The example displays the following output:
//       00:07:00
```

```
Dim value As String = "05"
Dim interval As TimeSpan
If TimeSpan.TryParseExact(value, "mm", Nothing, interval) Then
   Console.WriteLine(interval.ToString("c"))
Else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value)
End If
' The example displays the following output:
'       00:05:00
```

The following example illustrates the use of the "mm" custom format specifier.

```
TimeSpan departTime = new TimeSpan(11, 12, 00);
TimeSpan arriveTime = new TimeSpan(16, 28, 00);
Console.WriteLine("Travel time: {0:hh\\:mm}",
                  arriveTime - departTime);
// The example displays the following output:
//       Travel time: 05:16
```

```
Dim departTime As New TimeSpan(11, 12, 00)
Dim arriveTime As New TimeSpan(16, 28, 00)
Console.WriteLine("Travel time: {0:hh\:mm}",
                  arriveTime - departTime)
' The example displays the following output:
'       Travel time: 05:16
```

Back to table

The "s" Custom Format Specifier

The "s" custom format specifier outputs the value of the TimeSpan.Seconds property, which represents the number of whole seconds in the time interval that is not included as part of its hours, days, or minutes component. It returns a one-digit string value if the value of the TimeSpan.Seconds property is 0 through 9, and it returns a two-digit string value if the value of the TimeSpan.Seconds property ranges from 10 to 59.

If the "s" custom format specifier is used alone, specify "%s" so that it is not misinterpreted as a standard format string. The following example provides an illustration.

```
TimeSpan ts = TimeSpan.FromSeconds(12.465);
Console.WriteLine(ts.ToString("%s"));
// The example displays the following output:
//       12
```

```
Dim ts As TimeSpan = TimeSpan.FromSeconds(12.465)
Console.WriteLine(ts.ToString("%s"))
' The example displays the following output:
'       12
```

Ordinarily, in a parsing operation, an input string that includes only a single number is interpreted as the number of days. You can use the "%s" custom format specifier instead to interpret the numeric string as the number of seconds. The following example provides an illustration.

```
string value = "9";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "%s", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value);
// The example displays the following output:
//       00:00:09
```

```
Dim value As String = "9"
Dim interval As TimeSpan
If TimeSpan.TryParseExact(value, "%s", Nothing, interval) Then
   Console.WriteLine(interval.ToString("c"))
Else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value)
End If
' The example displays the following output:
'       00:00:09
```

The following example illustrates the use of the "s" custom format specifier.

```
TimeSpan startTime = new TimeSpan(0, 12, 30, 15, 0);
TimeSpan endTime = new TimeSpan(0, 12, 30, 21, 3);
Console.WriteLine(@"Elapsed Time: {0:s\:fff} seconds",
                  endTime - startTime);
// The example displays the following output:
//       Elapsed Time: 6:003 seconds
```

```
Dim startTime As New TimeSpan(0, 12, 30, 15, 0)
Dim endTime As New TimeSpan(0, 12, 30, 21, 3)
Console.WriteLine("Elapsed Time: {0:s\:fff} seconds",
                  endTime - startTime)
' The example displays the following output:
'       Elapsed Time: 6:003 seconds
```

Back to table

The "ss" Custom Format Specifier

The "ss" custom format specifier outputs the value of the TimeSpan.Seconds property, which represents the number of whole seconds in the time interval that is not included as part of its hours, days, or minutes component. For values from 0 through 9, the output string includes a leading zero.

Ordinarily, in a parsing operation, an input string that includes only a single number is interpreted as the number of days. You can use the "ss" custom format specifier instead to interpret the numeric string as the number of seconds. The following example provides an illustration.

```
string[] values = { "49", "9", "06" };
TimeSpan interval;
foreach (string value in values)
{
   if (TimeSpan.TryParseExact(value, "ss", null, out interval))
      Console.WriteLine(interval.ToString("c"));
   else
      Console.WriteLine("Unable to convert '{0}' to a time interval",
                        value);
}
// The example displays the following output:
//       00:00:49
//       Unable to convert '9' to a time interval
//       00:00:06
```

```
Dim values() As String = { "49", "9", "06" }
Dim interval As TimeSpan
For Each value As String In values
   If TimeSpan.TryParseExact(value, "ss", Nothing, interval) Then
      Console.WriteLine(interval.ToString("c"))
   Else
      Console.WriteLine("Unable to convert '{0}' to a time interval",
                        value)
   End If
Next
' The example displays the following output:
'       00:00:49
'       Unable to convert '9' to a time interval
'       00:00:06
```

The following example illustrates the use of the "ss" custom format specifier.

```
TimeSpan interval1 = TimeSpan.FromSeconds(12.60);
Console.WriteLine(interval1.ToString(@"ss\.fff"));

TimeSpan interval2 = TimeSpan.FromSeconds(6.485);
Console.WriteLine(interval2.ToString(@"ss\.fff"));
// The example displays the following output:
//       12.600
//       06.485
```

```
Dim interval1 As TimeSpan = TimeSpan.FromSeconds(12.60)
Console.WriteLine(interval1.ToString("ss\.fff"))
Dim interval2 As TimeSpan = TimeSpan.FromSeconds(6.485)
Console.WriteLine(interval2.ToString("ss\.fff"))
' The example displays the following output:
'       12.600
'       06.485
```

Back to table

The"f" Custom Format Specifier

The "f" custom format specifier outputs the tenths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the input string must contain exactly one fractional digit.

If the "f" custom format specifier is used alone, specify "%f" so that it is not misinterpreted as a standard format string.

The following example uses the "f" custom format specifier to display the tenths of a second in a TimeSpan value. "f" is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

```csharp
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//                %f: 8
//                ff: 87
//               fff: 876
//              ffff: 8765
//             fffff: 87654
//            ffffff: 876543
//           fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

```vb
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   If fmt.Length = 1 Then fmt = "%" + fmt
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'            %f: 8
'            ff: 87
'           fff: 876
'          ffff: 8765
'         fffff: 87654
'        ffffff: 876543
'       fffffff: 8765432
'
'           s\.f: 29.8
'          s\.ff: 29.87
'         s\.fff: 29.876
'        s\.ffff: 29.8765
'       s\.fffff: 29.87654
'      s\.ffffff: 29.876543
'     s\.fffffff: 29.8765432
```

[Back to table](#)

The "ff" Custom Format Specifier

The "ff" custom format specifier outputs the hundredths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the input string must contain exactly two fractional digits.

The following example uses the "ff" custom format specifier to display the hundredths of a second in a TimeSpan value. "ff" is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//             s\.f: 29.8
//            s\.ff: 29.87
//           s\.fff: 29.876
//          s\.ffff: 29.8765
//         s\.fffff: 29.87654
//        s\.ffffff: 29.876543
//       s\.fffffff: 29.8765432
```

```
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   If fmt.Length = 1 Then fmt = "%" + fmt
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'            %f: 8
'            ff: 87
'           fff: 876
'          ffff: 8765
'         fffff: 87654
'        ffffff: 876543
'       fffffff: 8765432
'
'           s\.f: 29.8
'          s\.ff: 29.87
'         s\.fff: 29.876
'        s\.ffff: 29.8765
'       s\.fffff: 29.87654
'      s\.ffffff: 29.876543
'     s\.fffffff: 29.8765432
```

Back to table

The "fff" Custom Format Specifier

The "fff" custom format specifier (with three "f" characters) outputs the milliseconds in a time interval. In a formatting operation, any remaining fractional digits are truncated. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the input string must contain exactly three fractional digits.

The following example uses the "fff" custom format specifier to display the milliseconds in a TimeSpan value. "fff" is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//                %f: 8
//                ff: 87
//               fff: 876
//              ffff: 8765
//             fffff: 87654
//            ffffff: 876543
//           fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

```
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   If fmt.Length = 1 Then fmt = "%" + fmt
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'            %f: 8
'            ff: 87
'           fff: 876
'          ffff: 8765
'         fffff: 87654
'        ffffff: 876543
'       fffffff: 8765432
'
'           s\.f: 29.8
'          s\.ff: 29.87
'         s\.fff: 29.876
'        s\.ffff: 29.8765
'       s\.fffff: 29.87654
'      s\.ffffff: 29.876543
'     s\.fffffff: 29.8765432
```

Back to table

The "ffff" Custom Format Specifier

The "ffff" custom format specifier (with four "f" characters) outputs the ten-thousandths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the input string must contain exactly four fractional digits.

The following example uses the "ffff" custom format specifier to display the ten-thousandths of a second in a TimeSpan value. "ffff" is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

```
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   If fmt.Length = 1 Then fmt = "%" + fmt
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'            %f: 8
'            ff: 87
'           fff: 876
'          ffff: 8765
'         fffff: 87654
'        ffffff: 876543
'       fffffff: 8765432
'
'           s\.f: 29.8
'          s\.ff: 29.87
'         s\.fff: 29.876
'        s\.ffff: 29.8765
'       s\.fffff: 29.87654
'      s\.ffffff: 29.876543
'     s\.fffffff: 29.8765432
```

Back to table

The "fffff" Custom Format Specifier

The "fffff" custom format specifier (with five "f" characters) outputs the hundred-thousandths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the input string must contain exactly five fractional digits.

The following example uses the "fffff" custom format specifier to display the hundred-thousandths of a second in a TimeSpan value. "fffff" is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//                %f: 8
//                ff: 87
//               fff: 876
//              ffff: 8765
//             fffff: 87654
//            ffffff: 876543
//           fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

```
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   If fmt.Length = 1 Then fmt = "%" + fmt
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'            %f: 8
'            ff: 87
'           fff: 876
'          ffff: 8765
'         fffff: 87654
'        ffffff: 876543
'       fffffff: 8765432
'
'          s\.f: 29.8
'         s\.ff: 29.87
'        s\.fff: 29.876
'       s\.ffff: 29.8765
'      s\.fffff: 29.87654
'     s\.ffffff: 29.876543
'    s\.fffffff: 29.8765432
```

Back to table

The "ffffff" Custom Format Specifier

The "ffffff" custom format specifier (with six "f" characters) outputs the millionths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the input string must contain exactly six fractional digits.

The following example uses the "ffffff" custom format specifier to display the millionths of a second in a TimeSpan value. It is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//             s\.f: 29.8
//            s\.ff: 29.87
//           s\.fff: 29.876
//          s\.ffff: 29.8765
//         s\.fffff: 29.87654
//        s\.ffffff: 29.876543
//       s\.fffffff: 29.8765432
```

```vb
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   If fmt.Length = 1 Then fmt = "%" + fmt
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'            %f: 8
'            ff: 87
'           fff: 876
'          ffff: 8765
'         fffff: 87654
'        ffffff: 876543
'       fffffff: 8765432
'
'           s\.f: 29.8
'          s\.ff: 29.87
'         s\.fff: 29.876
'        s\.ffff: 29.8765
'       s\.fffff: 29.87654
'      s\.ffffff: 29.876543
'     s\.fffffff: 29.8765432
```

Back to table

The "fffffff" Custom Format Specifier

The "fffffff" custom format specifier (with seven "f" characters) outputs the ten-millionths of a second (or the fractional number of ticks) in a time interval. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the input string must contain exactly seven fractional digits.

The following example uses the "fffffff" custom format specifier to display the fractional number of ticks in a TimeSpan value. It is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

```
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   If fmt.Length = 1 Then fmt = "%" + fmt
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
   fmt = New String("f"c, ctr)
   Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'            %f: 8
'            ff: 87
'           fff: 876
'          ffff: 8765
'         fffff: 87654
'        ffffff: 876543
'       fffffff: 8765432
'
'           s\.f: 29.8
'          s\.ff: 29.87
'         s\.fff: 29.876
'        s\.ffff: 29.8765
'       s\.fffff: 29.87654
'      s\.ffffff: 29.876543
'     s\.fffffff: 29.8765432
```

Back to table

The "F" Custom Format Specifier

The "F" custom format specifier outputs the tenths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. If the value of the time interval's tenths of a second is zero, it is not included in the result string. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the presence of the tenths of a second digit is optional.

If the "F" custom format specifier is used alone, specify "%F" so that it is not misinterpreted as a standard format string.

The following example uses the "F" custom format specifier to display the tenths of a second in a TimeSpan value. It also uses this custom format specifier in a parsing operation.

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.669");
Console.WriteLine("{0} ('%F') --> {0:%F}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.091");
Console.WriteLine("{0} ('ss\\.F') --> {0:ss\\.F}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.1", "0:0:03.12" };
string fmt = @"h\:m\:ss\.F";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6690000 ('%F') --> 6
//       00:00:03.0910000 ('ss\.F') --> 03.
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.F') --> 00:00:03
//       0:0:03.1 ('h\:m\:ss\.F') --> 00:00:03.1000000
//       Cannot parse 0:0:03.12 with 'h\:m\:ss\.F'.
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.669")
Console.WriteLine("{0} ('%F') --> {0:%F}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.091")
Console.WriteLine("{0} ('ss\.F') --> {0:ss\.F}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = { "0:0:03.", "0:0:03.1", "0:0:03.12" }
Dim fmt As String = "h\:m\:ss\.F"
Dim ts3 As TimeSpan

For Each input As String In inputs
   If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
   Else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt)
   End If
Next
' The example displays the following output:
'       Formatting:
'       00:00:03.6690000 ('%F') --> 6
'       00:00:03.0910000 ('ss\.F') --> 03.
'
'       Parsing:
'       0:0:03. ('h\:m\:ss\.F') --> 00:00:03
'       0:0:03.1 ('h\:m\:ss\.F') --> 00:00:03.1000000
'       Cannot parse 0:0:03.12 with 'h\:m\:ss\.F'.
```

Back to table

The "FF" Custom Format Specifier

The "FF" custom format specifier outputs the hundredths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. If there are any trailing fractional zeros, they are not included in the result string. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the presence of the tenths and hundredths of a second digit is optional.

The following example uses the "FF" custom format specifier to display the hundredths of a second in a TimeSpan value. It also uses this custom format specifier in a parsing operation.

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.697");
Console.WriteLine("{0} ('FF') --> {0:FF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.809");
Console.WriteLine("{0} ('ss\\.FF') --> {0:ss\\.FF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.1", "0:0:03.127" };
string fmt = @"h\:m\:ss\.FF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6970000 ('FF') --> 69
//       00:00:03.8090000 ('ss\.FF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FF') --> 00:00:03
//       0:0:03.1 ('h\:m\:ss\.FF') --> 00:00:03.1000000
//       Cannot parse 0:0:03.127 with 'h\:m\:ss\.FF'.
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.697")
Console.WriteLine("{0} ('FF') --> {0:FF}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.809")
Console.WriteLine("{0} ('ss\.FF') --> {0:ss\.FF}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = { "0:0:03.", "0:0:03.1", "0:0:03.127" }
Dim fmt As String = "h\:m\:ss\.FF"
Dim ts3 As TimeSpan

For Each input As String In inputs
   If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
   Else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt)
   End If
Next
' The example displays the following output:
'       Formatting:
'       00:00:03.6970000 ('FF') --> 69
'       00:00:03.8090000 ('ss\.FF') --> 03.8
'
'       Parsing:
'       0:0:03. ('h\:m\:ss\.FF') --> 00:00:03
'       0:0:03.1 ('h\:m\:ss\.FF') --> 00:00:03.1000000
'       Cannot parse 0:0:03.127 with 'h\:m\:ss\.FF'.
```

Back to table

The "FFF" Custom Format Specifier

The "FFF" custom format specifier (with three "F" characters) outputs the milliseconds in a time interval. In a formatting operation, any remaining fractional digits are truncated. If there are any trailing fractional zeros, they are not included in the result string. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the presence of the tenths, hundredths, and thousandths of a second digit is optional.

The following example uses the "FFF" custom format specifier to display the thousandths of a second in a TimeSpan value. It also uses this custom format specifier in a parsing operation.

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.6974");
Console.WriteLine("{0} ('FFF') --> {0:FFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.8009");
Console.WriteLine("{0} ('ss\\.FFF') --> {0:ss\\.FFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.1279" };
string fmt = @"h\:m\:ss\.FFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6974000 ('FFF') --> 697
//       00:00:03.8009000 ('ss\.FFF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FFF') --> 00:00:03
//       0:0:03.12 ('h\:m\:ss\.FFF') --> 00:00:03.1200000
//       Cannot parse 0:0:03.1279 with 'h\:m\:ss\.FFF'.
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.6974")
Console.WriteLine("{0} ('FFF') --> {0:FFF}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.8009")
Console.WriteLine("{0} ('ss\.FFF') --> {0:ss\.FFF}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = { "0:0:03.", "0:0:03.12", "0:0:03.1279" }
Dim fmt As String = "h\:m\:ss\.FFF"
Dim ts3 As TimeSpan

For Each input As String In inputs
   If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
   Else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt)
   End If
Next
' The example displays the following output:
'       Formatting:
'       00:00:03.6974000 ('FFF') --> 697
'       00:00:03.8009000 ('ss\.FFF') --> 03.8
'
'       Parsing:
'       0:0:03. ('h\:m\:ss\.FFF') --> 00:00:03
'       0:0:03.12 ('h\:m\:ss\.FFF') --> 00:00:03.1200000
'       Cannot parse 0:0:03.1279 with 'h\:m\:ss\.FFF'.
```

Back to table

The "FFFF" Custom Format Specifier

The "FFFF" custom format specifier (with four "F" characters) outputs the ten-thousandths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. If there are any trailing fractional zeros, they are not included in the result string. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the presence of the tenths, hundredths, thousandths, and ten-thousandths of a second digit is optional.

The following example uses the "FFFF" custom format specifier to display the ten-thousandths of a second in a TimeSpan value. It also uses the "FFFF" custom format specifier in a parsing operation.

```csharp
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.69749");
Console.WriteLine("{0} ('FFFF') --> {0:FFFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.80009");
Console.WriteLine("{0} ('ss\\.FFFF') --> {0:ss\\.FFFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.12795" };
string fmt = @"h\:m\:ss\.FFFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6974900 ('FFFF') --> 6974
//       00:00:03.8000900 ('ss\.FFFF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FFFF') --> 00:00:03
//       0:0:03.12 ('h\:m\:ss\.FFFF') --> 00:00:03.1200000
//       Cannot parse 0:0:03.12795 with 'h\:m\:ss\.FFFF'.
```

```vb
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.69749")
Console.WriteLine("{0} ('FFFF') --> {0:FFFF}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.80009")
Console.WriteLine("{0} ('ss\.FFFF') --> {0:ss\.FFFF}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = { "0:0:03.", "0:0:03.12", "0:0:03.12795" }
Dim fmt As String = "h\:m\:ss\.FFFF"
Dim ts3 As TimeSpan

For Each input As String In inputs
   If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
   Else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt)
   End If
Next
' The example displays the following output:
'       Formatting:
'       00:00:03.6974900 ('FFFF') --> 6974
'       00:00:03.8000900 ('ss\.FFFF') --> 03.8
'
'       Parsing:
'       0:0:03. ('h\:m\:ss\.FFFF') --> 00:00:03
'       0:0:03.12 ('h\:m\:ss\.FFFF') --> 00:00:03.1200000
'       Cannot parse 0:0:03.12795 with 'h\:m\:ss\.FFFF'.
```

Back to table

The "FFFFF" Custom Format Specifier

The "FFFFF" custom format specifier (with five "F" characters) outputs the hundred-thousandths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. If there are any trailing fractional zeros, they are not included in the result string. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the presence of the tenths, hundredths, thousandths, ten-thousandths, and hundred-thousandths of a second digit is optional.

The following example uses the "FFFFF" custom format specifier to display the hundred-thousandths of a second in a TimeSpan value. It also uses the "FFFFF" custom format specifier in a parsing operation.

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.697497");
Console.WriteLine("{0} ('FFFFF') --> {0:FFFFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.800009");
Console.WriteLine("{0} ('ss\\.FFFFF') --> {0:ss\\.FFFFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.127956" };
string fmt = @"h\:m\:ss\.FFFFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6974970 ('FFFFF') --> 69749
//       00:00:03.8000090 ('ss\.FFFFF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FFFF') --> 00:00:03
//       0:0:03.12 ('h\:m\:ss\.FFFF') --> 00:00:03.1200000
//       Cannot parse 0:0:03.127956 with 'h\:m\:ss\.FFFF'.
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.697497")
Console.WriteLine("{0} ('FFFFF') --> {0:FFFFF}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.800009")
Console.WriteLine("{0} ('ss\.FFFFF') --> {0:ss\.FFFFF}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = { "0:0:03.", "0:0:03.12", "0:0:03.127956" }
Dim fmt As String = "h\:m\:ss\.FFFFF"
Dim ts3 As TimeSpan

For Each input As String In inputs
   If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
   Else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt)
   End If
Next
' The example displays the following output:
'       Formatting:
'       00:00:03.6974970 ('FFFFF') --> 69749
'       00:00:03.8000090 ('ss\.FFFFF') --> 03.8
'
'       Parsing:
'       0:0:03. ('h\:m\:ss\.FFFF') --> 00:00:03
'       0:0:03.12 ('h\:m\:ss\.FFFF') --> 00:00:03.1200000
'       Cannot parse 0:0:03.127956 with 'h\:m\:ss\.FFFF'.
```

Back to table

The "FFFFFF" Custom Format Specifier

The "FFFFFF" custom format specifier (with six "F" characters) outputs the millionths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. If there are any trailing fractional zeros, they are not included in the result string. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the presence of the tenths, hundredths, thousandths, ten-thousandths, hundred-thousandths, and millionths of a second digit is optional.

The following example uses the "FFFFFF" custom format specifier to display the millionths of a second in a TimeSpan value. It also uses this custom format specifier in a parsing operation.

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.6974974");
Console.WriteLine("{0} ('FFFFFF') --> {0:FFFFFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.8000009");
Console.WriteLine("{0} ('ss\\.FFFFFF') --> {0:ss\\.FFFFFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.1279569" };
string fmt = @"h\:m\:ss\.FFFFFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6974974 ('FFFFFF') --> 697497
//       00:00:03.8000009 ('ss\.FFFFFF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FFFFFF') --> 00:00:03
//       0:0:03.12 ('h\:m\:ss\.FFFFFF') --> 00:00:03.1200000
//       Cannot parse 0:0:03.1279569 with 'h\:m\:ss\.FFFFFF'.
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.6974974")
Console.WriteLine("{0} ('FFFFFF') --> {0:FFFFFF}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.8000009")
Console.WriteLine("{0} ('ss\.FFFFFF') --> {0:ss\.FFFFFF}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = { "0:0:03.", "0:0:03.12", "0:0:03.1279569" }
Dim fmt As String = "h\:m\:ss\.FFFFFF"
Dim ts3 As TimeSpan

For Each input As String In inputs
   If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
   Else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt)
   End If
Next
' The example displays the following output:
'       Formatting:
'       00:00:03.6974974 ('FFFFFF') --> 697497
'       00:00:03.8000009 ('ss\.FFFFFF') --> 03.8
'
'       Parsing:
'       0:0:03. ('h\:m\:ss\.FFFFFF') --> 00:00:03
'       0:0:03.12 ('h\:m\:ss\.FFFFFF') --> 00:00:03.1200000
'       Cannot parse 0:0:03.1279569 with 'h\:m\:ss\.FFFFFF'.
```

Back to table

The "FFFFFFF" Custom Format Specifier

The "FFFFFFF" custom format specifier (with seven "F" characters) outputs the ten-millionths of a second (or the fractional number of ticks) in a time interval. If there are any trailing fractional zeros, they are not included in the result string. In a parsing operation that calls the TimeSpan.ParseExact or TimeSpan.TryParseExact method, the presence of the seven fractional digits in the input string is optional.

The following example uses the "FFFFFFF" custom format specifier to display the fractional parts of a second in a TimeSpan value. It also uses this custom format specifier in a parsing operation.

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.6974974");
Console.WriteLine("{0} ('FFFFFFF') --> {0:FFFFFFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.9500000");
Console.WriteLine("{0} ('ss\\.FFFFFFF') --> {0:ss\\.FFFFFFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.1279569" };
string fmt = @"h\:m\:ss\.FFFFFFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//    Formatting:
//    00:00:03.6974974 ('FFFFFFF') --> 6974974
//    00:00:03.9500000 ('ss\.FFFFFFF') --> 03.95
//
//    Parsing:
//    0:0:03. ('h\:m\:ss\.FFFFFFF') --> 00:00:03
//    0:0:03.12 ('h\:m\:ss\.FFFFFFF') --> 00:00:03.1200000
//    0:0:03.1279569 ('h\:m\:ss\.FFFFFFF') --> 00:00:03.1279569
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.6974974")
Console.WriteLine("{0} ('FFFFFFF') --> {0:FFFFFFF}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.9500000")
Console.WriteLine("{0} ('ss\.FFFFFFF') --> {0:ss\.FFFFFFF}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = { "0:0:03.", "0:0:03.12", "0:0:03.1279569" }
Dim fmt As String = "h\:m\:ss\.FFFFFFF"
Dim ts3 As TimeSpan

For Each input As String In inputs
   If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
   Else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt)
   End If
Next
' The example displays the following output:
'    Formatting:
'    00:00:03.6974974 ('FFFFFFF') --> 6974974
'    00:00:03.9500000 ('ss\.FFFFFFF') --> 03.95
'
'    Parsing:
'    0:0:03. ('h\:m\:ss\.FFFFFFF') --> 00:00:03
'    0:0:03.12 ('h\:m\:ss\.FFFFFFF') --> 00:00:03.1200000
'    0:0:03.1279569 ('h\:m\:ss\.FFFFFFF') --> 00:00:03.1279569
```

Back to table

Other Characters

Any other unescaped character in a format string, including a white-space character, is interpreted as a custom format specifier. In most cases, the presence of any other unescaped character results in a FormatException.

There are two ways to include a literal character in a format string:

- Enclose it in single quotation marks (the literal string delimiter).
- Precede it with a backslash ("\"), which is interpreted as an escape character. This means that, in C#, the format string must either be @-quoted, or the literal character must be preceded by an additional backslash.

  In some cases, you may have to use conditional logic to include an escaped literal in a format string. The following example uses conditional logic to include a sign symbol for negative time intervals.

```
using System;

public class Example
{
   public static void Main()
   {
      TimeSpan result = new DateTime(2010, 01, 01) - DateTime.Now;
      String fmt = (result < TimeSpan.Zero ?  "\\-" : "") + "dd\\.hh\\:mm";

      Console.WriteLine(result.ToString(fmt));
      Console.WriteLine("Interval: {0:" + fmt + "}", result);
   }
}
// The example displays output like the following:
//       -1291.10:54
//       Interval: -1291.10:54
```

```
Module Example
   Public Sub Main()
      Dim result As TimeSpan = New DateTime(2010, 01, 01) - Date.Now
      Dim fmt As String = If(result < TimeSpan.Zero, "\-", "") + "dd\.hh\:mm"

      Console.WriteLine(result.ToString(fmt))
      Console.WriteLine("Interval: {0:" + fmt + "}", result)
   End Sub
End Module
' The example displays output like the following:
'       -1291.10:54
'       Interval: -1291.10:54
```

.NET does not define a grammar for separators in time intervals. This means that the separators between days and hours, hours and minutes, minutes and seconds, and seconds and fractions of a second must all be treated as character literals in a format string.

The following example uses both the escape character and the single quote to define a custom format string that includes the word "minutes" in the output string.

```
TimeSpan interval = new TimeSpan(0, 32, 45);
// Escape literal characters in a format string.
string fmt = @"mm\:ss\ \m\i\n\u\t\e\s";
Console.WriteLine(interval.ToString(fmt));
// Delimit literal characters in a format string with the ' symbol.
fmt = "mm':'ss' minutes'";
Console.WriteLine(interval.ToString(fmt));
// The example displays the following output:
//       32:45 minutes
//       32:45 minutes
```

```
Dim interval As New TimeSpan(0, 32, 45)
' Escape literal characters in a format string.
Dim fmt As String = "mm\:ss\ \m\i\n\u\t\e\s"
Console.WriteLine(interval.ToString(fmt))
' Delimit literal characters in a format string with the ' symbol.
fmt = "mm':'ss' minutes'"
Console.WriteLine(interval.ToString(fmt))
' The example displays the following output:
'       32:45 minutes
'       32:45 minutes
```

Back to table

See also

- Formatting Types
- Standard TimeSpan Format Strings

Enumeration Format Strings

9/6/2018 • 3 minutes to read • Edit Online

You can use the Enum.ToString method to create a new string object that represents the numeric, hexadecimal, or string value of an enumeration member. This method takes one of the enumeration formatting strings to specify the value that you want returned.

The following table lists the enumeration formatting strings and the values they return. These format specifiers are not case-sensitive.

FORMAT STRING	RESULT

G or g	Displays the enumeration entry as a string value, if possible, and otherwise displays the integer value of the current instance. If the enumeration is defined with the **Flags** attribute set, the string values of each valid entry are concatenated together, separated by commas. If the **Flags** attribute is not set, an invalid value is displayed as a numeric entry. The following example illustrates the G format specifier.
	`Console.WriteLine(ConsoleColor.Red.ToString("G"));         // Displays Red`
	`FileAttributes attributes = FileAttributes.Hidden |`
	`FileAttributes.Archive;`
	`Console.WriteLine(attributes.ToString("G"));   // Displays Hidden, Archive`
	
	`Console.WriteLine(ConsoleColor.Red.ToString("G"))           ' Displays Red`
	`Dim attributes As FileAttributes = FileAttributes.Hidden Or _`
	`FileAttributes.Archive`
	`Console.WriteLine(attributes.ToString("G"))   ' Displays Hidden, Archive`

<table>
<tr><th>FORMAT STRING</th><th>RESULT</th></tr>
<tr><td>F or f</td><td>Displays the enumeration entry as a string value, if possible. If the value can be completely displayed as a summation of the entries in the enumeration (even if the Flags attribute is not present), the string values of each valid entry are concatenated together, separated by commas. If the value cannot be completely determined by the enumeration entries, then the value is formatted as the integer value. The following example illustrates the F format specifier.
<pre>
Console.WriteLine(ConsoleColor.Blue.ToString("F"
));   // Displays Blue
FileAttributes attributes =
FileAttributes.Hidden |

FileAttributes.Archive;
Console.WriteLine(attributes.ToString("F"));
// Displays Hidden, Archive
</pre>
<pre>
Console.WriteLine(ConsoleColor.Blue.ToString("F"
))   ' Displays Blue
Dim attributes As FileAttributes =
FileAttributes.Hidden Or _

FileAttributes.Archive
Console.WriteLine(attributes.ToString("F"))
' Displays Hidden, Archive
</pre></td></tr>
<tr><td>D or d</td><td>Displays the enumeration entry as an integer value in the shortest representation possible. The following example illustrates the D format specifier.
<pre>
Console.WriteLine(ConsoleColor.Cyan.ToString("D"
));   // Displays 11
FileAttributes attributes =
FileAttributes.Hidden |

FileAttributes.Archive;
Console.WriteLine(attributes.ToString("D"));
// Displays 34
</pre>
<pre>
Console.WriteLine(ConsoleColor.Cyan.ToString("D"
))   ' Displays 11
Dim attributes As FileAttributes =
FileAttributes.Hidden Or _

FileAttributes.Archive
Console.WriteLine(attributes.ToString("D"))
' Displays 34
</pre></td></tr>
</table>

FORMAT STRING	RESULT

X or x	Displays the enumeration entry as a hexadecimal value. The value is represented with leading zeros as necessary, to ensure that the value is a minimum eight digits in length. The following example illustrates the X format specifier.
	`Console.WriteLine(ConsoleColor.Cyan.ToString("X"));   // Displays 0000000B`
	`FileAttributes attributes = FileAttributes.Hidden |`
	`FileAttributes.Archive;`
	`Console.WriteLine(attributes.ToString("X"));   // Displays 00000022`
	
	`Console.WriteLine(ConsoleColor.Cyan.ToString("X"))     ' Displays 0000000B`
	`Dim attributes As FileAttributes = FileAttributes.Hidden Or _`
	`FileAttributes.Archive`
	`Console.WriteLine(attributes.ToString("X"))   ' Displays 00000022`

Example

The following example defines an enumeration called `Colors` that consists of three entries: `Red`, `Blue`, and `Green`.

```
public enum Color {Red = 1, Blue = 2, Green = 3}
```

```
Public Enum Color
   Red = 1
   Blue = 2
   Green = 3
End Enum
```

After the enumeration is defined, an instance can be declared in the following manner.

```
Color myColor = Color.Green;
```

```
Dim myColor As Color = Color.Green
```

The `Color.ToString(System.String)` method can then be used to display the enumeration value in different ways, depending on the format specifier passed to it.

```csharp
Console.WriteLine("The value of myColor is {0}.",
                  myColor.ToString("G"));
Console.WriteLine("The value of myColor is {0}.",
                  myColor.ToString("F"));
Console.WriteLine("The value of myColor is {0}.",
                  myColor.ToString("D"));
Console.WriteLine("The value of myColor is 0x{0}.",
                  myColor.ToString("X"));
// The example displays the following output to the console:
//       The value of myColor is Green.
//       The value of myColor is Green.
//       The value of myColor is 3.
//       The value of myColor is 0x00000003.
```

```vb
Console.WriteLine("The value of myColor is {0}.", _
                  myColor.ToString("G"))
Console.WriteLine("The value of myColor is {0}.", _
                  myColor.ToString("F"))
Console.WriteLine("The value of myColor is {0}.", _
                  myColor.ToString("D"))
Console.WriteLine("The value of myColor is 0x{0}.", _
                  myColor.ToString("X"))
' The example displays the following output to the console:
'       The value of myColor is Green.
'       The value of myColor is Green.
'       The value of myColor is 3.
'       The value of myColor is 0x00000003.
```

See also

- Formatting Types

Composite Formatting

9/6/2018 • 12 minutes to read • Edit Online

The .NET composite formatting feature takes a list of objects and a composite format string as input. A composite format string consists of fixed text intermixed with indexed placeholders, called format items, that correspond to the objects in the list. The formatting operation yields a result string that consists of the original fixed text intermixed with the string representation of the objects in the list.

The composite formatting feature is supported by methods such as the following:

- String.Format, which returns a formatted result string.
- StringBuilder.AppendFormat, which appends a formatted result string to a StringBuilder object.
- Some overloads of the Console.WriteLine method, which display a formatted result string to the console.
- Some overloads of the TextWriter.WriteLine method, which write the formatted result string to a stream or file. The classes derived from TextWriter, such as StreamWriter and HtmlTextWriter, also share this functionality.
- Debug.WriteLine(String, Object[]), which outputs a formatted message to trace listeners.
- The Trace.TraceError(String, Object[]), Trace.TraceInformation(String, Object[]), and Trace.TraceWarning(String, Object[]) methods, which output formatted messages to trace listeners.
- The TraceSource.TraceInformation(String, Object[]) method, which writes an informational method to trace listeners.

Composite Format String

A composite format string and object list are used as arguments of methods that support the composite formatting feature. A composite format string consists of zero or more runs of fixed text intermixed with one or more format items. The fixed text is any string that you choose, and each format item corresponds to an object or boxed structure in the list. The composite formatting feature returns a new result string where each format item is replaced by the string representation of the corresponding object in the list.

Consider the following Format code fragment.

```
string name = "Fred";
String.Format("Name = {0}, hours = {1:hh}", name, DateTime.Now);
```

```
Dim name As String = "Fred"
String.Format("Name = {0}, hours = {1:hh}", name, DateTime.Now)
```

The fixed text is "`Name =` " and "`, hours =` ". The format items are "`{0}`", whose index is 0, which corresponds to the object `name`, and "`{1:hh}`", whose index is 1, which corresponds to the object `DateTime.Now`.

Format Item Syntax

Each format item takes the following form and consists of the following components:

`{` *index*[`,` *alignment*][`:` *formatString*] `}`

The matching braces ("{" and "}") are required.

Index Component

The mandatory *index* component, also called a parameter specifier, is a number starting from 0 that identifies a corresponding item in the list of objects. That is, the format item whose parameter specifier is 0 formats the first object in the list, the format item whose parameter specifier is 1 formats the second object in the list, and so on. The following example includes four parameter specifiers, numbered zero through three, to represent prime numbers less than ten:

```
string primes;
primes = String.Format("Prime numbers less than 10: {0}, {1}, {2}, {3}",
                       2, 3, 5, 7 );
Console.WriteLine(primes);
// The example displays the following output:
//      Prime numbers less than 10: 2, 3, 5, 7
```

```
Dim primes As String
primes = String.Format("Prime numbers less than 10: {0}, {1}, {2}, {3}",
                       2, 3, 5, 7 )
Console.WriteLine(primes)
' The example displays the following output:
'      Prime numbers less than 10: 2, 3, 5, 7
```

Multiple format items can refer to the same element in the list of objects by specifying the same parameter specifier. For example, you can format the same numeric value in hexadecimal, scientific, and number format by specifying a composite format string such as : "0x{0:X} {0:E} {0:N}", as the following example shows.

```
string multiple = String.Format("0x{0:X} {0:E} {0:N}",
                                Int64.MaxValue);
Console.WriteLine(multiple);
// The example displays the following output:
//      0x7FFFFFFFFFFFFFFF 9.223372E+018 9,223,372,036,854,775,807.00
```

```
Dim multiple As String = String.Format("0x{0:X} {0:E} {0:N}",
                                       Int64.MaxValue)
Console.WriteLine(multiple)
' The example displays the following output:
'      0x7FFFFFFFFFFFFFFF 9.223372E+018 9,223,372,036,854,775,807.00
```

Each format item can refer to any object in the list. For example, if there are three objects, you can format the second, first, and third object by specifying a composite format string like this: "{1} {0} {2}". An object that is not referenced by a format item is ignored. A FormatException is thrown at runtime if a parameter specifier designates an item outside the bounds of the list of objects.

Alignment Component

The optional *alignment* component is a signed integer indicating the preferred formatted field width. If the value of *alignment* is less than the length of the formatted string, *alignment* is ignored and the length of the formatted string is used as the field width. The formatted data in the field is right-aligned if *alignment* is positive and left-aligned if *alignment* is negative. If padding is necessary, white space is used. The comma is required if *alignment* is specified.

The following example defines two arrays, one containing the names of employees and the other containing the hours they worked over a two-week period. The composite format string left-aligns the names in a 20-character field, and right-aligns their hours in a 5-character field. Note that the "N1" standard format string is also used to format the hours with one fractional digit.

```
using System;

public class Example
{
   public static void Main()
   {
      string[] names = { "Adam", "Bridgette", "Carla", "Daniel",
                         "Ebenezer", "Francine", "George" };
      decimal[] hours = { 40, 6.667m, 40.39m, 82, 40.333m, 80,
                                 16.75m };

      Console.WriteLine("{0,-20} {1,5}\n", "Name", "Hours");
      for (int ctr = 0; ctr < names.Length; ctr++)
         Console.WriteLine("{0,-20} {1,5:N1}", names[ctr], hours[ctr]);

   }
}
// The example displays the following output:
//       Name                 Hours
//
//       Adam                  40.0
//       Bridgette              6.7
//       Carla                 40.4
//       Daniel                82.0
//       Ebenezer              40.3
//       Francine              80.0
//       George                16.8
```

```
Module Example
   Public Sub Main()
      Dim names() As String = { "Adam", "Bridgette", "Carla", "Daniel",
                                "Ebenezer", "Francine", "George" }
      Dim hours() As Decimal = { 40, 6.667d, 40.39d, 82, 40.333d, 80,
                                 16.75d }

      Console.WriteLine("{0,-20} {1,5}", "Name", "Hours")
      Console.WriteLine()
      For ctr As Integer = 0 To names.Length - 1
         Console.WriteLine("{0,-20} {1,5:N1}", names(ctr), hours(ctr))
      Next
   End Sub
End Module
' The example displays the following output:
'       Name                 Hours
'
'       Adam                  40.0
'       Bridgette              6.7
'       Carla                 40.4
'       Daniel                82.0
'       Ebenezer              40.3
'       Francine              80.0
'       George                16.8
```

Format String Component

The optional *formatString* component is a format string that is appropriate for the type of object being formatted. Specify a standard or custom numeric format string if the corresponding object is a numeric value, a standard or custom date and time format string if the corresponding object is a DateTime object, or an enumeration format string if the corresponding object is an enumeration value. If *formatString* is not specified, the general ("G") format specifier for a numeric, date and time, or enumeration type is used. The colon is required if *formatString* is specified.

The following table lists types or categories of types in the .NET Framework class library that support a predefined set of format strings, and provides links to the topics that list the supported format strings. Note that string formatting is an extensible mechanism that makes it possible to define new format strings for all existing types as well as to define a set of format strings supported by an application-defined type. For more information, see the IFormattable and ICustomFormatter interface topics.

TYPE OR TYPE CATEGORY	SEE
Date and time types (DateTime, DateTimeOffset)	Standard Date and Time Format Strings Custom Date and Time Format Strings
Enumeration types (all types derived from System.Enum)	Enumeration Format Strings
Numeric types (BigInteger, Byte, Decimal, Double, Int16, Int32, Int64, SByte, Single, UInt16, UInt32, UInt64)	Standard Numeric Format Strings Custom Numeric Format Strings
Guid	Guid.ToString(String)
TimeSpan	Standard TimeSpan Format Strings Custom TimeSpan Format Strings

Escaping Braces

Opening and closing braces are interpreted as starting and ending a format item. Consequently, you must use an escape sequence to display a literal opening brace or closing brace. Specify two opening braces ("{{") in the fixed text to display one opening brace ("{"), or two closing braces ("}}") to display one closing brace ("}"). Braces in a format item are interpreted sequentially in the order they are encountered. Interpreting nested braces is not supported.

The way escaped braces are interpreted can lead to unexpected results. For example, consider the format item "{{{0:D}}}", which is intended to display an opening brace, a numeric value formatted as a decimal number, and a closing brace. However, the format item is actually interpreted in the following manner:

1. The first two opening braces ("{{") are escaped and yield one opening brace.
2. The next three characters ("{0:") are interpreted as the start of a format item.
3. The next character ("D") would be interpreted as the Decimal standard numeric format specifier, but the next two escaped braces ("}}") yield a single brace. Because the resulting string ("D}") is not a standard numeric format specifier, the resulting string is interpreted as a custom format string that means display the literal string "D}".
4. The last brace ("}") is interpreted as the end of the format item.
5. The final result that is displayed is the literal string, "{D}". The numeric value that was to be formatted is not displayed.

One way to write your code to avoid misinterpreting escaped braces and format items is to format the braces and format item separately. That is, in the first format operation display a literal opening brace, in the next operation display the result of the format item, then in the final operation display a literal closing brace. The following example illustrates this approach.

```
int value = 6324;
string output = string.Format("{0}{1:D}{2}",
                             "{", value, "}");
Console.WriteLine(output);
// The example displays the following output:
//       {6324}
```

```
Dim value As Integer = 6324
Dim output As String = String.Format("{0}{1:D}{2}", _
                                     "{", value, "}")
Console.WriteLine(output)
' The example displays the following output:
'       {6324}
```

Processing Order

If the call to the composite formatting method includes an IFormatProvider argument whose value is not `null`, the runtime calls its IFormatProvider.GetFormat method to request an ICustomFormatter implementation. If the method is able to return an ICustomFormatter implementation, it's cached for the duration of the call of the composite formatting method.

Each value in the parameter list that corresponds to a format item is converted to a string as follows:

1. If the value to be formatted is `null`, an empty string String.Empty is returned.

2. If an ICustomFormatter implementation is available, the runtime calls its Format method. It passes the method the format item's *formatString* value, if one is present, or `null` if it's not, along with the IFormatProvider implementation. If the call to the ICustomFormatter.Format method returns `null`, execution proceeds to the next step; otherwise, the result of the ICustomFormatter.Format call is returned.

3. If the value implements the IFormattable interface, the interface's ToString(String, IFormatProvider) method is called. The method is passed the *formatString* value, if one is present in the format item, or `null` if it's not. The IFormatProvider argument is determined as follows:

   - For a numeric value, if a composite formatting method with a non-null IFormatProvider argument is called, the runtime requests a NumberFormatInfo object from its IFormatProvider.GetFormat method. If it's unable to supply one, if the value of the argument is `null`, or if the composite formatting method doesn't have an IFormatProvider parameter, the NumberFormatInfo object for the current thread culture is used.

   - For a date and time value, if a composite formatting method with a non-null IFormatProvider argument is called, the runtime requests a DateTimeFormatInfo object from its IFormatProvider.GetFormat method. If it's unable to supply one, if the value of the argument is `null`, or if the composite formatting method doesn't have an IFormatProvider parameter, the DateTimeFormatInfo object for the current thread culture is used.

   - For objects of other types, if a composite formatting method is called with an IFormatProvider argument, its value is passed directly to the IFormattable.ToString implementation. Otherwise, `null` is passed to the IFormattable.ToString implementation.

4. The type's parameterless `ToString` method, which either overrides Object.ToString() or inherits the behavior of its base class, is called. In this case, the format string specified by the *formatString* component in the format item, if it's present, is ignored.

Alignment is applied after the preceding steps have been performed.

Code Examples

The following example shows one string created using composite formatting and another created using an object's `ToString` method. Both types of formatting produce equivalent results.

```
string FormatString1 = String.Format("{0:dddd MMMM}", DateTime.Now);
string FormatString2 = DateTime.Now.ToString("dddd MMMM");
```

```
Dim FormatString1 As String = String.Format("{0:dddd MMMM}", DateTime.Now)
Dim FormatString2 As String = DateTime.Now.ToString("dddd MMMM")
```

Assuming that the current day is a Thursday in May, the value of both strings in the preceding example is `Thursday May` in the U.S. English culture.

Console.WriteLine exposes the same functionality as String.Format. The only difference between the two methods is that String.Format returns its result as a string, while Console.WriteLine writes the result to the output stream associated with the Console object. The following example uses the Console.WriteLine method to format the value of `MyInt` to a currency value.

```
int MyInt = 100;
Console.WriteLine("{0:C}", MyInt);
// The example displays the following output
// if en-US is the current culture:
//        $100.00
```

```
Dim MyInt As Integer = 100
Console.WriteLine("{0:C}", MyInt)
' The example displays the following output
' if en-US is the current culture:
'        $100.00
```

The following example demonstrates formatting multiple objects, including formatting one object two different ways.

```
string myName = "Fred";
Console.WriteLine(String.Format("Name = {0}, hours = {1:hh}, minutes = {1:mm}",
      myName, DateTime.Now));
// Depending on the current time, the example displays output like the following:
//    Name = Fred, hours = 11, minutes = 30
```

```
Dim myName As String = "Fred"
Console.WriteLine(String.Format("Name = {0}, hours = {1:hh}, minutes = {1:mm}", _
                  myName, DateTime.Now))
' Depending on the current time, the example displays output like the following:
'    Name = Fred, hours = 11, minutes = 30
```

The following example demonstrates the use of alignment in formatting. The arguments that are formatted are placed between vertical bar characters (|) to highlight the resulting alignment.

```
string myFName = "Fred";
string myLName = "Opals";
int myInt = 100;
string FormatFName = String.Format("First Name = |{0,10}|", myFName);
string FormatLName = String.Format("Last Name = |{0,10}|", myLName);
string FormatPrice = String.Format("Price = |{0,10:C}|", myInt);
Console.WriteLine(FormatFName);
Console.WriteLine(FormatLName);
Console.WriteLine(FormatPrice);
Console.WriteLine();

FormatFName = String.Format("First Name = |{0,-10}|", myFName);
FormatLName = String.Format("Last Name = |{0,-10}|", myLName);
FormatPrice = String.Format("Price = |{0,-10:C}|", myInt);
Console.WriteLine(FormatFName);
Console.WriteLine(FormatLName);
Console.WriteLine(FormatPrice);
// The example displays the following output on a system whose current
// culture is en-US:
//          First Name = |      Fred|
//          Last Name = |     Opals|
//          Price = |   $100.00|
//
//          First Name = |Fred      |
//          Last Name = |Opals     |
//          Price = |$100.00   |
```

```
Dim myFName As String = "Fred"
Dim myLName As String = "Opals"

Dim myInt As Integer = 100
Dim FormatFName As String = String.Format("First Name = |{0,10}|", myFName)
Dim FormatLName As String = String.Format("Last Name = |{0,10}|", myLName)
Dim FormatPrice As String = String.Format("Price = |{0,10:C}|", myInt)
Console.WriteLine(FormatFName)
Console.WriteLine(FormatLName)
Console.WriteLine(FormatPrice)
Console.WriteLine()

FormatFName = String.Format("First Name = |{0,-10}|", myFName)
FormatLName = String.Format("Last Name = |{0,-10}|", myLName)
FormatPrice = String.Format("Price = |{0,-10:C}|", myInt)
Console.WriteLine(FormatFName)
Console.WriteLine(FormatLName)
Console.WriteLine(FormatPrice)
' The example displays the following output on a system whose current
' culture is en-US:
'          First Name = |      Fred|
'          Last Name = |     Opals|
'          Price = |   $100.00|
'
'          First Name = |Fred      |
'          Last Name = |Opals     |
'          Price = |$100.00   |
```

See also

- WriteLine
- String.Format
- String interpolation (C#)
- String interpolation (Visual Basic)
- Formatting Types

- Standard Numeric Format Strings
- Custom Numeric Format Strings
- Standard Date and Time Format Strings
- Custom Date and Time Format Strings
- Standard TimeSpan Format Strings
- Custom TimeSpan Format Strings
- Enumeration Format Strings

Performing Formatting Operations

9/6/2018 • 2 minutes to read • Edit Online

The following topics provide step-by-step instructions for performing specific formatting operations.

- How to: Pad a Number with Leading Zeros
- How to: Define and Use Custom Numeric Format Providers
- How to: Convert Numeric User Input in Web Controls to Numbers
- How to: Extract the Day of the Week from a Specific Date.
- How to: Round-trip Date and Time Values
- How to: Display Localized Date and Time Information to Web Users
- How to: Display Milliseconds in Date and Time Values
- How to: Display Dates in Non-Gregorian Calendars

See also

- Formatting Types

How to: Pad a Number with Leading Zeros

9/6/2018 • 6 minutes to read • Edit Online

You can add leading zeros to an integer by using the "D" standard numeric format string with a precision specifier. You can add leading zeros to both integer and floating-point numbers by using a custom numeric format string. This topic shows how to use both methods to pad a number with leading zeros.

To pad an integer with leading zeros to a specific length

1. Determine the minimum number of digits you want the integer value to display. Include any leading digits in this number.

2. Determine whether you want to display the integer as a decimal value or a hexadecimal value.

   - To display the integer as a decimal value, call its `ToString(String)` method, and pass the string "D*n*" as the value of the `format` parameter, where *n* represents the minimum length of the string.

   - To display the integer as a hexadecimal value, call its `ToString(String)` method and pass the string "X*n*" as the value of the `format` parameter, where *n* represents the minimum length of the string.

   You can also use the format string in a method, such as Format or WriteLine, that uses composite formatting.

The following example formats several integer values with leading zeros so that the total length of the formatted number is at least eight characters.

```csharp
byte byteValue = 254;
short shortValue = 10342;
int intValue = 1023983;
long lngValue = 6985321;
ulong ulngValue = UInt64.MaxValue;

// Display integer values by calling the ToString method.
Console.WriteLine("{0,22} {1,22}", byteValue.ToString("D8"), byteValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", shortValue.ToString("D8"), shortValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", intValue.ToString("D8"), intValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", lngValue.ToString("D8"), lngValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", ulngValue.ToString("D8"), ulngValue.ToString("X8"));
Console.WriteLine();

// Display the same integer values by using composite formatting.
Console.WriteLine("{0,22:D8} {0,22:X8}", byteValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", shortValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", intValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", lngValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", ulngValue);
// The example displays the following output:
//                     00000254               000000FE
//                     00010342               00002866
//                     01023983               000F9FEF
//                     06985321               006A9669
//         18446744073709551615       FFFFFFFFFFFFFFFF
//
//                     00000254               000000FE
//                     00010342               00002866
//                     01023983               000F9FEF
//                     06985321               006A9669
//         18446744073709551615       FFFFFFFFFFFFFFFF
//         18446744073709551615       FFFFFFFFFFFFFFFF
```

```
Dim byteValue As Byte = 254
Dim shortValue As Short = 10342
Dim intValue As Integer = 1023983
Dim lngValue As Long = 6985321
Dim ulngValue As ULong = UInt64.MaxValue

' Display integer values by calling the ToString method.
Console.WriteLine("{0,22} {1,22}", byteValue.ToString("D8"), byteValue.ToString("X8"))
Console.WriteLine("{0,22} {1,22}", shortValue.ToString("D8"), shortValue.ToString("X8"))
Console.WriteLine("{0,22} {1,22}", intValue.ToString("D8"), intValue.ToString("X8"))
Console.WriteLine("{0,22} {1,22}", lngValue.ToString("D8"), lngValue.ToString("X8"))
Console.WriteLine("{0,22} {1,22}", ulngValue.ToString("D8"), ulngValue.ToString("X8"))
Console.WriteLine()

' Display the same integer values by using composite formatting.
Console.WriteLine("{0,22:D8} {0,22:X8}", byteValue)
Console.WriteLine("{0,22:D8} {0,22:X8}", shortValue)
Console.WriteLine("{0,22:D8} {0,22:X8}", intValue)
Console.WriteLine("{0,22:D8} {0,22:X8}", lngValue)
Console.WriteLine("{0,22:D8} {0,22:X8}", ulngValue)
' The example displays the following output:
'                     00000254               000000FE
'                     00010342               00002866
'                     01023983               000F9FEF
'                     06985321               006A9669
'         18446744073709551615       FFFFFFFFFFFFFFFF
'
'                     00000254               000000FE
'                     00010342               00002866
'                     01023983               000F9FEF
'                     06985321               006A9669
'         18446744073709551615       FFFFFFFFFFFFFFFF
```

To pad an integer with a specific number of leading zeros

1. Determine how many leading zeros you want the integer value to display.

2. Determine whether you want to display the integer as a decimal value or a hexadecimal value. Formatting it as a decimal value requires that you use the "D" standard format specifier; formatting it as a hexadecimal value requires that you use the "X" standard format specifier.

3. Determine the length of the unpadded numeric string by calling the integer value's `ToString("D").Length` or `ToString("X").Length` method.

4. Add the number of leading zeros that you want to include in the formatted string to the length of the unpadded numeric string. This defines the total length of the padded string.

5. Call the integer value's `ToString(String)` method, and pass the string "D*n*" for decimal strings and "X*n*" for hexadecimal strings, where *n* represents the total length of the padded string. You can also use the "D*n*" or "X*n*" format string in a method that supports composite formatting.

The following example pads an integer value with five leading zeros.

```
int value = 160934;
int decimalLength = value.ToString("D").Length + 5;
int hexLength = value.ToString("X").Length + 5;
Console.WriteLine(value.ToString("D" + decimalLength.ToString()));
Console.WriteLine(value.ToString("X" + hexLength.ToString()));
// The example displays the following output:
//       00000160934
//       00000274A6
```

```
Dim value As Integer = 160934
Dim decimalLength As Integer = value.ToString("D").Length + 5
Dim hexLength As Integer = value.ToString("X").Length + 5
Console.WriteLine(value.ToString("D" + decimalLength.ToString()))
Console.WriteLine(value.ToString("X" + hexLength.ToString()))
' The example displays the following output:
'       00000160934
'       00000274A6
```

To pad a numeric value with leading zeros to a specific length

1. Determine how many digits to the left of the decimal you want the string representation of the number to have. Include any leading zeros in this total number of digits.

2. Define a custom numeric format string that uses the zero placeholder ("0") to represent the minimum number of zeros.

3. Call the number's `ToString(String)` method and pass it the custom format string. You can also use the custom format string with a method that supports composite formatting.

The following example formats several numeric values with leading zeros so that the total length of the formatted number is at least eight digits to the left of the decimal.

```
string fmt = "00000000.##";
int intValue = 1053240;
decimal decValue = 103932.52m;
float sngValue = 1549230.10873992f;
double dblValue = 9034521202.93217412;

// Display the numbers using the ToString method.
Console.WriteLine(intValue.ToString(fmt));
Console.WriteLine(decValue.ToString(fmt));
Console.WriteLine(sngValue.ToString(fmt));
Console.WriteLine(dblValue.ToString(fmt));
Console.WriteLine();

// Display the numbers using composite formatting.
string formatString = " {0,15:" + fmt + "}";
Console.WriteLine(formatString, intValue);
Console.WriteLine(formatString, decValue);
Console.WriteLine(formatString, sngValue);
Console.WriteLine(formatString, dblValue);
// The example displays the following output:
//       01053240
//       00103932.52
//       01549230
//       9034521202.93
//
//               01053240
//            00103932.52
//               01549230
//          9034521202.93
```

```
Dim fmt As String = "00000000.##"
Dim intValue As Integer = 1053240
Dim decValue As Decimal = 103932.52d
Dim sngValue As Single = 1549230.10873992
Dim dblValue As Double = 9034521202.93217412

' Display the numbers using the ToString method.
Console.WriteLine(intValue.ToString(fmt))
Console.WriteLine(decValue.ToString(fmt))
Console.WriteLine(sngValue.ToString(fmt))
Console.WriteLine(dblValue.ToString(fmt))
Console.WriteLine()

' Display the numbers using composite formatting.
Dim formatString As String = " {0,15:" + fmt + "}"
Console.WriteLine(formatString, intValue)
Console.WriteLine(formatString, decValue)
Console.WriteLine(formatString, sngValue)
Console.WriteLine(formatString, dblValue)
' The example displays the following output:
'       01053240
'       00103932.52
'       01549230
'       9034521202.93
'
'               01053240
'            00103932.52
'               01549230
'          9034521202.93
```

To pad a numeric value with a specific number of leading zeros

1. Determine how many leading zeros you want the numeric value to have.

2. Determine the number of digits to the left of the decimal in the unpadded numeric string. To do this:

   a. Determine whether the string representation of a number includes a decimal point symbol.

   b. If it does include a decimal point symbol, determine the number of characters to the left of the decimal point.

      -or-

      If it does not include a decimal point symbol, determine the string's length.

3. Create a custom format string that uses the zero placeholder ("0") for each of the leading zeros to appear in the string, and that uses either the zero placeholder or the digit placeholder ("#") to represent each digit in the default string.

4. Supply the custom format string as a parameter either to the number's `ToString(String)` method or to a method that supports composite formatting.

The following example pads two Double values with five leading zeros.

```csharp
double[] dblValues = { 9034521202.93217412, 9034521202 };
foreach (double dblValue in dblValues)
{
   string decSeparator = System.Globalization.NumberFormatInfo.CurrentInfo.NumberDecimalSeparator;
   string fmt, formatString;

   if (dblValue.ToString().Contains(decSeparator))
   {
      int digits = dblValue.ToString().IndexOf(decSeparator);
      fmt = new String('0', 5) + new String('#', digits) + ".##";
   }
   else
   {
      fmt = new String('0', dblValue.ToString().Length);
   }
   formatString = "{0,20:" + fmt + "}";

   Console.WriteLine(dblValue.ToString(fmt));
   Console.WriteLine(formatString, dblValue);
}
// The example displays the following output:
//       000009034521202.93
//         000009034521202.93
//       9034521202
//                 9034521202
```

```vb
Dim dblValues() As Double = { 9034521202.93217412, 9034521202 }
For Each dblValue As Double In dblValues
   Dim decSeparator As String = System.Globalization.NumberFormatInfo.CurrentInfo.NumberDecimalSeparator
   Dim fmt, formatString As String

   If dblValue.ToString.Contains(decSeparator) Then
      Dim digits As Integer = dblValue.ToString().IndexOf(decSeparator)
      fmt = New String("0"c, 5) + New String("#"c, digits) + ".##"
   Else
      fmt = New String("0"c, dblValue.ToString.Length)
   End If
   formatString = "{0,20:" + fmt + "}"

   Console.WriteLine(dblValue.ToString(fmt))
   Console.WriteLine(formatString, dblValue)
Next
' The example displays the following output:
'       000009034521202.93
'         000009034521202.93
'       9034521202
'                 9034521202
```

See also

- Custom Numeric Format Strings
- Standard Numeric Format Strings
- Composite Formatting

How to: Extract the Day of the Week from a Specific Date

9/6/2018 • 7 minutes to read • Edit Online

The .NET Framework makes it easy to determine the ordinal day of the week for a particular date, and to display the localized weekday name for a particular date. An enumerated value that indicates the day of the week corresponding to a particular date is available from the DayOfWeek or DayOfWeek property. In contrast, retrieving the weekday name is a formatting operation that can be performed by calling a formatting method, such as a date and time value's `ToString` method or the String.Format method. This topic shows how to perform these formatting operations.

To extract a number indicating the day of the week from a specific date

1. If you are working with the string representation of a date, convert it to a DateTime or a DateTimeOffset value by using the static DateTime.Parse or DateTimeOffset.Parse method.

2. Use the DateTime.DayOfWeek or DateTimeOffset.DayOfWeek property to retrieve a DayOfWeek value that indicates the day of the week.

3. If necessary, cast (in C#) or convert (in Visual Basic) the DayOfWeek value to an integer.

The following example displays an integer that represents the day of the week of a specific date.

```
using System;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine((int) dateValue.DayOfWeek);
   }
}
// The example displays the following output:
//       3
```

```
Module Example
   Public Sub Main()
      Dim dateValue As Date = #6/11/2008#
      Console.WriteLine(dateValue.DayOfWeek)
   End Sub
End Module
' The example displays the following output:
'    3
```

To extract the abbreviated weekday name from a specific date

1. If you are working with the string representation of a date, convert it to a DateTime or a DateTimeOffset value by using the static DateTime.Parse or DateTimeOffset.Parse method.

2. You can extract the abbreviated weekday name of the current culture or of a specific culture:

   a. To extract the abbreviated weekday name for the current culture, call the date and time value's DateTime.ToString(String) or DateTimeOffset.ToString(String) instance method, and pass the string "ddd" as the `format` parameter. The following example illustrates the call to the ToString(String) method.

```
using System;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine(dateValue.ToString("ddd"));
   }
}
// The example displays the following output:
 //       Wed
```

```
Module Example
   Public Sub Main()
      Dim dateValue As Date = #6/11/2008#
      Console.WriteLine(dateValue.ToString("ddd"))
   End Sub
End Module
' The example displays the following output:
'       Wed
```

b. To extract the abbreviated weekday name for a specific culture, call the date and time value's DateTime.ToString(String, IFormatProvider) or DateTimeOffset.ToString(String, IFormatProvider) instance method. Pass the string "ddd" as the `format` parameter. Pass either a CultureInfo or a DateTimeFormatInfo object that represents the culture whose weekday name you want to retrieve as the `provider` parameter. The following code illustrates a call to the ToString(String, IFormatProvider) method using a CultureInfo object that represents the fr-FR culture.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine(dateValue.ToString("ddd",
                        new CultureInfo("fr-FR")));
   }
}
// The example displays the following output:
//       mer.
```

```
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim dateValue As Date = #6/11/2008#
      Console.WriteLine(dateValue.ToString("ddd",
                        New CultureInfo("fr-FR")))
   End Sub
End Module
' The example displays the following output:
'       mer.
```

To extract the full weekday name from a specific date

1. If you are working with the string representation of a date, convert it to a DateTime or a DateTimeOffset value by using the static DateTime.Parse or DateTimeOffset.Parse method.

2. You can extract the full weekday name of the current culture or of a specific culture:

   a. To extract the weekday name for the current culture, call the date and time value's DateTime.ToString(String) or DateTimeOffset.ToString(String) instance method, and pass the string "dddd" as the `format` parameter. The following example illustrates the call to the ToString(String) method.

```
using System;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine(dateValue.ToString("dddd"));
   }
}
// The example displays the following output:
//       Wednesday
```

```
Module Example
   Public Sub Main()
      Dim dateValue As Date = #6/11/2008#
      Console.WriteLine(dateValue.ToString("dddd"))
   End Sub
End Module
' The example displays the following output:
'       Wednesday
```

   b. To extract the weekday name for a specific culture, call the date and time value's DateTime.ToString(String, IFormatProvider) or DateTimeOffset.ToString(String, IFormatProvider) instance method. Pass the string "dddd" as the `format` parameter. Pass either a CultureInfo or a DateTimeFormatInfo object that represents the culture whose weekday name you want to retrieve as the `provider` parameter. The following code illustrates a call to the ToString(String, IFormatProvider) method using a CultureInfo object that represents the es-ES culture.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine(dateValue.ToString("dddd",
                        new CultureInfo("es-ES")));
   }
}
// The example displays the following output:
//       miércoles.
```

```
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim dateValue As Date = #6/11/2008#
      Console.WriteLine(dateValue.ToString("dddd", _
                        New CultureInfo("es-ES")))
   End Sub
End Module
' The example displays the following output:
'       miércoles.
```

Example

The example illustrates calls to the DateTime.DayOfWeek and DateTimeOffset.DayOfWeek properties and the DateTime.ToString and DateTimeOffset.ToString methods to retrieve the number that represents the day of the week, the abbreviated weekday name, and the full weekday name for a particular date.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string dateString = "6/11/2007";
      DateTime dateValue;
      DateTimeOffset dateOffsetValue;

      try
      {
         DateTimeFormatInfo dateTimeFormats;
         // Convert date representation to a date value
         dateValue = DateTime.Parse(dateString, CultureInfo.InvariantCulture);
         dateOffsetValue = new DateTimeOffset(dateValue,
                                     TimeZoneInfo.Local.GetUtcOffset(dateValue));

         // Convert date representation to a number indicating the day of week
         Console.WriteLine((int) dateValue.DayOfWeek);
         Console.WriteLine((int) dateOffsetValue.DayOfWeek);

         // Display abbreviated weekday name using current culture
         Console.WriteLine(dateValue.ToString("ddd"));
         Console.WriteLine(dateOffsetValue.ToString("ddd"));

         // Display full weekday name using current culture
         Console.WriteLine(dateValue.ToString("dddd"));
         Console.WriteLine(dateOffsetValue.ToString("dddd"));

         // Display abbreviated weekday name for de-DE culture
         Console.WriteLine(dateValue.ToString("ddd", new CultureInfo("de-DE")));
         Console.WriteLine(dateOffsetValue.ToString("ddd",
                                                    new CultureInfo("de-DE")));

         // Display abbreviated weekday name with de-DE DateTimeFormatInfo object
         dateTimeFormats = new CultureInfo("de-DE").DateTimeFormat;
         Console.WriteLine(dateValue.ToString("ddd", dateTimeFormats));
         Console.WriteLine(dateOffsetValue.ToString("ddd", dateTimeFormats));

         // Display full weekday name for fr-FR culture
         Console.WriteLine(dateValue.ToString("ddd", new CultureInfo("fr-FR")));
         Console.WriteLine(dateOffsetValue.ToString("ddd",
                                                   new CultureInfo("fr-FR")));
```

```
         // Display abbreviated weekday name with fr-FR DateTimeFormatInfo object
         dateTimeFormats = new CultureInfo("fr-FR").DateTimeFormat;
         Console.WriteLine(dateValue.ToString("dddd", dateTimeFormats));
         Console.WriteLine(dateOffsetValue.ToString("dddd", dateTimeFormats));
      }
      catch (FormatException)
      {
         Console.WriteLine("Unable to convert {0} to a date.", dateString);
      }
   }
}
// The example displays the following output:
//       1
//       1
//       Mon
//       Mon
//       Monday
//       Monday
//       Mo
//       Mo
//       Mo
//       Mo
//       lun.
//       lun.
//       lundi
//       lundi
```

```
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim dateString As String = "6/11/2007"
      Dim dateValue As Date
      Dim dateOffsetValue As DateTimeOffset

      Try
         Dim dateTimeFormats As DateTimeFormatInfo
         ' Convert date representation to a date value
         dateValue = Date.Parse(dateString, CultureInfo.InvariantCulture)
         dateOffsetValue = New DateTimeOffset(dateValue, _
                                    TimeZoneInfo.Local.GetUtcOffset(dateValue))
         ' Convert date representation to a number indicating the day of week
         Console.WriteLine(dateValue.DayOfWeek)
         Console.WriteLine(dateOffsetValue.DayOfWeek)

         ' Display abbreviated weekday name using current culture
         Console.WriteLine(dateValue.ToString("ddd"))
         Console.WriteLine(dateOffsetValue.ToString("ddd"))

         ' Display full weekday name using current culture
         Console.WriteLine(dateValue.ToString("dddd"))
         Console.WriteLine(dateOffsetValue.ToString("dddd"))

         ' Display abbreviated weekday name for de-DE culture
         Console.WriteLine(dateValue.ToString("ddd", New CultureInfo("de-DE")))
         Console.WriteLine(dateOffsetValue.ToString("ddd", _
                                                    New CultureInfo("de-DE")))

         ' Display abbreviated weekday name with de-DE DateTimeFormatInfo object
         dateTimeFormats = New CultureInfo("de-DE").DateTimeFormat
         Console.WriteLine(dateValue.ToString("ddd", dateTimeFormats))
         Console.WriteLine(dateOffsetValue.ToString("ddd", dateTimeFormats))

         ' Display full weekday name for fr-FR culture
         Console.WriteLine(dateValue.ToString("ddd", New CultureInfo("fr-FR")))
         Console.WriteLine(dateOffsetValue.ToString("ddd", _
                                                    New CultureInfo("fr-FR")))

         ' Display abbreviated weekday name with fr-FR DateTimeFormatInfo object
         dateTimeFormats = New CultureInfo("fr-FR").DateTimeFormat
         Console.WriteLine(dateValue.ToString("dddd", dateTimeFormats))
         Console.WriteLine(dateOffsetValue.ToString("dddd", dateTimeFormats))
      Catch e As FormatException
         Console.WriteLine("Unable to convert {0} to a date.", dateString)
      End Try
   End Sub
End Module
' The example displays the following output to the console:
'       1
'       1
'       Mon
'       Mon
'       Monday
'       Monday
'       Mo
'       Mo
'       Mo
'       Mo
'       lun.
'       lun.
'       lundi
'       lundi
```

Individual languages may provide functionality that duplicates or supplements the functionality provided by the .NET Framework. For example, Visual Basic includes two such functions:

- `Weekday`, which returns a number that indicates the day of the week of a particular date. It considers the ordinal value of the first day of the week to be one, whereas the DateTime.DayOfWeek property considers it to be zero.
- `WeekdayName`, which returns the name of the week in the current culture that corresponds to a particular weekday number.

The following example illustrates the use of the Visual Basic `Weekday` and `WeekdayName` functions.

```
Imports System.Globalization
Imports System.Threading

Module Example
   Public Sub Main()
      Dim dateValue As Date = #6/11/2008#

      ' Get weekday number using Visual Basic Weekday function
      Console.WriteLine(Weekday(dateValue))                 ' Displays 4
      ' Compare with .NET DateTime.DayOfWeek property
      Console.WriteLine(dateValue.DayOfWeek)                ' Displays 3

      ' Get weekday name using Weekday and WeekdayName functions
      Console.WriteLine(WeekdayName(Weekday(dateValue)))    ' Displays Wednesday

      ' Change culture to de-DE
      Dim originalCulture As CultureInfo = Thread.CurrentThread.CurrentCulture
      Thread.CurrentThread.CurrentCulture = New CultureInfo("de-DE")
      ' Get weekday name using Weekday and WeekdayName functions
      Console.WriteLine(WeekdayName(Weekday(dateValue)))   ' Displays Donnerstag

      ' Restore original culture
      Thread.CurrentThread.CurrentCulture = originalCulture
   End Sub
End Module
```

You can also use the value returned by the DateTime.DayOfWeek property to retrieve the weekday name of a particular date. This requires only a call to the ToString method on the DayOfWeek value returned by the property. However, this technique does not produce a localized weekday name for the current culture, as the following example illustrates.

```csharp
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Change current culture to fr-FR
      CultureInfo originalCulture = Thread.CurrentThread.CurrentCulture;
      Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");

      DateTime dateValue = new DateTime(2008, 6, 11);
      // Display the DayOfWeek string representation
      Console.WriteLine(dateValue.DayOfWeek.ToString());
      // Restore original current culture
      Thread.CurrentThread.CurrentCulture = originalCulture;
   }
}
// The example displays the following output:
//       Wednesday
```

```vb
Imports System.Globalization
Imports System.Threading

Module Example
   Public Sub Main()
      ' Change current culture to fr-FR
      Dim originalCulture As CultureInfo = Thread.CurrentThread.CurrentCulture
      Thread.CurrentThread.CurrentCulture = New CultureInfo("fr-FR")

      Dim dateValue As Date = #6/11/2008#
      ' Display the DayOfWeek string representation
      Console.WriteLine(dateValue.DayOfWeek.ToString())
      ' Restore original current culture
      Thread.CurrentThread.CurrentCulture = originalCulture
   End Sub
End Module
' The example displays the following output:
'       Wednesday
```

See also

- Performing Formatting Operations
- Standard Date and Time Format Strings
- Custom Date and Time Format Strings

How to: Define and Use Custom Numeric Format Providers

9/6/2018 • 7 minutes to read • Edit Online

The .NET Framework gives you extensive control over the string representation of numeric values. It supports the following features for customizing the format of numeric values:

- Standard numeric format strings, which provide a predefined set of formats for converting numbers to their string representation. You can use them with any numeric formatting method, such as Decimal.ToString(String), that has a `format` parameter. For details, see Standard Numeric Format Strings.
- Custom numeric format strings, which provide a set of symbols that can be combined to define custom numeric format specifiers. They can also be used with any numeric formatting method, such as Decimal.ToString(String), that has a `format` parameter. For details, see Custom Numeric Format Strings.
- Custom CultureInfo or NumberFormatInfo objects, which define the symbols and format patterns used in displaying the string representations of numeric values. You can use them with any numeric formatting method, such as ToString, that has a `provider` parameter. Typically, the `provider` parameter is used to specify culture-specific formatting.

In some cases (such as when an application must display a formatted account number, an identification number, or a postal code) these three techniques are inappropriate. The .NET Framework also enables you to define a formatting object that is neither a CultureInfo nor a NumberFormatInfo object to determine how a numeric value is formatted. This topic provides the step-by-step instructions for implementing such an object, and provides an example that formats telephone numbers.

To define a custom format provider

1. Define a class that implements the IFormatProvider and ICustomFormatter interfaces.
2. Implement the IFormatProvider.GetFormat method. GetFormat is a callback method that the formatting method (such as the String.Format(IFormatProvider, String, Object[]) method) invokes to retrieve the object that is actually responsible for performing custom formatting. A typical implementation of GetFormat does the following:

   a. Determines whether the Type object passed as a method parameter represents an ICustomFormatter interface.

   b. If the parameter does represent the ICustomFormatter interface, GetFormat returns an object that implements the ICustomFormatter interface that is responsible for providing custom formatting. Typically, the custom formatting object returns itself.

   c. If the parameter does not represent the ICustomFormatter interface, GetFormat returns `null`.
3. Implement the Format method. This method is called by the String.Format(IFormatProvider, String, Object[]) method and is responsible for returning the string representation of a number. Implementing the method typically involves the following:

   a. Optionally, make sure that the method is legitimately intended to provide formatting services by examining the `provider` parameter. For formatting objects that implement both IFormatProvider and ICustomFormatter, this involves testing the `provider` parameter for equality with the current formatting object.

b. Determine whether the formatting object should support custom format specifiers. (For example, an "N" format specifier might indicate that a U.S. telephone number should be output in NANP format, and an "I" might indicate output in ITU-T Recommendation E.123 format.) If format specifiers are used, the method should handle the specific format specifier. It is passed to the method in the `format` parameter. If no specifier is present, the value of the `format` parameter is String.Empty.

c. Retrieve the numeric value passed to the method as the `arg` parameter. Perform whatever manipulations are required to convert it to its string representation.

d. Return the string representation of the `arg` parameter.

To use a custom numeric formatting object

1. Create a new instance of the custom formatting class.

2. Call the String.Format(IFormatProvider, String, Object[]) formatting method, passing it the custom formatting object, the formatting specifier (or String.Empty, if one is not used), and the numeric value to be formatted.

Example

The following example defines a custom numeric format provider named `TelephoneFormatter` that converts a number that represents a U.S. telephone number to its NANP or E.123 format. The method handles two format specifiers, "N" (which outputs the NANP format) and "I" (which outputs the international E.123 format).

```
using System;
using System.Globalization;

public class TelephoneFormatter : IFormatProvider, ICustomFormatter
{
   public object GetFormat(Type formatType)
   {
      if (formatType == typeof(ICustomFormatter))
         return this;
      else
         return null;
   }

   public string Format(string format, object arg, IFormatProvider formatProvider)
   {
      // Check whether this is an appropriate callback
      if (! this.Equals(formatProvider))
         return null;

      // Set default format specifier
      if (string.IsNullOrEmpty(format))
         format = "N";

      string numericString = arg.ToString();

      if (format == "N")
      {
         if (numericString.Length <= 4)
            return numericString;
         else if (numericString.Length == 7)
            return numericString.Substring(0, 3) + "-" + numericString.Substring(3, 4);
         else if (numericString.Length == 10)
              return "(" + numericString.Substring(0, 3) + ") " +
                     numericString.Substring(3, 3) + "-" + numericString.Substring(6);
         else
            throw new FormatException(
                     string.Format("'{0}' cannot be used to format {1}.",
                                   format, arg.ToString()));
      }
```

```
      else if (format == "I")
      {
         if (numericString.Length < 10)
            throw new FormatException(string.Format("{0} does not have 10 digits.", arg.ToString()));
         else
            numericString = "+1 " + numericString.Substring(0, 3) + " " + numericString.Substring(3, 3) + " "
+ numericString.Substring(6);
      }
      else
      {
         throw new FormatException(string.Format("The {0} format specifier is invalid.", format));
      }
      return numericString;
   }
}

public class TestTelephoneFormatter
{
   public static void Main()
   {
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}", 0));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}", 911));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}", 8490216));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}", 4257884748));

      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}", 0));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}", 911));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}", 8490216));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}", 4257884748));

      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:I}", 4257884748));
   }
}
```

```vb
Public Class TelephoneFormatter : Implements IFormatProvider, ICustomFormatter
   Public Function GetFormat(formatType As Type) As Object _
                   Implements IFormatProvider.GetFormat
      If formatType Is GetType(ICustomFormatter) Then
         Return Me
      Else
         Return Nothing
      End If
   End Function

   Public Function Format(fmt As String, arg As Object, _
                          formatProvider As IFormatProvider) As String _
                   Implements ICustomFormatter.Format
      ' Check whether this is an appropriate callback
      If Not Me.Equals(formatProvider) Then Return Nothing

      ' Set default format specifier
      If String.IsNullOrEmpty(fmt) Then fmt = "N"

      Dim numericString As String = arg.ToString

      If fmt = "N" Then
         Select Case numericString.Length
            Case <= 4
               Return numericString
            Case 7
               Return Left(numericString, 3) & "-" & Mid(numericString, 4)
            Case 10
               Return "(" & Left(numericString, 3) & ") " & _
                      Mid(numericString, 4, 3) & "-" & Mid(numericString, 7)
            Case Else
               Throw New FormatException( _
                         String.Format("'{0}' cannot be used to format {1}.", _
                                       fmt, arg.ToString()))
         End Select
      ElseIf fmt = "I" Then
         If numericString.Length < 10 Then
            Throw New FormatException(String.Format("{0} does not have 10 digits.", arg.ToString()))
         Else
            numericString = "+1 " & Left(numericString, 3) & " " & Mid(numericString, 4, 3) & " " &
Mid(numericString, 7)
         End If
      Else
         Throw New FormatException(String.Format("The {0} format specifier is invalid.", fmt))
      End If
      Return numericString
   End Function
End Class

Public Module TestTelephoneFormatter
   Public Sub Main
      Console.WriteLine(String.Format(New TelephoneFormatter, "{0}", 0))
      Console.WriteLine(String.Format(New TelephoneFormatter, "{0}", 911))
      Console.WriteLine(String.Format(New TelephoneFormatter, "{0}", 8490216))
      Console.WriteLine(String.Format(New TelephoneFormatter, "{0}", 4257884748))

      Console.WriteLine(String.Format(New TelephoneFormatter, "{0:N}", 0))
      Console.WriteLine(String.Format(New TelephoneFormatter, "{0:N}", 911))
      Console.WriteLine(String.Format(New TelephoneFormatter, "{0:N}", 8490216))
      Console.WriteLine(String.Format(New TelephoneFormatter, "{0:N}", 4257884748))

      Console.WriteLine(String.Format(New TelephoneFormatter, "{0:I}", 4257884748))
   End Sub
End Module
```

The custom numeric format provider can be used only with the String.Format(IFormatProvider, String, Object[])

method. The other overloads of numeric formatting methods (such as `ToString`) that have a parameter of type IFormatProvider all pass the IFormatProvider.GetFormat implementation a Type object that represents the NumberFormatInfo type. In return, they expect the method to return a NumberFormatInfo object. If it does not, the custom numeric format provider is ignored, and the NumberFormatInfo object for the current culture is used in its place. In the example, the `TelephoneFormatter.GetFormat` method handles the possibility that it may be inappropriately passed to a numeric formatting method by examining the method parameter and returning `null` if it represents a type other than ICustomFormatter.

If a custom numeric format provider supports a set of format specifiers, make sure you provide a default behavior if no format specifier is supplied in the format item used in the String.Format(IFormatProvider, String, Object[]) method call. In the example, "N" is the default format specifier. This allows for a number to be converted to a formatted telephone number by providing an explicit format specifier. The following example illustrates such a method call.

```
Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}", 4257884748));
```

```
Console.WriteLine(String.Format(New TelephoneFormatter, "{0:N}", 4257884748))
```

But it also allows the conversion to occur if no format specifier is present. The following example illustrates such a method call.

```
Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}", 4257884748));
```

```
Console.WriteLine(String.Format(New TelephoneFormatter, "{0}", 4257884748))
```

If no default format specifier is defined, your implementation of the ICustomFormatter.Format method should include code such as the following so that .NET can provide formatting that your code does not support.

```
if (arg is IFormattable)
   s = ((IFormattable)arg).ToString(format, formatProvider);
else if (arg != null)
   s = arg.ToString();
```

```
If TypeOf(arg) Is IFormattable Then
   s = DirectCast(arg, IFormattable).ToString(fmt, formatProvider)
ElseIf arg IsNot Nothing Then
   s = arg.ToString()
End If
```

In the case of this example, the method that implements ICustomFormatter.Format is intended to serve as a callback method for the String.Format(IFormatProvider, String, Object[]) method. Therefore, it examines the `formatProvider` parameter to determine whether it contains a reference to the current `TelephoneFormatter` object. However, the method can also be called directly from code. In that case, you can use the `formatProvider` parameter to provide a CultureInfo or NumberFormatInfo object that supplies culture-specific formatting information.

Compiling the Code

Compile the code at the command line using csc.exe or vb.exe. To compile the code in Visual Studio, put it in a console application project template.

See also

- Performing Formatting Operations

How to: Round-trip Date and Time Values

9/6/2018 • 8 minutes to read • Edit Online

In many applications, a date and time value is intended to unambiguously identify a single point in time. This topic shows how to save and restore a DateTime value, a DateTimeOffset value, and a date and time value with time zone information so that the restored value identifies the same time as the saved value.

To round-trip a DateTime value

1. Convert the DateTime value to its string representation by calling the DateTime.ToString(String) method with the "o" format specifier.

2. Save the string representation of the DateTime value to a file, or pass it across a process, application domain, or machine boundary.

3. Retrieve the string that represents the DateTime value.

4. Call the DateTime.Parse(String, IFormatProvider, DateTimeStyles) method, and pass DateTimeStyles.RoundtripKind as the value of the `styles` parameter.

The following example illustrates how to round-trip a DateTime value.

```
const string fileName = @".\DateFile.txt";

StreamWriter outFile = new StreamWriter(fileName);

// Save DateTime value.
DateTime dateToSave = DateTime.SpecifyKind(new DateTime(2008, 6, 12, 18, 45, 15),
                                           DateTimeKind.Local);
string dateString = dateToSave.ToString("o");
Console.WriteLine("Converted {0} ({1}) to {2}.",
                  dateToSave.ToString(),
                  dateToSave.Kind.ToString(),
                  dateString);
outFile.WriteLine(dateString);
Console.WriteLine("Wrote {0} to {1}.", dateString, fileName);
outFile.Close();

// Restore DateTime value.
DateTime restoredDate;

StreamReader inFile = new StreamReader(fileName);
dateString = inFile.ReadLine();
inFile.Close();
restoredDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Read {0} ({2}) from {1}.", restoredDate.ToString(),
                                              fileName,
                                              restoredDate.Kind.ToString());
// The example displays the following output:
//    Converted 6/12/2008 6:45:15 PM (Local) to 2008-06-12T18:45:15.0000000-05:00.
//    Wrote 2008-06-12T18:45:15.0000000-05:00 to .\DateFile.txt.
//    Read 6/12/2008 6:45:15 PM (Local) from .\DateFile.txt.
```

```
Const fileName As String = ".\DateFile.txt"

Dim outFile As New StreamWriter(fileName)

' Save DateTime value.
Dim dateToSave As Date = DateTime.SpecifyKind(#06/12/2008 6:45:15 PM#, _
                                              DateTimeKind.Local)
Dim dateString As String = dateToSave.ToString("o")
Console.WriteLine("Converted {0} ({1}) to {2}.", dateToSave.ToString(), _
                  dateToSave.Kind.ToString(), dateString)
outFile.WriteLine(dateString)
Console.WriteLine("Wrote {0} to {1}.", dateString, fileName)
outFile.Close()

' Restore DateTime value.
Dim restoredDate As Date

Dim inFile As New StreamReader(fileName)
dateString = inFile.ReadLine()
inFile.Close()
restoredDate = DateTime.Parse(dateString, Nothing, DateTimeStyles.RoundTripKind)
Console.WriteLine("Read {0} ({2}) from {1}.", restoredDate.ToString(), _
                  fileName, restoredDAte.Kind.ToString())
' The example displays the following output:
'    Converted 6/12/2008 6:45:15 PM (Local) to 2008-06-12T18:45:15.0000000-05:00.
'    Wrote 2008-06-12T18:45:15.0000000-05:00 to .\DateFile.txt.
'    Read 6/12/2008 6:45:15 PM (Local) from .\DateFile.txt.
```

When round-tripping a DateTime value, this technique successfully preserves the time for all local and universal times. For example, if a local DateTime value is saved on a system in the U.S. Pacific Standard Time zone and is restored on a system in the U.S. Central Standard Time zone, the restored date and time will be two hours later than the original time, which reflects the time difference between the two time zones. However, this technique is not necessarily accurate for unspecified times. All DateTime values whose Kind property is Unspecified are treated as if they are local times. If this is not the case, the DateTime will not successfully identify the correct point in time. The workaround for this limitation is to tightly couple a date and time value with its time zone for the save and restore operation.

To round-trip a DateTimeOffset value

1. Convert the DateTimeOffset value to its string representation by calling the DateTimeOffset.ToString(String) method with the "o" format specifier.

2. Save the string representation of the DateTimeOffset value to a file, or pass it across a process, application domain, or machine boundary.

3. Retrieve the string that represents the DateTimeOffset value.

4. Call the DateTimeOffset.Parse(String, IFormatProvider, DateTimeStyles) method, and pass DateTimeStyles.RoundtripKind as the value of the `styles` parameter.

The following example illustrates how to round-trip a DateTimeOffset value.

```csharp
const string fileName = @".\DateOff.txt";

StreamWriter outFile = new StreamWriter(fileName);

// Save DateTime value.
DateTimeOffset dateToSave = new DateTimeOffset(2008, 6, 12, 18, 45, 15,
                                               new TimeSpan(7, 0, 0));
string dateString = dateToSave.ToString("o");
Console.WriteLine("Converted {0} to {1}.", dateToSave.ToString(),
                  dateString);
outFile.WriteLine(dateString);
Console.WriteLine("Wrote {0} to {1}.", dateString, fileName);
outFile.Close();

// Restore DateTime value.
DateTimeOffset restoredDateOff;

StreamReader inFile = new StreamReader(fileName);
dateString = inFile.ReadLine();
inFile.Close();
restoredDateOff = DateTimeOffset.Parse(dateString, null,
                                       DateTimeStyles.RoundtripKind);
Console.WriteLine("Read {0} from {1}.", restoredDateOff.ToString(),
                  fileName);
// The example displays the following output:
//    Converted 6/12/2008 6:45:15 PM +07:00 to 2008-06-12T18:45:15.0000000+07:00.
//    Wrote 2008-06-12T18:45:15.0000000+07:00 to .\DateOff.txt.
//    Read 6/12/2008 6:45:15 PM +07:00 from .\DateOff.txt.
```

```vb
Const fileName As String = ".\DateOff.txt"

Dim outFile As New StreamWriter(fileName)

' Save DateTime value.
Dim dateToSave As New DateTimeOffset(2008, 6, 12, 18, 45, 15, _
                                     New TimeSpan(7, 0, 0))
Dim dateString As String = dateToSave.ToString("o")
Console.WriteLine("Converted {0} to {1}.", dateToSave.ToString(), dateString)
outFile.WriteLine(dateString)
Console.WriteLine("Wrote {0} to {1}.", dateString, fileName)
outFile.Close()

' Restore DateTime value.
Dim restoredDateOff As DateTimeOffset

Dim inFile As New StreamReader(fileName)
dateString = inFile.ReadLine()
inFile.Close()
restoredDateOff = DateTimeOffset.Parse(dateString, Nothing, DateTimeStyles.RoundTripKind)
Console.WriteLine("Read {0} from {1}.", restoredDateOff.ToString(), fileName)
' The example displays the following output:
'    Converted 6/12/2008 6:45:15 PM +07:00 to 2008-06-12T18:45:15.0000000+07:00.
'    Wrote 2008-06-12T18:45:15.0000000+07:00 to .\DateOff.txt.
'    Read 6/12/2008 6:45:15 PM +07:00 from .\DateOff.txt.
```

This technique always unambiguously identifies a DateTimeOffset value as a single point in time. The value can then be converted to Coordinated Universal Time (UTC) by calling the DateTimeOffset.ToUniversalTime method, or it can be converted to the time in a particular time zone by calling the DateTimeOffset.ToOffset or TimeZoneInfo.ConvertTime(DateTimeOffset, TimeZoneInfo) method. The major limitation of this technique is that date and time arithmetic, when performed on a DateTimeOffset value that represents the time in a particular time zone, may not produce accurate results for that time zone. This is because when a DateTimeOffset value is instantiated, it is disassociated from its time zone. Therefore, that time zone's adjustment rules can no longer be applied when you perform date and time calculations. You can work around this problem by defining a custom type that includes both a date and time value and its accompanying time zone.

To round-trip a date and time value with its time zone

1. Define a class or a structure with two fields. The first field is either a DateTime or a DateTimeOffset object, and the second is a TimeZoneInfo object. The following example is a simple version of such a type.

```
[Serializable] public class DateInTimeZone
{
   private TimeZoneInfo tz;
   private DateTimeOffset thisDate;

   public DateInTimeZone() {}

   public DateInTimeZone(DateTimeOffset date, TimeZoneInfo timeZone)
   {
      if (timeZone == null)
         throw new ArgumentNullException("The time zone cannot be null.");

      this.thisDate = date;
      this.tz = timeZone;
   }

   public DateTimeOffset DateAndTime
   {
      get {
         return this.thisDate;
      }
      set {
         if (value.Offset != this.tz.GetUtcOffset(value))
            this.thisDate = TimeZoneInfo.ConvertTime(value, tz);
         else
            this.thisDate = value;
      }
   }

   public TimeZoneInfo TimeZone
   {
      get {
         return this.tz;
      }
   }
}
```

```vb
<Serializable> Public Class DateInTimeZone
   Private tz As TimeZoneInfo
   Private thisDate As DateTimeOffset

   Public Sub New()
   End Sub

   Public Sub New(date1 As DateTimeOffset, timeZone As TimeZoneInfo)
      If timeZone Is Nothing Then
         Throw New ArgumentNullException("The time zone cannot be null.")
      End If
      Me.thisDate = date1
      Me.tz = timeZone
   End Sub

   Public Property DateAndTime As DateTimeOffset
      Get
         Return Me.thisDate
      End Get
      Set
         If Value.Offset <> Me.tz.GetUtcOffset(Value) Then
            Me.thisDate = TimeZoneInfo.ConvertTime(Value, tz)
         Else
            Me.thisDate = Value
         End If
      End Set
   End Property

   Public ReadOnly Property TimeZone As TimeZoneInfo
      Get
         Return tz
      End Get
   End Property
End Class
```

2. Mark the class with the SerializableAttribute attribute.

3. Serialize the object using the BinaryFormatter.Serialize method.

4. Restore the object using the Deserialize method.

5. Cast (in C#) or convert (in Visual Basic) the deserialized object to an object of the appropriate type.

The following example illustrates how to round-trip an object that stores both date and time and time zone information.

```
const string fileName = @".\DateWithTz.dat";

DateTime tempDate = new DateTime(2008, 9, 3, 19, 0, 0);
TimeZoneInfo tempTz = TimeZoneInfo.FindSystemTimeZoneById("Central Standard Time");
DateInTimeZone dateWithTz = new DateInTimeZone(new DateTimeOffset(tempDate,
                                tempTz.GetUtcOffset(tempDate)),
                                tempTz);

// Store DateInTimeZone value to a file
FileStream outFile = new FileStream(fileName, FileMode.Create);
try
{
   BinaryFormatter formatter = new BinaryFormatter();
   formatter.Serialize(outFile, dateWithTz);
   Console.WriteLine("Saving {0} {1} to {2}", dateWithTz.DateAndTime,
                     dateWithTz.TimeZone.IsDaylightSavingTime(dateWithTz.DateAndTime) ?
                     dateWithTz.TimeZone.DaylightName : dateWithTz.TimeZone.DaylightName,
                     fileName);
}
catch (SerializationException)
{
   Console.WriteLine("Unable to serialize time data to {0}.", fileName);
}
finally
{
   outFile.Close();
}

// Retrieve DateInTimeZone value
if (File.Exists(fileName))
{
   FileStream inFile = new FileStream(fileName, FileMode.Open);
   DateInTimeZone dateWithTz2 = new DateInTimeZone();
   try
   {
      BinaryFormatter formatter = new BinaryFormatter();
      dateWithTz2 = formatter.Deserialize(inFile) as DateInTimeZone;
      Console.WriteLine("Restored {0} {1} from {2}", dateWithTz2.DateAndTime,
                        dateWithTz2.TimeZone.IsDaylightSavingTime(dateWithTz2.DateAndTime) ?
                        dateWithTz2.TimeZone.DaylightName : dateWithTz2.TimeZone.DaylightName,
                        fileName);
   }
   catch (SerializationException)
   {
      Console.WriteLine("Unable to retrieve date and time information from {0}",
                        fileName);
   }
   finally
   {
      inFile.Close();
   }
}
// This example displays the following output to the console:
//    Saving 9/3/2008 7:00:00 PM -05:00 Central Daylight Time to .\DateWithTz.dat
//    Restored 9/3/2008 7:00:00 PM -05:00 Central Daylight Time from .\DateWithTz.dat
```

```vb
Const fileName As String = ".\DateWithTz.dat"

Dim tempDate As Date = #9/3/2008 7:00:00 PM#
Dim tempTz As TimeZoneInfo = TimeZoneInfo.FindSystemTimeZoneById("Central Standard Time")
Dim dateWithTz As New DateInTimeZone(New DateTimeOffset(tempDate, _
                                         tempTz.GetUtcOffset(tempDate)), _
                                     tempTz)

' Store DateInTimeZone value to a file
Dim outFile As New FileStream(fileName, FileMode.Create)
Try
   Dim formatter As New BinaryFormatter()
   formatter.Serialize(outFile, dateWithTz)
   Console.WriteLine("Saving {0} {1} to {2}", dateWithTz.DateAndTime, _
           IIf(dateWithTz.TimeZone.IsDaylightSavingTime(dateWithTz.DateAndTime), _
               dateWithTz.TimeZone.DaylightName, dateWithTz.TimeZone.DaylightName), _
           fileName)
Catch e As SerializationException
   Console.WriteLine("Unable to serialize time data to {0}.", fileName)
Finally
   outFile.Close()
End Try

' Retrieve DateInTimeZone value
If File.Exists(fileName) Then
   Dim inFile As New FileStream(fileName, FileMode.Open)
   Dim dateWithTz2 As New DateInTimeZone()
   Try
      Dim formatter As New BinaryFormatter()
      dateWithTz2 = DirectCast(formatter.Deserialize(inFile), DateInTimeZone)
      Console.WriteLine("Restored {0} {1} from {2}", dateWithTz2.DateAndTime, _
                        IIf(dateWithTz2.TimeZone.IsDaylightSavingTime(dateWithTz2.DateAndTime), _
                        dateWithTz2.TimeZone.DaylightName, dateWithTz2.TimeZone.DaylightName), _
                        fileName)
   Catch e As SerializationException
      Console.WriteLine("Unable to retrieve date and time information from {0}", _
                        fileName)
   Finally
      inFile.Close
   End Try
End If
' This example displays the following output to the console:
'    Saving 9/3/2008 7:00:00 PM -05:00 Central Daylight Time to .\DateWithTz.dat
'    Restored 9/3/2008 7:00:00 PM -05:00 Central Daylight Time from .\DateWithTz.dat
```

This technique should always unambiguously reflect the correct point of time both before and after it is saved and restored, provided that the implementation of the combined date and time and time zone object does not allow the date value to become out of sync with the time zone value.

Compiling the Code

These examples require:

- That the following namespaces be imported with C# `using` statements or Visual Basic `Imports` statements:
  - System (C# only).
  - System.Globalization.
  - System.IO.
  - System.Runtime.Serialization.
  - System.Runtime.Serialization.Formatters.Binary.

- A reference to System.Core.dll.
- Each code example, other than the `DateInTimeZone` class, should be included in a class or Visual Basic module, wrapped in methods, and called from the `Main` method.

See also

- Performing Formatting Operations
- Choosing Between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo
- Standard Date and Time Format Strings

How to: Convert Numeric User Input in Web Controls to Numbers

9/6/2018 • 6 minutes to read • Edit Online

Because a Web page can be displayed anywhere in the world, users can input numeric data into a TextBox control in an almost unlimited number of formats. As a result, it is very important to determine the locale and culture of the Web page's user. When you parse user input, you can then apply the formatting conventions defined by the user's locale and culture.

To convert numeric input from a Web TextBox control to a number

1. Determine whether the string array returned by the HttpRequest.UserLanguages property is populated. If it is not, continue to step 6.
2. If the string array returned by the UserLanguages property is populated, retrieve its first element. The first element indicates the user's default or preferred language and region.
3. Instantiate a CultureInfo object that represents the user's preferred culture by calling the CultureInfo.CultureInfo(String, Boolean) constructor.
4. Call either the `TryParse` or the `Parse` method of the numeric type that you want to convert the user's input to. Use an overload of the `TryParse` or the `Parse` method with a `provider` parameter, and pass it either of the following:
   - The CultureInfo object created in step 3.
   - The NumberFormatInfo object that is returned by the NumberFormat property of the CultureInfo object created in step 3.
5. If the conversion fails, repeat steps 2 through 4 for each remaining element in the string array returned by the UserLanguages property.
6. If the conversion still fails or if the string array returned by the UserLanguages property is empty, parse the string by using the invariant culture, which is returned by the CultureInfo.InvariantCulture property.

Example

The following example is the complete code-behind page for a Web form that asks the user to enter a numeric value in a TextBox control and converts it to a number. That number is then doubled and displayed by using the same formatting rules as the original input.

```
using System;
using System.Globalization;

partial class NumericUserInput : System.Web.UI.Page
{
   protected void OKButton_Click(object sender, EventArgs e)
   {
      string locale;
      CultureInfo culture = null;
      double number = 0;
      bool result = false;

      // Exit if input is absent.
      if (String.IsNullOrEmpty(this.NumericString.Text)) return;
```

```
      // Hide form elements.
      this.NumericInput.Visible = false;

      // Get user culture/region
      if (!(Request.UserLanguages.Length == 0 || String.IsNullOrEmpty(Request.UserLanguages[0])))
      {
         try
         {
            locale = Request.UserLanguages[0];
            culture = new CultureInfo(locale, false);

            // Parse input using user culture.
            result = Double.TryParse(this.NumericString.Text, NumberStyles.Any,
                                     culture.NumberFormat, out number);
         }
         catch { }
         // If parse fails, parse input using any additional languages.
         if (!result)
         {
            if (Request.UserLanguages.Length > 1)
            {
               for (int ctr = 1; ctr <= Request.UserLanguages.Length - 1; ctr++)
               {
                  try
                  {
                     locale = Request.UserLanguages[ctr];
                     // Remove quality specifier, if present.
                     locale = locale.Substring(1, locale.IndexOf(';') - 1);
                     culture = new CultureInfo(Request.UserLanguages[ctr], false);
                     result = Double.TryParse(this.NumericString.Text, NumberStyles.Any, culture.NumberFormat,
out number);
                     if (result) break;
                  }
                  catch { }
               }
            }
         }
      }
      // If parse operation fails, use invariant culture.
      if (!result)
         result = Double.TryParse(this.NumericString.Text, NumberStyles.Any, CultureInfo.InvariantCulture, out
number);

      // Double result.
      number *= 2;

      // Display result to user.
      if (result)
      {
         Response.Write("<P />");
         Response.Write(Server.HtmlEncode(this.NumericString.Text) + " * 2 = " + number.ToString("N", culture)
+ "<BR />");
      }
      else
      {
         // Unhide form.
         this.NumericInput.Visible = true;

         Response.Write("<P />");
         Response.Write("Unable to recognize " + Server.HtmlEncode(this.NumericString.Text));
      }
   }
}
```

```
Imports System.Globalization

Partial Class NumericUserInput
```

```
   Inherits System.Web.UI.Page

   Protected Sub OKButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles OKButton.Click
      Dim locale As String
      Dim culture As CultureInfo = Nothing
      Dim number As Double
      Dim result As Boolean

      ' Exit if input is absent.
      If String.IsNullOrEmpty(Me.NumericString.Text) Then Exit Sub

      ' Hide form elements.
      Me.NumericInput.Visible = False

      ' Get user culture/region
      If Not (Request.UserLanguages.Length = 0 OrElse String.IsNullOrEmpty(Request.UserLanguages(0))) Then
         Try
            locale = Request.UserLanguages(0)
            culture = New CultureInfo(locale, False)

            ' Parse input using user culture.
            result = Double.TryParse(Me.NumericString.Text, NumberStyles.Any, culture.NumberFormat, number)
         Catch
         End Try
         ' If parse fails, parse input using any additional languages.
         If Not result Then
            If Request.UserLanguages.Length > 1 Then
               For ctr As Integer = 1 To Request.UserLanguages.Length - 1
                  Try
                     locale = Request.UserLanguages(ctr)
                     ' Remove quality specifier, if present.
                     locale = Left(locale, InStr(locale, ";") - 1)
                     culture = New CultureInfo(Request.UserLanguages(ctr), False)
                     result = Double.TryParse(Me.NumericString.Text, NumberStyles.Any, culture.NumberFormat,
number)
                     If result Then Exit For
                  Catch
                  End Try
               Next
            End If
         End If
      End If
      ' If parse operation fails, use invariant culture.
      If Not result Then
         result = Double.TryParse(Me.NumericString.Text, NumberStyles.Any, CultureInfo.InvariantCulture,
number)
      End If
      ' Double result
      number *= 2

      ' Display result to user.
      If result Then
         Response.Write("<P />")
         Response.Write(Server.HtmlEncode(Me.NumericString.Text) + " * 2 = " + number.ToString("N", culture) +
"<BR />")
      Else
         ' Unhide form.
         Me.NumericInput.Visible = True

         Response.Write("<P />")
         Response.Write("Unable to recognize " + Server.HtmlEncode(Me.NumericString.Text))
      End If
   End Sub
End Class
```

The HttpRequest.UserLanguages property is populated from the culture names that are contained in `Accept-Language` headers included in an HTTP request. However, not all browsers include `Accept-Language` headers in their requests, and users can also suppress the headers completely. This makes it important to have a fallback culture when parsing user input. Typically, the fallback culture is the invariant culture returned by CultureInfo.InvariantCulture. Users can also provide Internet Explorer with culture names that they input in a text box, which creates the possibility that the culture names may not be valid. This makes it important to use exception handling when instantiating a CultureInfo object.

When retrieved from an HTTP request submitted by Internet Explorer, the HttpRequest.UserLanguages array is populated in order of user preference. The first element in the array contains the name of the user's primary culture/region. If the array contains any additional items, Internet Explorer arbitrarily assigns them a quality specifier, which is delimited from the culture name by a semicolon. For example, an entry for the fr-FR culture might take the form `fr-FR;q=0.7`.

The example calls the CultureInfo constructor with its `useUserOverride` parameter set to `false` to create a new CultureInfo object. This ensures that, if the culture name is the default culture name on the server, the new CultureInfo object created by the class constructor contains a culture's default settings and does not reflect any settings overridden by using the server's **Regional and Language Options** application. The values from any overridden settings on the server are unlikely to exist on the user's system or to be reflected in the user's input.

Your code can call either the `Parse` or the `TryParse` method of the numeric type that the user's input will be converted to. Repeated calls to a parse method may be required for a single parsing operation. As a result, the `TryParse` method is better, because it returns `false` if a parse operation fails. In contrast, handling the repeated exceptions that may be thrown by the `Parse` method can be a very expensive proposition in a Web application.

Compiling the Code

To compile the code, copy it into an ASP.NET code-behind page so that it replaces all the existing code. The ASP.NET Web page should contain the following controls:

- A Label control, which is not referenced in code. Set its Text property to "Enter a Number:".
- A TextBox control named `NumericString`.
- A Button control named `OKButton`. Set its Text property to "OK".

Change the name of the class from `NumericUserInput` to the name of the class that is defined by the `Inherits` attribute of the ASP.NET page's `Page` directive. Change the name of the `NumericInput` object reference to the name defined by the `id` attribute of the ASP.NET page's `form` tag.

.NET Framework Security

To prevent a user from injecting script into the HTML stream, user input should never be directly echoed back in the server response. Instead, it should be encoded by using the HttpServerUtility.HtmlEncode method.

See also

- Performing Formatting Operations
- Parsing Numeric Strings

How to: Display Localized Date and Time Information to Web Users

9/6/2018 • 11 minutes to read • Edit Online

Because a Web page can be displayed anywhere in the world, operations that parse and format date and time values should not rely on a default format (which most often is the format of the Web server's local culture) when interacting with the user. Instead, Web forms that handle date and time strings input by the user should parse the strings using the user's preferred culture. Similarly, date and time data should be displayed to the user in a format that conforms to the user's culture. This topic shows how to do this.

To parse date and time strings input by the user

1. Determine whether the string array returned by the HttpRequest.UserLanguages property is populated. If it is not, continue to step 6.

2. If the string array returned by the UserLanguages property is populated, retrieve its first element. The first element indicates the user's default or preferred language and region.

3. Instantiate a CultureInfo object that represents the user's preferred culture by calling the CultureInfo.CultureInfo(String, Boolean) constructor.

4. Call either the `TryParse` or the `Parse` method of the DateTime or DateTimeOffset type to try the conversion. Use an overload of the `TryParse` or the `Parse` method with a `provider` parameter, and pass it either of the following:

   - The CultureInfo object created in step 3.

   - The DateTimeFormatInfo object that is returned by the DateTimeFormat property of the CultureInfo object created in step 3.

5. If the conversion fails, repeat steps 2 through 4 for each remaining element in the string array returned by the UserLanguages property.

6. If the conversion still fails or if the string array returned by the UserLanguages property is empty, parse the string by using the invariant culture, which is returned by the CultureInfo.InvariantCulture property.

To parse the local date and time of the user's request

1. Add a HiddenField control to a Web form.

2. Create a JavaScript function that handles the `onClick` event of a `Submit` button by writing the current date and time and the local time zone's offset from Coordinated Universal Time (UTC) to the Value property. Use a delimiter (such as a semicolon) to separate the two components of the string.

3. Use the Web form's PreRender event to inject the function into the HTML output stream by passing the text of the script to the ClientScriptManager.RegisterClientScriptBlock(Type, String, String, Boolean) method.

4. Connect the event handler to the `Submit` button's `onClick` event by providing the name of the JavaScript function to the `OnClientClick` attribute of the `Submit` button.

5. Create a handler for the `Submit` button's Click event.

6. In the event handler, determine whether the string array returned by the HttpRequest.UserLanguages property is populated. If it is not, continue to step 14.

7. If the string array returned by the UserLanguages property is populated, retrieve its first element. The first

element indicates the user's default or preferred language and region.

8. Instantiate a CultureInfo object that represents the user's preferred culture by calling the CultureInfo.CultureInfo(String, Boolean) constructor.

9. Pass the string assigned to the Value property to the Split method to store the string representation of the user's local date and time and the string representation of the user's local time zone offset in separate array elements.

10. Call either the DateTime.Parse or DateTime.TryParse(String, IFormatProvider, DateTimeStyles, DateTime) method to convert the date and time of the user's request to a DateTime value. Use an overload of the method with a `provider` parameter, and pass it either of the following:

    - The CultureInfo object created in step 8.

    - The DateTimeFormatInfo object that is returned by the DateTimeFormat property of the CultureInfo object created in step 8.

11. If the parse operation in step 10 fails, go to step 13. Otherwise, call the UInt32.Parse(String) method to convert the string representation of the user's time zone offset to an integer.

12. Instantiate a DateTimeOffset that represents the user's local time by calling the DateTimeOffset.DateTimeOffset(DateTime, TimeSpan) constructor.

13. If the conversion in step 10 fails, repeat steps 7 through 12 for each remaining element in the string array returned by the UserLanguages property.

14. If the conversion still fails or if the string array returned by the UserLanguages property is empty, parse the string by using the invariant culture, which is returned by the CultureInfo.InvariantCulture property. Then repeat steps 7 through 12.

The result is a DateTimeOffset object that represents the local time of the user of your Web page. You can then determine the equivalent UTC by calling the ToUniversalTime method. You can also determine the equivalent date and time on your Web server by calling the TimeZoneInfo.ConvertTime(DateTimeOffset, TimeZoneInfo) method and passing a value of TimeZoneInfo.Local as the time zone to convert the time to.

Example

The following example contains both the HTML source and the code for an ASP.NET Web form that asks the user to input a date and time value. A client-side script also writes information on the local date and time of the user's request and the offset of the user's time zone from UTC to a hidden field. This information is then parsed by the server, which returns a Web page that displays the user's input. It also displays the date and time of the user's request using the user's local time, the time on the server, and UTC.

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Globalization" %>
<%@ Assembly Name="System.Core" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<script runat="server">

    protected void OKButton_Click(object sender, EventArgs e)
    {
        string locale = "";
        DateTimeStyles styles = DateTimeStyles.AllowInnerWhite | DateTimeStyles.AllowLeadingWhite |
                                       DateTimeStyles.AllowTrailingWhite;
        DateTime inputDate;
```

```
DateTime localDate = DateTime.Now;
DateTimeOffset localDateOffset = DateTimeOffset.Now;
int integerOffset;
bool result = false;

// Exit if input is absent.
if (string.IsNullOrEmpty(this.DateString.Text)) return;

// Hide form elements.
this.DateForm.Visible = false;

// Create array of CultureInfo objects
CultureInfo[] cultures = new CultureInfo[Request.UserLanguages.Length + 1];
for (int ctr = Request.UserLanguages.GetLowerBound(0); ctr <= Request.UserLanguages.GetUpperBound(0);
     ctr++)
{
    locale = Request.UserLanguages[ctr];
    if (! string.IsNullOrEmpty(locale))
    {

        // Remove quality specifier, if present.
        if (locale.Contains(";"))
           locale = locale.Substring(locale.IndexOf(';') -1);
        try
        {
            cultures[ctr] = new CultureInfo(Request.UserLanguages[ctr], false);
        }
        catch (Exception) { }
    }
    else
    {
        cultures[ctr] = CultureInfo.CurrentCulture;
    }
}
cultures[Request.UserLanguages.Length] = CultureInfo.InvariantCulture;
// Parse input using each culture.
foreach (CultureInfo culture in cultures)
{
    result = DateTime.TryParse(this.DateString.Text, culture.DateTimeFormat, styles, out inputDate);
    if (result) break;
}
// Display result to user.
if (result)
{
    Response.Write("<P />");
    Response.Write("The date you input was " + Server.HtmlEncode(this.DateString.Text) + "<BR />");
}
else
{
    // Unhide form.
    this.DateForm.Visible = true;
    Response.Write("<P />");
    Response.Write("Unable to recognize " + Server.HtmlEncode(this.DateString.Text) + ".<BR />");
}

// Get date and time information from hidden field.
string[] dates= Request.Form["DateInfo"].Split(';');

// Parse local date using each culture.
foreach (CultureInfo culture in cultures)
{
    result = DateTime.TryParse(dates[0], culture.DateTimeFormat, styles, out localDate);
    if (result) break;
}
// Parse offset
result = int.TryParse(dates[1], out integerOffset);
// Instantiate DateTimeOffset object representing user's local time
if (result)
{
```

```
            try
            {
                localDateOffset = new DateTimeOffset(localDate, new TimeSpan(0, -integerOffset, 0));
            }
            catch (Exception)
            {
                result = false;
            }
        }
        // Display result to user.
        if (result)
        {
            Response.Write("<P />");
            Response.Write("Your local date and time is " + localDateOffset.ToString() + ".<BR />");
            Response.Write("The date and time on the server is " +
                           TimeZoneInfo.ConvertTime(localDateOffset,
                                                    TimeZoneInfo.Local).ToString() + ".<BR />");
            Response.Write("Coordinated Universal Time is " + localDateOffset.ToUniversalTime().ToString() +
".<BR />");
        }
        else
        {
            Response.Write("<P />");
            Response.Write("Unable to recognize " + Server.HtmlEncode(dates[0]) + ".<BR />");
        }
    }

    protected void Page_PreRender(object sender, System.EventArgs e)
    {
        string script = "function AddDateInformation() { \n" +
                  "var today = new Date();\n" +
                  "document.DateForm.DateInfo.value = today.toLocaleString() + \";\" +
today.getTimezoneOffset();\n" +
                  " }";
        // Register client script
        ClientScriptManager scriptMgr = Page.ClientScript;
        scriptMgr.RegisterClientScriptBlock(this.GetType(), "SubmitOnClick", script, true);
    }

</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Parsing a Date and Time Value</title>
</head>
<body>
    <form id="DateForm" runat="server">
    <div>
    <center>
       <asp:Label ID="Label1" runat="server" Text="Enter a Date and Time:" Width="248px"></asp:Label>
       <asp:TextBox ID="DateString" runat="server" Width="176px"></asp:TextBox><br />
    </center>
    <br />
    <center>
    <asp:Button ID="OKButton" runat="server" Text="Button"
            OnClientClick="AddDateInformation()" onclick="OKButton_Click" />
    <asp:HiddenField ID="DateInfo"  Value=""  runat="server" />
    </center>
    <br />
    </div>
    </form>
</body>
</html>
```

```
<%@ Page Language="VB" %>
```

```
<%@ Import  Namespace="System.Globalization" %>
<%@ Assembly Name="System.Core" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<script runat="server">

    Protected Sub OKButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles OKButton.Click
        Dim locale As String = ""
        Dim styles As DateTimeStyles = DateTimeStyles.AllowInnerWhite Or DateTimeStyles.AllowLeadingWhite Or _
                                       DateTimeStyles.AllowTrailingWhite
        Dim inputDate, localDate As Date
        Dim localDateOffset As DateTimeOffset
        Dim integerOffset As Integer
        Dim result As Boolean

        ' Exit if input is absent.
        If String.IsNullOrEmpty(Me.DateString.Text) Then Exit Sub

        ' Hide form elements.
        Me.DateForm.Visible = False

        ' Create array of CultureInfo objects
        Dim cultures(Request.UserLanguages.Length) As CultureInfo
        For ctr As Integer = Request.UserLanguages.GetLowerBound(0) To Request.UserLanguages.GetUpperBound(0)
            locale = Request.UserLanguages(ctr)
            If Not String.IsNullOrEmpty(locale) Then
                ' Remove quality specifier, if present.
                If locale.Contains(";") Then _
                   locale = Left(locale, InStr(locale, ";") - 1)
                Try
                   cultures(ctr) = New CultureInfo(Request.UserLanguages(ctr), False)
                Catch
                End Try
            Else
                cultures(ctr) = CultureInfo.CurrentCulture
            End If
        Next
        cultures(Request.UserLanguages.Length) = CultureInfo.InvariantCulture
        ' Parse input using each culture.
        For Each culture As CultureInfo In cultures
            result = Date.TryParse(Me.DateString.Text, culture.DateTimeFormat, styles, inputDate)
            If result Then Exit For
        Next
        ' Display result to user.
        If result Then
            Response.Write("<P />")
            Response.Write("The date you input was " + Server.HtmlEncode(CStr(Me.DateString.Text)) + "<BR />")
        Else
            ' Unhide form.
            Me.DateForm.Visible = True
            Response.Write("<P />")
            Response.Write("Unable to recognize " + Server.HtmlEncode(Me.DateString.Text) + ".<BR />")
        End If

        ' Get date and time information from hidden field.
        Dim dates() As String = Request.Form.Item("DateInfo").Split(";")

        ' Parse local date using each culture.
        For Each culture As CultureInfo In cultures
            result = Date.TryParse(dates(0), culture.DateTimeFormat, styles, localDate)
            If result Then Exit For
        Next
        ' Parse offset
        result = Integer.TryParse(dates(1), integerOffset)
        ' Instantiate DateTimeOffset object representing user's local time
        If result Then
            Try
```

```
                localDateOffset = New DateTimeOffset(localDate, New TimeSpan(0, -integerOffset, 0))
            Catch ex As Exception
                result = False
            End Try
        End If
        ' Display result to user.
        If result Then
            Response.Write("<P />")
            Response.Write("Your local date and time is " + localDateOffset.ToString() + ".<BR />")
            Response.Write("The date and time on the server is " & _
                           TimeZoneInfo.ConvertTime(localDateOffset, _
                                                    TimeZoneInfo.Local).ToString() & ".<BR />")
            Response.Write("Coordinated Universal Time is " + localDateOffset.ToUniversalTime.ToString() + ".
<BR />")
        Else
            Response.Write("<P />")
            Response.Write("Unable to recognize " + Server.HtmlEncode(dates(0)) & ".<BR />")
        End If
    End Sub

    Protected Sub Page_PreRender(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.PreRender
        Dim script As String = "function AddDateInformation() { " & vbCrLf & _
                  "var today = new Date();" & vbCrLf & _
                  "document.DateForm.DateInfo.value = today.toLocaleString() + " & Chr(34) & Chr(59) & Chr(34)
& " + today.getTimezoneOffset();" & vbCrLf & _
                  " }"
        ' Register client script
        Dim scriptMgr As ClientScriptManager = Page.ClientScript
        scriptMgr.RegisterClientScriptBlock(Me.GetType(), "SubmitOnClick", script, True)
    End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Parsing a Date and Time Value</title>
</head>
<body>
    <form id="DateForm" runat="server">
    <div>
    <center>
       <asp:Label ID="Label1" runat="server" Text="Enter a Date and Time:" Width="248px"></asp:Label>
       <asp:TextBox ID="DateString" runat="server" Width="176px"></asp:TextBox><br />
    </center>
    <br />
    <center>
    <asp:Button ID="OKButton" runat="server" Text="Button"
            OnClientClick="AddDateInformation()" onclick="OKButton_Click" />
    <asp:HiddenField ID="DateInfo"  Value=""  runat="server" />
    </center>
    <br />
    </div>
    </form>
</body>
</html>
```

The client-side script calls the JavaScript `toLocaleString` method. This produces a string that follows the formatting conventions of the user's locale, which is more likely to be successfully parsed on the server.

The HttpRequest.UserLanguages property is populated from the culture names that are contained in `Accept-Language` headers included in an HTTP request. However, not all browsers include `Accept-Language` headers in their requests, and users can also suppress the headers completely. This makes it important to have a fallback culture when parsing user input. Typically the fallback culture is the invariant culture returned by CultureInfo.InvariantCulture. Users can also provide Internet Explorer with culture names that they input in a text box, which creates the possibility that the culture names may not be valid. This makes it important to use exception handling when instantiating a CultureInfo object.

When retrieved from an HTTP request submitted by Internet Explorer, the HttpRequest.UserLanguages array is populated in order of user preference. The first element in the array contains the name of the user's primary culture/region. If the array contains any additional items, Internet Explorer arbitrarily assigns them a quality specifier, which is delimited from the culture name by a semicolon. For example, an entry for the fr-FR culture might take the form `fr-FR;q=0.7`.

The example calls the CultureInfo constructor with its `useUserOverride` parameter set to `false` to create a new CultureInfo object. This ensures that, if the culture name is the default culture name on the server, the new CultureInfo object created by the class constructor contains a culture's default settings and does not reflect any settings overridden by using the server's **Regional and Language Options** application. The values from any overridden settings on the server are unlikely to exist on the user's system or to be reflected in the user's input.

Because this example parses two string representations of a date and time (one input by the user, the other stored to the hidden field), it defines the possible CultureInfo objects that may be required in advance. It creates an array of CultureInfo objects that is one greater than the number of elements returned by the HttpRequest.UserLanguages property. It then instantiates a CultureInfo object for each language/region string, and also instantiates a CultureInfo object that represents CultureInfo.InvariantCulture.

Your code can call either the Parse or the TryParse method to convert the user's string representation of a date and time to a DateTime value. Repeated calls to a parse method may be required for a single parsing operation. As a result, the TryParse method is better because it returns `false` if a parse operation fails. In contrast, handling the repeated exceptions that may be thrown by the Parse method can be a very expensive proposition in a Web application.

Compiling the Code

To compile the code, create an ASP.NET Web page without a code-behind. Then copy the example into the Web page so that it replaces all the existing code. The ASP.NET Web page should contain the following controls:

- A Label control, which is not referenced in code. Set its Text property to "Enter a Number:".
- A TextBox control named `DateString`.
- A Button control named `OKButton`. Set its Text property to "OK".
- A HiddenField control named `DateInfo`.

.NET Framework Security

To prevent a user from injecting script into the HTML stream, user input should never be directly echoed back in the server response. Instead, it should be encoded by using the HttpServerUtility.HtmlEncode method.

See also

- Performing Formatting Operations
- Standard Date and Time Format Strings
- Custom Date and Time Format Strings
- Parsing Date and Time Strings

How to: Display Milliseconds in Date and Time Values

9/6/2018 • 5 minutes to read • Edit Online

The default date and time formatting methods, such as DateTime.ToString(), include the hours, minutes, and seconds of a time value but exclude its milliseconds component. This topic shows how to include a date and time's millisecond component in formatted date and time strings.

To display the millisecond component of a DateTime value

1. If you are working with the string representation of a date, convert it to a DateTime or a DateTimeOffset value by using the static DateTime.Parse(String) or DateTimeOffset.Parse(String) method.

2. To extract the string representation of a time's millisecond component, call the date and time value's DateTime.ToString(String) or ToString method, and pass the `fff` or `FFF` custom format pattern either alone or with other custom format specifiers as the `format` parameter.

Example

The example displays the millisecond component of a DateTime and a DateTimeOffset value to the console, both alone and included in a longer date and time string.

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class MillisecondDisplay
{
   public static void Main()
   {
      string dateString = "7/16/2008 8:32:45.126 AM";

      try
      {
         DateTime dateValue = DateTime.Parse(dateString);
         DateTimeOffset dateOffsetValue = DateTimeOffset.Parse(dateString);

         // Display Millisecond component alone.
         Console.WriteLine("Millisecond component only: {0}",
                           dateValue.ToString("fff"));
         Console.WriteLine("Millisecond component only: {0}",
                           dateOffsetValue.ToString("fff"));

         // Display Millisecond component with full date and time.
         Console.WriteLine("Date and Time with Milliseconds: {0}",
                           dateValue.ToString("MM/dd/yyyy hh:mm:ss.fff tt"));
         Console.WriteLine("Date and Time with Milliseconds: {0}",
                           dateOffsetValue.ToString("MM/dd/yyyy hh:mm:ss.fff tt"));

         // Append millisecond pattern to current culture's full date time pattern
         string fullPattern = DateTimeFormatInfo.CurrentInfo.FullDateTimePattern;
         fullPattern = Regex.Replace(fullPattern, "(:ss|:s)", "$1.fff");

         // Display Millisecond component with modified full date and time pattern.
         Console.WriteLine("Modified full date time pattern: {0}",
                           dateValue.ToString(fullPattern));
         Console.WriteLine("Modified full date time pattern: {0}",
                           dateOffsetValue.ToString(fullPattern));
      }
      catch (FormatException)
      {
         Console.WriteLine("Unable to convert {0} to a date.", dateString);
      }
   }
}
// The example displays the following output if the current culture is en-US:
//    Millisecond component only: 126
//    Millisecond component only: 126
//    Date and Time with Milliseconds: 07/16/2008 08:32:45.126 AM
//    Date and Time with Milliseconds: 07/16/2008 08:32:45.126 AM
//    Modified full date time pattern: Wednesday, July 16, 2008 8:32:45.126 AM
//    Modified full date time pattern: Wednesday, July 16, 2008 8:32:45.126 AM
```

```
Imports System.Globalization
Imports System.Text.REgularExpressions

Module MillisecondDisplay
   Public Sub Main()

      Dim dateString As String = "7/16/2008 8:32:45.126 AM"

      Try
         Dim dateValue As Date = Date.Parse(dateString)
         Dim dateOffsetValue As DateTimeOffset = DateTimeOffset.Parse(dateString)

         ' Display Millisecond component alone.
         Console.WriteLine("Millisecond component only: {0}", _
                           dateValue.ToString("fff"))
         Console.WriteLine("Millisecond component only: {0}", _
                           dateOffsetValue.ToString("fff"))

         ' Display Millisecond component with full date and time.
         Console.WriteLine("Date and Time with Milliseconds: {0}", _
                           dateValue.ToString("MM/dd/yyyy hh:mm:ss.fff tt"))
         Console.WriteLine("Date and Time with Milliseconds: {0}", _
                           dateOffsetValue.ToString("MM/dd/yyyy hh:mm:ss.fff tt"))

         ' Append millisecond pattern to current culture's full date time pattern
         Dim fullPattern As String = DateTimeFormatInfo.CurrentInfo.FullDateTimePattern
         fullPattern = Regex.Replace(fullPattern, "(:ss|:s)", "$1.fff")

         ' Display Millisecond component with modified full date and time pattern.
         Console.WriteLine("Modified full date time pattern: {0}", _
                           dateValue.ToString(fullPattern))
         Console.WriteLine("Modified full date time pattern: {0}", _
                           dateOffsetValue.ToString(fullPattern))
      Catch e As FormatException
         Console.WriteLine("Unable to convert {0} to a date.", dateString)
      End Try
   End Sub
End Module
' The example displays the following output if the current culture is en-US:
'    Millisecond component only: 126
'    Millisecond component only: 126
'    Date and Time with Milliseconds: 07/16/2008 08:32:45.126 AM
'    Date and Time with Milliseconds: 07/16/2008 08:32:45.126 AM
'    Modified full date time pattern: Wednesday, July 16, 2008 8:32:45.126 AM
'    Modified full date time pattern: Wednesday, July 16, 2008 8:32:45.126 AM
```

The `fff` format pattern includes any trailing zeros in the millisecond value. The `FFF` format pattern suppresses them. The difference is illustrated in the following example.

```
DateTime dateValue = new DateTime(2008, 7, 16, 8, 32, 45, 180);
Console.WriteLine(dateValue.ToString("fff"));
Console.WriteLine(dateValue.ToString("FFF"));
// The example displays the following output to the console:
//    180
//    18
```

```
Dim dateValue As New Date(2008, 7, 16, 8, 32, 45, 180)
Console.WriteLIne(dateValue.ToString("fff"))
Console.WriteLine(dateValue.ToString("FFF"))
' The example displays the following output to the console:
'    180
'    18
```

A problem with defining a complete custom format specifier that includes the millisecond component of a date and time is that it defines a hard-coded format that may not correspond to the arrangement of time elements in the application's current culture. A better alternative is to retrieve one of the date and time display patterns defined by the current culture's DateTimeFormatInfo object and modify it to include milliseconds. The example also illustrates this approach. It retrieves the current culture's full date and time pattern from the DateTimeFormatInfo.FullDateTimePattern property, and then inserts the custom pattern `.ffff` after its seconds pattern. Note that the example uses a regular expression to perform this operation in a single method call.

You can also use a custom format specifier to display a fractional part of seconds other than milliseconds. For example, the `f` or `F` custom format specifier displays tenths of a second, the `ff` or `FF` custom format specifier displays hundredths of a second, and the `ffff` or `FFFF` custom format specifier displays ten thousandths of a second. Fractional parts of a millisecond are truncated instead of rounded in the returned string. These format specifiers are used in the following example.

```
DateTime dateValue = new DateTime(2008, 7, 16, 8, 32, 45, 180);
Console.WriteLine("{0} seconds", dateValue.ToString("s.f"));
Console.WriteLine("{0} seconds", dateValue.ToString("s.ff"));
Console.WriteLine("{0} seconds", dateValue.ToString("s.ffff"));
// The example displays the following output to the console:
//    45.1 seconds
//    45.18 seconds
//    45.1800 seconds
```

```
Dim dateValue As New DateTime(2008, 7, 16, 8, 32, 45, 180)
Console.WriteLine("{0} seconds", dateValue.ToString("s.f"))
Console.WriteLine("{0} seconds", dateValue.ToString("s.ff"))
Console.WriteLine("{0} seconds", dateValue.ToString("s.ffff"))
' The example displays the following output to the console:
'    45.1 seconds
'    45.18 seconds
'    45.1800 seconds
```

NOTE

It is possible to display very small fractional units of a second, such as ten thousandths of a second or hundred-thousandths of a second. However, these values may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On Windows NT 3.5 and later, and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

Compiling the Code

Compile the code at the command line using csc.exe or vb.exe. To compile the code in Visual Studio, put it in a console application project template.

See also

- DateTimeFormatInfo
- Custom Date and Time Format Strings

How to: Display Dates in Non-Gregorian Calendars

9/6/2018 • 8 minutes to read • Edit Online

The DateTime and DateTimeOffset types use the Gregorian calendar as their default calendar. This means that calling a date and time value's `ToString` method displays the string representation of that date and time in the Gregorian calendar, even if that date and time was created using another calendar. This is illustrated in the following example, which uses two different ways to create a date and time value with the Persian calendar, but still displays those date and time values in the Gregorian calendar when it calls the ToString method. This example reflects two commonly used but incorrect techniques for displaying the date in a particular calendar.

```
PersianCalendar persianCal = new PersianCalendar();

DateTime persianDate = persianCal.ToDateTime(1387, 3, 18, 12, 0, 0, 0);
Console.WriteLine(persianDate.ToString());

persianDate = new DateTime(1387, 3, 18, persianCal);
Console.WriteLine(persianDate.ToString());
// The example displays the following output to the console:
//       6/7/2008 12:00:00 PM
//       6/7/2008 12:00:00 AM
```

```
Dim persianCal As New PersianCalendar()

Dim persianDate As Date = persianCal.ToDateTime(1387, 3, 18, _
                                                12, 0, 0, 0)
Console.WriteLine(persianDate.ToString())

persianDate = New DateTime(1387, 3, 18, persianCal)
Console.WriteLine(persianDate.ToString())
' The example displays the following output to the console:
'       6/7/2008 12:00:00 PM
'       6/7/2008 12:00:00 AM
```

Two different techniques can be used to display the date in a particular calendar. The first requires that the calendar be the default calendar for a particular culture. The second can be used with any calendar.

To display the date for a culture's default calendar

1. Instantiate a calendar object derived from the Calendar class that represents the calendar to be used.

2. Instantiate a CultureInfo object representing the culture whose formatting will be used to display the date.

3. Call the Array.Exists method to determine whether the calendar object is a member of the array returned by the CultureInfo.OptionalCalendars property. This indicates that the calendar can serve as the default calendar for the CultureInfo object. If it is not a member of the array, follow the instructions in the "To Display the Date in Any Calendar" section.

4. Assign the calendar object to the Calendar property of the DateTimeFormatInfo object returned by the CultureInfo.DateTimeFormat property.

> **NOTE**
>
> The CultureInfo class also has a Calendar property. However, it is read-only and constant; it does not change to reflect the new default calendar assigned to the DateTimeFormatInfo.Calendar property.

5. Call either the ToString or the ToString method, and pass it the CultureInfo object whose default calendar was modified in the previous step.

To display the date in any calendar

1. Instantiate a calendar object derived from the Calendar class that represents the calendar to be used.

2. Determine which date and time elements should appear in the string representation of the date and time value.

3. For each date and time element that you want to display, call the calendar object's `Get`... method. The following methods are available:

   - GetYear, to display the year in the appropriate calendar.
   - GetMonth, to display the month in the appropriate calendar.
   - GetDayOfMonth, to display the number of the day of the month in the appropriate calendar.
   - GetHour, to display the hour of the day in the appropriate calendar.
   - GetMinute, to display the minutes in the hour in the appropriate calendar.
   - GetSecond, to display the seconds in the minute in the appropriate calendar.
   - GetMilliseconds , to display the milliseconds in the second in the appropriate calendar.

Example

The example displays a date using two different calendars. It displays the date after defining the Hijri calendar as the default calendar for the ar-JO culture, and displays the date using the Persian calendar, which is not supported as an optional calendar by the fa-IR culture.

```
using System;
using System.Globalization;

public class CalendarDates
{
   public static void Main()
   {
      HijriCalendar hijriCal = new HijriCalendar();
      CalendarUtility hijriUtil = new CalendarUtility(hijriCal);
      DateTime dateValue1 = new DateTime(1429, 6, 29, hijriCal);
      DateTimeOffset dateValue2 = new DateTimeOffset(dateValue1,
                                  TimeZoneInfo.Local.GetUtcOffset(dateValue1));
      CultureInfo jc = CultureInfo.CreateSpecificCulture("ar-JO");

      // Display the date using the Gregorian calendar.
      Console.WriteLine("Using the system default culture: {0}",
                        dateValue1.ToString("d"));
      // Display the date using the ar-JO culture's original default calendar.
      Console.WriteLine("Using the ar-JO culture's original default calendar: {0}",
                        dateValue1.ToString("d", jc));
      // Display the date using the Hijri calendar.
      Console.WriteLine("Using the ar-JO culture with Hijri as the default calendar:");
      // Display a Date value.
      Console.WriteLine(hijriUtil.DisplayDate(dateValue1, jc));
      // Display a DateTimeOffset value.
      Console.WriteLine(hijriUtil.DisplayDate(dateValue2, jc));

      Console.WriteLine();

      PersianCalendar persianCal = new PersianCalendar();
      CalendarUtility persianUtil = new CalendarUtility(persianCal);
      CultureInfo ic = CultureInfo.CreateSpecificCulture("fa-IR");
```

```
      // Display the date using the ir-FA culture's default calendar.
      Console.WriteLine("Using the ir-FA culture's default calendar: {0}",
                        dateValue1.ToString("d", ic));
      // Display a Date value.
      Console.WriteLine(persianUtil.DisplayDate(dateValue1, ic));
      // Display a DateTimeOffset value.
      Console.WriteLine(persianUtil.DisplayDate(dateValue2, ic));
   }
}

public class CalendarUtility
{
   private Calendar thisCalendar;
   private CultureInfo targetCulture;

   public CalendarUtility(Calendar cal)
   {
      this.thisCalendar = cal;
   }

   private bool CalendarExists(CultureInfo culture)
   {
      this.targetCulture = culture;
      return Array.Exists(this.targetCulture.OptionalCalendars,
                          this.HasSameName);
   }

   private bool HasSameName(Calendar cal)
   {
      if (cal.ToString() == thisCalendar.ToString())
         return true;
      else
         return false;
   }

   public string DisplayDate(DateTime dateToDisplay, CultureInfo culture)
   {
      DateTimeOffset displayOffsetDate = dateToDisplay;
      return DisplayDate(displayOffsetDate, culture);
   }

   public string DisplayDate(DateTimeOffset dateToDisplay,
                             CultureInfo culture)
   {
      string specifier = "yyyy/MM/dd";

      if (this.CalendarExists(culture))
      {
         Console.WriteLine("Displaying date in supported {0} calendar...",
                           this.thisCalendar.GetType().Name);
         culture.DateTimeFormat.Calendar = this.thisCalendar;
         return dateToDisplay.ToString(specifier, culture);
      }
      else
      {
         Console.WriteLine("Displaying date in unsupported {0} calendar...",
                           thisCalendar.GetType().Name);

         string separator = targetCulture.DateTimeFormat.DateSeparator;

         return thisCalendar.GetYear(dateToDisplay.DateTime).ToString("0000") +
                separator +
                thisCalendar.GetMonth(dateToDisplay.DateTime).ToString("00") +
                separator +
                thisCalendar.GetDayOfMonth(dateToDisplay.DateTime).ToString("00");
      }
   }
}
```

```
// The example displays the following output to the console:
//       Using the system default culture: 7/3/2008
//       Using the ar-JO culture's original default calendar: 03/07/2008
//       Using the ar-JO culture with Hijri as the default calendar:
//       Displaying date in supported HijriCalendar calendar...
//       1429/06/29
//       Displaying date in supported HijriCalendar calendar...
//       1429/06/29
//
//       Using the ir-FA culture's default calendar: 7/3/2008
//       Displaying date in unsupported PersianCalendar calendar...
//       1387/04/13
//       Displaying date in unsupported PersianCalendar calendar...
//       1387/04/13
```

```
Imports System.Globalization

Public Class CalendarDates
   Public Shared Sub Main()
      Dim hijriCal As New HijriCalendar()
      Dim hijriUtil As New CalendarUtility(hijriCal)
      Dim dateValue1 As Date = New Date(1429, 6, 29, hijriCal)
      Dim dateValue2 As DateTimeOffset = New DateTimeOffset(dateValue1, _
                                         TimeZoneInfo.Local.GetUtcOffset(dateValue1))
      Dim jc As CultureInfo = CultureInfo.CreateSpecificCulture("ar-JO")

      ' Display the date using the Gregorian calendar.
      Console.WriteLine("Using the system default culture: {0}", _
                        dateValue1.ToString("d"))
      ' Display the date using the ar-JO culture's original default calendar.
      Console.WriteLine("Using the ar-JO culture's original default calendar: {0}", _
                        dateValue1.ToString("d", jc))
      ' Display the date using the Hijri calendar.
      Console.WriteLine("Using the ar-JO culture with Hijri as the default calendar:")
      ' Display a Date value.
      Console.WriteLine(hijriUtil.DisplayDate(dateValue1, jc))
      ' Display a DateTimeOffset value.
      Console.WriteLine(hijriUtil.DisplayDate(dateValue2, jc))

      Console.WriteLine()

      Dim persianCal As New PersianCalendar()
      Dim persianUtil As New CalendarUtility(persianCal)
      Dim ic As CultureInfo = CultureInfo.CreateSpecificCulture("fa-IR")

      ' Display the date using the ir-FA culture's default calendar.
      Console.WriteLine("Using the ir-FA culture's default calendar: {0}", _
                        dateValue1.ToString("d", ic))
      ' Display a Date value.
      Console.WriteLine(persianUtil.DisplayDate(dateValue1, ic))
      ' Display a DateTimeOffset value.
      Console.WriteLine(persianUtil.DisplayDate(dateValue2, ic))
   End Sub
End Class

Public Class CalendarUtility
   Private thisCalendar As Calendar
   Private targetCulture As CultureInfo

   Public Sub New(cal As Calendar)
      Me.thisCalendar = cal
   End Sub

   Private Function CalendarExists(culture As CultureInfo) As Boolean
      Me.targetCulture = culture
      Return Array.Exists(Me.targetCulture.OptionalCalendars, _
                          AddressOf Me.HasSameName)
```

```
    End Function

    Private Function HasSameName(cal As Calendar) As Boolean
       If cal.ToString() = thisCalendar.ToString() Then
          Return True
       Else
          Return False
       End If
    End Function

    Public Function DisplayDate(dateToDisplay As Date, _
                                culture As CultureInfo) As String
       Dim displayOffsetDate As DateTimeOffset = dateToDisplay
       Return DisplayDate(displayOffsetDate, culture)
    End Function

    Public Function DisplayDate(dateToDisplay As DateTimeOffset, _
                                culture As CultureInfo) As String
       Dim specifier As String = "yyyy/MM/dd"

       If Me.CalendarExists(culture) Then
          Console.WriteLine("Displaying date in supported {0} calendar...", _
                            thisCalendar.GetType().Name)
          culture.DateTimeFormat.Calendar = Me.thisCalendar
          Return dateToDisplay.ToString(specifier, culture)
       Else
          Console.WriteLine("Displaying date in unsupported {0} calendar...", _
                            thisCalendar.GetType().Name)

          Dim separator As String = targetCulture.DateTimeFormat.DateSeparator

          Return thisCalendar.GetYear(dateToDisplay.DateTime).ToString("0000") & separator & _
                 thisCalendar.GetMonth(dateToDisplay.DateTime).ToString("00") & separator & _
                 thisCalendar.GetDayOfMonth(dateToDisplay.DateTime).ToString("00")
       End If
    End Function
End Class
' The example displays the following output to the console:
'       Using the system default culture: 7/3/2008
'       Using the ar-JO culture's original default calendar: 03/07/2008
'       Using the ar-JO culture with Hijri as the default calendar:
'       Displaying date in supported HijriCalendar calendar...
'       1429/06/29
'       Displaying date in supported HijriCalendar calendar...
'       1429/06/29
'
'       Using the ir-FA culture's default calendar: 7/3/2008
'       Displaying date in unsupported PersianCalendar calendar...
'       1387/04/13
'       Displaying date in unsupported PersianCalendar calendar...
'       1387/04/13
```

Each CultureInfo object can support one or more calendars, which are indicated by the OptionalCalendars property. One of these is designated as the culture's default calendar and is returned by the read-only CultureInfo.Calendar property. Another of the optional calendars can be designated as the default by assigning a Calendar object that represents that calendar to the DateTimeFormatInfo.Calendar property returned by the CultureInfo.DateTimeFormat property. However, some calendars, such as the Persian calendar represented by the PersianCalendar class, do not serve as optional calendars for any culture.

The example defines a reusable calendar utility class, `CalendarUtility`, to handle many of the details of generating the string representation of a date using a particular calendar. The `CalendarUtility` class has the following members:

- A parameterized constructor whose single parameter is a Calendar object in which a date is to be represented. This is assigned to a private field of the class.

- `CalendarExists`, a private method that returns a Boolean value indicating whether the calendar represented by the `CalendarUtility` object is supported by the CultureInfo object that is passed to the method as a parameter. The method wraps a call to the Array.Exists method, to which it passes the CultureInfo.OptionalCalendars array.

- `HasSameName`, a private method assigned to the Predicate<T> delegate that is passed as a parameter to the Array.Exists method. Each member of the array is passed to the method until the method returns `true`. The method determines whether the name of an optional calendar is the same as the calendar represented by the `CalendarUtility` object.

- `DisplayDate`, an overloaded public method that is passed two parameters: either a DateTime or DateTimeOffset value to express in the calendar represented by the `CalendarUtility` object; and the culture whose formatting rules are to be used. Its behavior in returning the string representation of a date depends on whether the target calendar is supported by the culture whose formatting rules are to be used.

Regardless of the calendar used to create a DateTime or DateTimeOffset value in this example, that value is typically expressed as a Gregorian date. This is because the DateTime and DateTimeOffset types do not preserve any calendar information. Internally, they are represented as the number of ticks that have elapsed since midnight of January 1, 0001. The interpretation of that number depends on the calendar. For most cultures, the default calendar is the Gregorian calendar.

Compiling the Code

This example requires a reference to System.Core.dll.

Compile the code at the command line using csc.exe or vb.exe. To compile the code in Visual Studio, put it in a console application project template.

See also

- Performing Formatting Operations

Manipulating Strings in .NET

5/2/2018 • 2 minutes to read • Edit Online

.NET provides an extensive set of routines that enable you to efficiently create, compare, and modify strings as well as rapidly parse large amounts of text and data to search for, remove, and replace text patterns.

In This Section

Best Practices for Using Strings
Examines string-sorting, comparison, and casing methods in .NET, and provides recommendations for selecting a string-handling method .

.NET Regular Expressions
Provides detailed information about .NET regular expressions, including language elements, regular expression behavior, and examples.

Basic String Operations
Describes string operations provided by the System.String and System.Text.StringBuilder classes, including creating new strings from arrays of bytes, comparing string values, and modifying existing strings.

Related Sections

Type Conversion in .NET
Explains the techniques and rules used to convert types using .NET.

Formatting Types
Provides how to use the base class library to implement formatting, how to format numeric types, how to format string types, and how to format for a specific culture.

Parsing Strings
Describes how to initialize objects to the values described by string representations of those objects. Parsing is the inverse operation of formatting.

Best Practices for Using Strings in .NET

9/14/2018 • 31 minutes to read • Edit Online

.NET provides extensive support for developing localized and globalized applications, and makes it easy to apply the conventions of either the current culture or a specific culture when performing common operations such as sorting and displaying strings. But sorting or comparing strings is not always a culture-sensitive operation. For example, strings that are used internally by an application typically should be handled identically across all cultures. When culturally independent string data, such as XML tags, HTML tags, user names, file paths, and the names of system objects, are interpreted as if they were culture-sensitive, application code can be subject to subtle bugs, poor performance, and, in some cases, security issues.

This topic examines the string sorting, comparison, and casing methods in .NET, presents recommendations for selecting an appropriate string-handling method, and provides additional information about string-handling methods. It also examines how formatted data, such as numeric data and date and time data, is handled for display and for storage.

This topic contains the following sections:

Recommendations for String Usage

When you develop with .NET, follow these simple recommendations when you use strings:

- Use overloads that explicitly specify the string comparison rules for string operations. Typically, this involves calling a method overload that has a parameter of type StringComparison.
- Use StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase for comparisons as your safe default for culture-agnostic string matching.
- Use comparisons with StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase for better performance.
- Use string operations that are based on StringComparison.CurrentCulture when you display output to the user.
- Use the non-linguistic StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase values instead of string operations based on CultureInfo.InvariantCulture when the comparison is linguistically irrelevant (symbolic, for example).
- Use the String.ToUpperInvariant method instead of the String.ToLowerInvariant method when you normalize strings for comparison.
- Use an overload of the String.Equals method to test whether two strings are equal.

- Use the String.Compare and String.CompareTo methods to sort strings, not to check for equality.
- Use culture-sensitive formatting to display non-string data, such as numbers and dates, in a user interface. Use formatting with the invariant culture to persist non-string data in string form.

Avoid the following practices when you use strings:

- Do not use overloads that do not explicitly or implicitly specify the string comparison rules for string operations.
- Do not use string operations based on StringComparison.InvariantCulture in most cases. One of the few exceptions is when you are persisting linguistically meaningful but culturally agnostic data.
- Do not use an overload of the String.Compare or CompareTo method and test for a return value of zero to determine whether two strings are equal.
- Do not use culture-sensitive formatting to persist numeric data or date and time data in string form.

Back to top

Specifying String Comparisons Explicitly

Most of the string manipulation methods in .NET are overloaded. Typically, one or more overloads accept default settings, whereas others accept no defaults and instead define the precise way in which strings are to be compared or manipulated. Most of the methods that do not rely on defaults include a parameter of type StringComparison, which is an enumeration that explicitly specifies rules for string comparison by culture and case. The following table describes the StringComparison enumeration members.

STRINGCOMPARISON MEMBER	DESCRIPTION
CurrentCulture	Performs a case-sensitive comparison using the current culture.
CurrentCultureIgnoreCase	Performs a case-insensitive comparison using the current culture.
InvariantCulture	Performs a case-sensitive comparison using the invariant culture.
InvariantCultureIgnoreCase	Performs a case-insensitive comparison using the invariant culture.
Ordinal	Performs an ordinal comparison.
OrdinalIgnoreCase	Performs a case-insensitive ordinal comparison.

For example, the IndexOf method, which returns the index of a substring in a String object that matches either a character or a string, has nine overloads:

- IndexOf(Char), IndexOf(Char, Int32), and IndexOf(Char, Int32, Int32), which by default perform an ordinal (case-sensitive and culture-insensitive) search for a character in the string.
- IndexOf(String), IndexOf(String, Int32), and IndexOf(String, Int32, Int32), which by default perform a case-sensitive and culture-sensitive search for a substring in the string.
- IndexOf(String, StringComparison), IndexOf(String, Int32, StringComparison), and IndexOf(String, Int32, Int32, StringComparison), which include a parameter of type StringComparison that allows the form of the comparison to be specified.

We recommend that you select an overload that does not use default values, for the following reasons:

- Some overloads with default parameters (those that search for a Char in the string instance) perform an ordinal comparison, whereas others (those that search for a string in the string instance) are culture-sensitive. It is difficult to remember which method uses which default value, and easy to confuse the overloads.
- The intent of the code that relies on default values for method calls is not clear. In the following example, which relies on defaults, it is difficult to know whether the developer actually intended an ordinal or a linguistic comparison of two strings, or whether a case difference between `protocol` and "http" might cause the test for equality to return `false`.

```
string protocol = GetProtocol(url);
if (String.Equals(protocol, "http")) {
   // ...Code to handle HTTP protocol.
}
else {
   throw new InvalidOperationException();
}
```

```
Dim protocol As String = GetProtocol(url)
If String.Equals(protocol, "http") Then
   ' ...Code to handle HTTP protocol.
Else
   Throw New InvalidOperationException()
End If
```

In general, we recommend that you call a method that does not rely on defaults, because it makes the intent of the code unambiguous. This, in turn, makes the code more readable and easier to debug and maintain. The following example addresses the questions raised about the previous example. It makes it clear that ordinal comparison is used and that differences in case are ignored.

```
string protocol = GetProtocol(url);
if (String.Equals(protocol, "http", StringComparison.OrdinalIgnoreCase)) {
   // ...Code to handle HTTP protocol.
}
else {
   throw new InvalidOperationException();
}
```

```
Dim protocol As String = GetProtocol(url)
If String.Equals(protocol, "http", StringComparison.OrdinalIgnoreCase) Then
   ' ...Code to handle HTTP protocol.
Else
   Throw New InvalidOperationException()
End If
```

Back to top

The Details of String Comparison

String comparison is the heart of many string-related operations, particularly sorting and testing for equality. Strings sort in a determined order: If "my" appears before "string" in a sorted list of strings, "my" must compare less than or equal to "string". Additionally, comparison implicitly defines equality. The comparison operation returns zero for strings it deems equal. A good interpretation is that neither string is less than the other. Most meaningful operations involving strings include one or both of these procedures: comparing with another string, and executing a well-defined sort operation.

> **NOTE**
>
> You can download the Sorting Weight Tables, a set of text files that contain information on the character weights used in sorting and comparison operations for Windows operating systems, and the Default Unicode Collation Element Table, the latest version of the sort weight table for Linux and macOS. The specific version of the sort weight table on Linux and macOS depends on the version of the International Components for Unicode libraries installed on the system. For information on ICU versions and the Unicode versions that they implement, see Downloading ICU.

However, evaluating two strings for equality or sort order does not yield a single, correct result; the outcome depends on the criteria used to compare the strings. In particular, string comparisons that are ordinal or that are based on the casing and sorting conventions of the current culture or the invariant culture (a locale-agnostic culture based on the English language) may produce different results.

In addition, string comparisons using different versions of .NET or using .NET on different operating systems or operating system versions may return different results. For more information, see Strings and the Unicode Standard.

String Comparisons that Use the Current Culture

One criterion involves using the conventions of the current culture when comparing strings. Comparisons that are based on the current culture use the thread's current culture or locale. If the culture is not set by the user, it defaults to the setting in the **Regional Options** window in Control Panel. You should always use comparisons that are based on the current culture when data is linguistically relevant, and when it reflects culture-sensitive user interaction.

However, comparison and casing behavior in .NET changes when the culture changes. This happens when an application executes on a computer that has a different culture than the computer on which the application was developed, or when the executing thread changes its culture. This behavior is intentional, but it remains non-obvious to many developers. The following example illustrates differences in sort order between the U.S. English ("en-US") and Swedish ("sv-SE") cultures. Note that the words "ångström", "Windows", and "Visual Studio" appear in different positions in the sorted string arrays.

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string[] values= { "able", "ångström", "apple", "Æble",
                         "Windows", "Visual Studio" };
      Array.Sort(values);
      DisplayArray(values);

      // Change culture to Swedish (Sweden).
      string originalCulture = CultureInfo.CurrentCulture.Name;
      Thread.CurrentThread.CurrentCulture = new CultureInfo("sv-SE");
      Array.Sort(values);
      DisplayArray(values);

      // Restore the original culture.
      Thread.CurrentThread.CurrentCulture = new CultureInfo(originalCulture);
    }

    private static void DisplayArray(string[] values)
    {
      Console.WriteLine("Sorting using the {0} culture:",
                        CultureInfo.CurrentCulture.Name);
      foreach (string value in values)
         Console.WriteLine("   {0}", value);

      Console.WriteLine();
    }
}
// The example displays the following output:
//       Sorting using the en-US culture:
//          able
//          Æble
//          ångström
//          apple
//          Visual Studio
//          Windows
//
//       Sorting using the sv-SE culture:
//          able
//          Æble
//          apple
//          Windows
//          Visual Studio
//          ångström
```

```vb
Imports System.Globalization
Imports System.Threading

Module Example
   Public Sub Main()
      Dim values() As String = { "able", "ångström", "apple", _
                                 "Æble", "Windows", "Visual Studio" }
      Array.Sort(values)
      DisplayArray(values)

      ' Change culture to Swedish (Sweden).
      Dim originalCulture As String = CultureInfo.CurrentCulture.Name
      Thread.CurrentThread.CurrentCulture = New CultureInfo("sv-SE")
      Array.Sort(values)
      DisplayArray(values)

      ' Restore the original culture.
      Thread.CurrentThread.CurrentCulture = New CultureInfo(originalCulture)
    End Sub

    Private Sub DisplayArray(values() As String)
      Console.WRiteLine("Sorting using the {0} culture:", _
                        CultureInfo.CurrentCulture.Name)
      For Each value As String In values
         Console.WriteLine("   {0}", value)
      Next
      Console.WriteLine()
    End Sub
End Module
' The example displays the following output:
'       Sorting using the en-US culture:
'          able
'          Æble
'          ångström
'          apple
'          Visual Studio
'          Windows
'
'       Sorting using the sv-SE culture:
'          able
'          Æble
'          apple
'          Windows
'          Visual Studio
'          ångström
```

Case-insensitive comparisons that use the current culture are the same as culture-sensitive comparisons, except that they ignore case as dictated by the thread's current culture. This behavior may manifest itself in sort orders as well.

Comparisons that use current culture semantics are the default for the following methods:

- String.Compare overloads that do not include a StringComparison parameter.
- String.CompareTo overloads.
- The default String.StartsWith(String) method, and the String.StartsWith(String, Boolean, CultureInfo) method with a `null` CultureInfo parameter.
- The default String.EndsWith(String) method, and the String.EndsWith(String, Boolean, CultureInfo) method with a `null` CultureInfo parameter.
- String.IndexOf overloads that accept a String as a search parameter and that do not have a StringComparison parameter.

- String.LastIndexOf overloads that accept a String as a search parameter and that do not have a StringComparison parameter.

In any case, we recommend that you call an overload that has a StringComparison parameter to make the intent of the method call clear.

Subtle and not so subtle bugs can emerge when non-linguistic string data is interpreted linguistically, or when string data from a particular culture is interpreted using the conventions of another culture. The canonical example is the Turkish-I problem.

For nearly all Latin alphabets, including U.S. English, the character "i" (\u0069) is the lowercase version of the character "I" (\u0049). This casing rule quickly becomes the default for someone programming in such a culture. However, the Turkish ("tr-TR") alphabet includes an "I with a dot" character "İ" (\u0130), which is the capital version of "i". Turkish also includes a lowercase "i without a dot" character, "ı" (\u0131), which capitalizes to "I". This behavior occurs in the Azerbaijani ("az") culture as well.

Therefore, assumptions made about capitalizing "i" or lowercasing "I" are not valid among all cultures. If you use the default overloads for string comparison routines, they will be subject to variance between cultures. If the data to be compared is non-linguistic, using the default overloads can produce undesirable results, as the following attempt to perform a case-insensitive comparison of the strings "file" and "FILE" illustrates.

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string fileUrl = "file";
      Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
      Console.WriteLine("Culture = {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      Console.WriteLine("(file == FILE) = {0}",
                       fileUrl.StartsWith("FILE", true, null));
      Console.WriteLine();

      Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");
      Console.WriteLine("Culture = {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      Console.WriteLine("(file == FILE) = {0}",
                        fileUrl.StartsWith("FILE", true, null));
   }
}
// The example displays the following output:
//       Culture = English (United States)
//       (file == FILE) = True
//
//       Culture = Turkish (Turkey)
//       (file == FILE) = False
```

```
Imports System.Globalization
Imports System.Threading

Module Example
   Public Sub Main()
      Dim fileUrl = "file"
      Thread.CurrentThread.CurrentCulture = New CultureInfo("en-US")
      Console.WriteLine("Culture = {0}", _
                        Thread.CurrentThread.CurrentCulture.DisplayName)
      Console.WriteLine("(file == FILE) = {0}", _
                       fileUrl.StartsWith("FILE", True, Nothing))
      Console.WriteLine()

      Thread.CurrentThread.CurrentCulture = New CultureInfo("tr-TR")
      Console.WriteLine("Culture = {0}", _
                        Thread.CurrentThread.CurrentCulture.DisplayName)
      Console.WriteLine("(file == FILE) = {0}", _
                        fileUrl.StartsWith("FILE", True, Nothing))
   End Sub
End Module
' The example displays the following output:
'       Culture = English (United States)
'       (file == FILE) = True
'
'       Culture = Turkish (Turkey)
'       (file == FILE) = False
```

This comparison could cause significant problems if the culture is inadvertently used in security-sensitive settings, as in the following example. A method call such as `IsFileURI("file:")` returns `true` if the current culture is U.S. English, but `false` if the current culture is Turkish. Thus, on Turkish systems, someone could circumvent security measures that block access to case-insensitive URIs that begin with "FILE:".

```
public static bool IsFileURI(String path)
{
   return path.StartsWith("FILE:", true, null);
}
```

```
Public Shared Function IsFileURI(path As String) As Boolean
   Return path.StartsWith("FILE:", True, Nothing)
End Function
```

In this case, because "file:" is meant to be interpreted as a non-linguistic, culture-insensitive identifier, the code should instead be written as shown in the following example.

```
public static bool IsFileURI(string path)
{
   return path.StartsWith("FILE:", StringComparison.OrdinalIgnoreCase);
}
```

```
Public Shared Function IsFileURI(path As String) As Boolean
    Return path.StartsWith("FILE:", StringComparison.OrdinalIgnoreCase)
End Function
```

Ordinal String Operations

Specifying the StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase value in a method call signifies a non-linguistic comparison in which the features of natural languages are ignored. Methods that are invoked with these StringComparison values base string operation decisions on simple byte comparisons instead of casing or equivalence tables that are parameterized by culture. In most cases, this approach best fits the intended interpretation of strings while making code faster and more reliable.

Ordinal comparisons are string comparisons in which each byte of each string is compared without linguistic interpretation; for example, "windows" does not match "Windows". This is essentially a call to the C runtime `strcmp` function. Use this comparison when the context dictates that strings should be matched exactly or demands conservative matching policy. Additionally, ordinal comparison is the fastest comparison operation because it applies no linguistic rules when determining a result.

Strings in .NET can contain embedded null characters. One of the clearest differences between ordinal and culture-sensitive comparison (including comparisons that use the invariant culture) concerns the handling of embedded null characters in a string. These characters are ignored when you use the String.Compare and String.Equals methods to perform culture-sensitive comparisons (including comparisons that use the invariant culture). As a result, in culture-sensitive comparisons, strings that contain embedded null characters can be considered equal to strings that do not.

> **IMPORTANT**
>
> Although string comparison methods disregard embedded null characters, string search methods such as String.Contains, String.EndsWith, String.IndexOf, String.LastIndexOf, and String.StartsWith do not.

The following example performs a culture-sensitive comparison of the string "Aa" with a similar string that contains several embedded null characters between "A" and "a", and shows how the two strings are considered equal.

```
using System;

public class Example
{
   public static void Main()
   {
      string str1 = "Aa";
      string str2 = "A" + new String('\u0000', 3) + "a";
      Console.WriteLine("Comparing '{0}' ({1}) and '{2}' ({3}):",
                        str1, ShowBytes(str1), str2, ShowBytes(str2));
      Console.WriteLine("   With String.Compare:");
      Console.WriteLine("      Current Culture: {0}",
                        String.Compare(str1, str2, StringComparison.CurrentCulture));
      Console.WriteLine("      Invariant Culture: {0}",
                        String.Compare(str1, str2, StringComparison.InvariantCulture));

      Console.WriteLine("   With String.Equals:");
      Console.WriteLine("      Current Culture: {0}",
                        String.Equals(str1, str2, StringComparison.CurrentCulture));
      Console.WriteLine("      Invariant Culture: {0}",
                        String.Equals(str1, str2, StringComparison.InvariantCulture));
   }

   private static string ShowBytes(string str)
   {
      string hexString = String.Empty;
      for (int ctr = 0; ctr < str.Length; ctr++)
      {
         string result = String.Empty;
         result = Convert.ToInt32(str[ctr]).ToString("X4");
         result = " " + result.Substring(0,2) + " " + result.Substring(2, 2);
         hexString += result;
      }
      return hexString.Trim();
   }
}
// The example displays the following output:
//    Comparing 'Aa' (00 41 00 61) and 'A   a' (00 41 00 00 00 00 00 00 00 61):
//       With String.Compare:
//          Current Culture: 0
//          Invariant Culture: 0
//       With String.Equals:
//          Current Culture: True
//          Invariant Culture: True
```

```vb
Module Example
   Public Sub Main()
      Dim str1 As String = "Aa"
      Dim str2 As String = "A" + New String(Convert.ToChar(0), 3) + "a"
      Console.WriteLine("Comparing '{0}' ({1}) and '{2}' ({3}):", _
                        str1, ShowBytes(str1), str2, ShowBytes(str2))
      Console.WriteLine("   With String.Compare:")
      Console.WriteLine("      Current Culture: {0}", _
                        String.Compare(str1, str2, StringComparison.CurrentCulture))
      Console.WriteLine("      Invariant Culture: {0}", _
                        String.Compare(str1, str2, StringComparison.InvariantCulture))

      Console.WriteLine("   With String.Equals:")
      Console.WriteLine("      Current Culture: {0}", _
                        String.Equals(str1, str2, StringComparison.CurrentCulture))
      Console.WriteLine("      Invariant Culture: {0}", _
                        String.Equals(str1, str2, StringComparison.InvariantCulture))
   End Sub

   Private Function ShowBytes(str As String) As String
      Dim hexString As String = String.Empty
      For ctr As Integer = 0 To str.Length - 1
         Dim result As String = String.Empty
         result = Convert.ToInt32(str.Chars(ctr)).ToString("X4")
         result = " " + result.Substring(0,2) + " " + result.Substring(2, 2)
         hexString += result
      Next
      Return hexString.Trim()
   End Function
End Module
```

However, the strings are not considered equal when you use ordinal comparison, as the following example shows.

```csharp
Console.WriteLine("Comparing '{0}' ({1}) and '{2}' ({3}):",
                  str1, ShowBytes(str1), str2, ShowBytes(str2));
Console.WriteLine("   With String.Compare:");
Console.WriteLine("      Ordinal: {0}",
                  String.Compare(str1, str2, StringComparison.Ordinal));

Console.WriteLine("   With String.Equals:");
Console.WriteLine("      Ordinal: {0}",
                  String.Equals(str1, str2, StringComparison.Ordinal));
// The example displays the following output:
//    Comparing 'Aa' (00 41 00 61) and 'A   a' (00 41 00 00 00 00 00 00 00 61):
//       With String.Compare:
//          Ordinal: 97
//       With String.Equals:
//          Ordinal: False
```

```
Console.WriteLine("Comparing '{0}' ({1}) and '{2}' ({3}):", _
                  str1, ShowBytes(str1), str2, ShowBytes(str2))
Console.WriteLine("   With String.Compare:")
Console.WriteLine("      Ordinal: {0}", _
                  String.Compare(str1, str2, StringComparison.Ordinal))

Console.WriteLine("   With String.Equals:")
Console.WriteLine("      Ordinal: {0}", _
                  String.Equals(str1, str2, StringComparison.Ordinal))
' The example displays the following output:
'    Comparing 'Aa' (00 41 00 61) and 'A   a' (00 41 00 00 00 00 00 00 00 61):
'       With String.Compare:
'          Ordinal: 97
'       With String.Equals:
'          Ordinal: False
```

Case-insensitive ordinal comparisons are the next most conservative approach. These comparisons ignore most casing; for example, "windows" matches "Windows". When dealing with ASCII characters, this policy is equivalent to StringComparison.Ordinal, except that it ignores the usual ASCII casing. Therefore, any character in A, Z matches the corresponding character in a,z. Casing outside the ASCII range uses the invariant culture's tables. Therefore, the following comparison:

```
String.Compare(strA, strB, StringComparison.OrdinalIgnoreCase);
```

```
String.Compare(strA, strB, StringComparison.OrdinalIgnoreCase)
```

is equivalent to (but faster than) this comparison:

```
String.Compare(strA.ToUpperInvariant(), strB.ToUpperInvariant(),
               StringComparison.Ordinal);
```

```
String.Compare(strA.ToUpperInvariant(), strB.ToUpperInvariant(),
               StringComparison.Ordinal)
```

These comparisons are still very fast.

NOTE

The string behavior of the file system, registry keys and values, and environment variables is best represented by StringComparison.OrdinalIgnoreCase.

Both StringComparison.Ordinal and StringComparison.OrdinalIgnoreCase use the binary values directly, and are best suited for matching. When you are not sure about your comparison settings, use one of these two values. However, because they perform a byte-by-byte comparison, they do not sort by a linguistic sort order (like an English dictionary) but by a binary sort order. The results may look odd in most contexts if displayed to users.

Ordinal semantics are the default for String.Equals overloads that do not include a StringComparison argument (including the equality operator). In any case, we recommend that you call an overload that has a StringComparison parameter.

String Operations that Use the Invariant Culture

Comparisons with the invariant culture use the CompareInfo property returned by the static CultureInfo.InvariantCulture property. This behavior is the same on all systems; it translates any characters outside its range into what it believes are equivalent invariant characters. This policy can be useful for maintaining one set of string behavior across cultures, but it often provides unexpected results.

Case-insensitive comparisons with the invariant culture use the static CompareInfo property returned by the static CultureInfo.InvariantCulture property for comparison information as well. Any case differences among these translated characters are ignored.

Comparisons that use StringComparison.InvariantCulture and StringComparison.Ordinal work identically on ASCII strings. However, StringComparison.InvariantCulture makes linguistic decisions that might not be appropriate for strings that have to be interpreted as a set of bytes. The `CultureInfo.InvariantCulture.CompareInfo` object makes the Compare method interpret certain sets of characters as equivalent. For example, the following equivalence is valid under the invariant culture:

InvariantCulture: a + ̊ = å

The LATIN SMALL LETTER A character "a" (\u0061), when it is next to the COMBINING RING ABOVE character "+ " ̊" (\u030a), is interpreted as the LATIN SMALL LETTER A WITH RING ABOVE character "å" (\u00e5). As the following example shows, this behavior differs from ordinal comparison.

```
string separated = "\u0061\u030a";
string combined = "\u00e5";

Console.WriteLine("Equal sort weight of {0} and {1} using InvariantCulture: {2}",
                  separated, combined,
                  String.Compare(separated, combined,
                                 StringComparison.InvariantCulture) == 0);

Console.WriteLine("Equal sort weight of {0} and {1} using Ordinal: {2}",
                  separated, combined,
                  String.Compare(separated, combined,
                                 StringComparison.Ordinal) == 0);
// The example displays the following output:
//    Equal sort weight of a° and å using InvariantCulture: True
//    Equal sort weight of a° and å using Ordinal: False
```

```
Dim separated As String = ChrW(&h61) + ChrW(&h30a)
Dim combined As String = ChrW(&he5)

Console.WriteLine("Equal sort weight of {0} and {1} using InvariantCulture: {2}", _
                  separated, combined, _
                  String.Compare(separated, combined, _
                                 StringComparison.InvariantCulture) = 0)

Console.WriteLine("Equal sort weight of {0} and {1} using Ordinal: {2}", _
                  separated, combined, _
                  String.Compare(separated, combined, _
                                 StringComparison.Ordinal) = 0)
' The example displays the following output:
'    Equal sort weight of a° and å using InvariantCulture: True
'    Equal sort weight of a° and å using Ordinal: False
```

When interpreting file names, cookies, or anything else where a combination such as "å" can appear, ordinal comparisons still offer the most transparent and fitting behavior.

On balance, the invariant culture has very few properties that make it useful for comparison. It does comparison in a linguistically relevant manner, which prevents it from guaranteeing full symbolic equivalence, but it is not the choice for display in any culture. One of the few reasons to use StringComparison.InvariantCulture for comparison is to persist ordered data for a cross-culturally identical display. For example, if a large data file that contains a list of sorted identifiers for display accompanies an application, adding to this list would require an insertion with invariant-style sorting.

Back to top

Choosing a StringComparison Member for Your Method Call

The following table outlines the mapping from semantic string context to a StringComparison enumeration member.

DATA	BEHAVIOR	CORRESPONDING SYSTEM.STRINGCOMPARISON VALUE
Case-sensitive internal identifiers. Case-sensitive identifiers in standards such as XML and HTTP. Case-sensitive security-related settings.	A non-linguistic identifier, where bytes match exactly.	Ordinal
Case-insensitive internal identifiers. Case-insensitive identifiers in standards such as XML and HTTP. File paths. Registry keys and values. Environment variables. Resource identifiers (for example, handle names). Case-insensitive security-related settings.	A non-linguistic identifier, where case is irrelevant; especially data stored in most Windows system services.	OrdinalIgnoreCase
Some persisted, linguistically relevant data. Display of linguistic data that requires a fixed sort order.	Culturally agnostic data that still is linguistically relevant.	InvariantCulture -or- InvariantCultureIgnoreCase
Data displayed to the user. Most user input.	Data that requires local linguistic customs.	CurrentCulture -or- CurrentCultureIgnoreCase

Back to top

Common String Comparison Methods in .NET

The following sections describe the methods that are most commonly used for string comparison.

String.Compare

Default interpretation: StringComparison.CurrentCulture.

As the operation most central to string interpretation, all instances of these method calls should be examined to determine whether strings should be interpreted according to the current culture, or dissociated from the culture (symbolically). Typically, it is the latter, and a StringComparison.Ordinal comparison should be used instead.

The System.Globalization.CompareInfo class, which is returned by the CultureInfo.CompareInfo property, also includes a Compare method that provides a large number of matching options (ordinal, ignoring white space, ignoring kana type, and so on) by means of the CompareOptions flag enumeration.

String.CompareTo

Default interpretation: StringComparison.CurrentCulture.

This method does not currently offer an overload that specifies a StringComparison type. It is usually possible to convert this method to the recommended String.Compare(String, String, StringComparison) form.

Types that implement the IComparable and IComparable<T> interfaces implement this method. Because it does not offer the option of a StringComparison parameter, implementing types often let the user specify a StringComparer in their constructor. The following example defines a `FileName` class whose class constructor includes a StringComparer parameter. This StringComparer object is then used in the `FileName.CompareTo` method.

```
using System;

public class FileName : IComparable
{
   string fname;
   StringComparer comparer;

   public FileName(string name, StringComparer comparer)
   {
      if (String.IsNullOrEmpty(name))
         throw new ArgumentNullException("name");

      this.fname = name;

      if (comparer != null)
         this.comparer = comparer;
      else
         this.comparer = StringComparer.OrdinalIgnoreCase;
   }

   public string Name
   {
      get { return fname; }
   }

   public int CompareTo(object obj)
   {
      if (obj == null) return 1;

      if (! (obj is FileName))
         return comparer.Compare(this.fname, obj.ToString());
      else
         return comparer.Compare(this.fname, ((FileName) obj).Name);
   }
}
```

```vb
Public Class FileName : Implements IComparable
   Dim fname As String
   Dim comparer As StringComparer

   Public Sub New(name As String, comparer As StringComparer)
      If String.IsNullOrEmpty(name) Then
         Throw New ArgumentNullException("name")
      End If

      Me.fname = name

      If comparer IsNot Nothing Then
         Me.comparer = comparer
      Else
         Me.comparer = StringComparer.OrdinalIgnoreCase
      End If
   End Sub

   Public ReadOnly Property Name As String
      Get
         Return fname
      End Get
   End Property

   Public Function CompareTo(obj As Object) As Integer _
          Implements IComparable.CompareTo
      If obj Is Nothing Then Return 1

      If Not TypeOf obj Is FileName Then
         obj = obj.ToString()
      Else
         obj = CType(obj, FileName).Name
      End If
      Return comparer.Compare(Me.fname, obj)
   End Function
End Class
```

String.Equals

Default interpretation: StringComparison.Ordinal.

The String class lets you test for equality by calling either the static or instance Equals method overloads, or by using the static equality operator. The overloads and operator use ordinal comparison by default. However, we still recommend that you call an overload that explicitly specifies the StringComparison type even if you want to perform an ordinal comparison; this makes it easier to search code for a certain string interpretation.

String.ToUpper and String.ToLower

Default interpretation: StringComparison.CurrentCulture.

You should be careful when you use these methods, because forcing a string to a uppercase or lowercase is often used as a small normalization for comparing strings regardless of case. If so, consider using a case-insensitive comparison.

The String.ToUpperInvariant and String.ToLowerInvariant methods are also available. ToUpperInvariant is the standard way to normalize case. Comparisons made using StringComparison.OrdinalIgnoreCase are behaviorally the composition of two calls: calling ToUpperInvariant on both string arguments, and doing a comparison using StringComparison.Ordinal.

Overloads are also available for converting to uppercase and lowercase in a specific culture, by passing a CultureInfo object that represents that culture to the method.

Char.ToUpper and Char.ToLower

Default interpretation: StringComparison.CurrentCulture.

These methods work similarly to the String.ToUpper and String.ToLower methods described in the previous section.

String.StartsWith and String.EndsWith

Default interpretation: StringComparison.CurrentCulture.

By default, both of these methods perform a culture-sensitive comparison.

String.IndexOf and String.LastIndexOf

Default interpretation: StringComparison.CurrentCulture.

There is a lack of consistency in how the default overloads of these methods perform comparisons. All String.IndexOf and String.LastIndexOf methods that include a Char parameter perform an ordinal comparison, but the default String.IndexOf and String.LastIndexOf methods that include a String parameter perform a culture-sensitive comparison.

If you call the String.IndexOf(String) or String.LastIndexOf(String) method and pass it a string to locate in the current instance, we recommend that you call an overload that explicitly specifies the StringComparison type. The overloads that include a Char argument do not allow you to specify a StringComparison type.

Back to top

Methods that Perform String Comparison Indirectly

Some non-string methods that have string comparison as a central operation use the StringComparer type. The StringComparer class includes six static properties that return StringComparer instances whose StringComparer.Compare methods perform the following types of string comparisons:

- Culture-sensitive string comparisons using the current culture. This StringComparer object is returned by the StringComparer.CurrentCulture property.
- Case-insensitive comparisons using the current culture. This StringComparer object is returned by the StringComparer.CurrentCultureIgnoreCase property.
- Culture-insensitive comparisons using the word comparison rules of the invariant culture. This StringComparer object is returned by the StringComparer.InvariantCulture property.
- Case-insensitive and culture-insensitive comparisons using the word comparison rules of the invariant culture. This StringComparer object is returned by the StringComparer.InvariantCultureIgnoreCase property.
- Ordinal comparison. This StringComparer object is returned by the StringComparer.Ordinal property.
- Case-insensitive ordinal comparison. This StringComparer object is returned by the StringComparer.OrdinalIgnoreCase property.

Array.Sort and Array.BinarySearch

Default interpretation: StringComparison.CurrentCulture.

When you store any data in a collection, or read persisted data from a file or database into a collection, switching the current culture can invalidate the invariants in the collection. The Array.BinarySearch method assumes that the elements in the array to be searched are already sorted. To sort any string element in the array, the Array.Sort method calls the String.Compare method to order individual elements. Using a culture-sensitive comparer can be dangerous if the culture changes between the time that the array is sorted and its contents are searched. For example, in the following code, storage and retrieval operate on the comparer that is provided implicitly by the `Thread.CurrentThread.CurrentCulture` property. If the culture can change between the calls to `StoreNames` and

`DoesNameExist`, and especially if the array contents are persisted somewhere between the two method calls, the binary search may fail.

```
// Incorrect.
string []storedNames;

public void StoreNames(string [] names)
{
   int index = 0;
   storedNames = new string[names.Length];

   foreach (string name in names)
   {
      this.storedNames[index++] = name;
   }

   Array.Sort(names); // Line A.
}

public bool DoesNameExist(string name)
{
   return (Array.BinarySearch(this.storedNames, name) >= 0);  // Line B.
}
```

```
' Incorrect.
Dim storedNames() As String

Public Sub StoreNames(names() As String)
   Dim index As Integer = 0
   ReDim storedNames(names.Length - 1)

   For Each name As String In names
      Me.storedNames(index) = name
      index+= 1
   Next

   Array.Sort(names)          ' Line A.
End Sub

Public Function DoesNameExist(name As String) As Boolean
   Return Array.BinarySearch(Me.storedNames, name) >= 0      ' Line B.
End Function
```

A recommended variation appears in the following example, which uses the same ordinal (culture-insensitive) comparison method both to sort and to search the array. The change code is reflected in the lines labeled `Line A` and `Line B` in the two examples.

```csharp
// Correct.
string []storedNames;

public void StoreNames(string [] names)
{
   int index = 0;
   storedNames = new string[names.Length];

   foreach (string name in names)
   {
      this.storedNames[index++] = name;
   }

   Array.Sort(names, StringComparer.Ordinal);  // Line A.
}

public bool DoesNameExist(string name)
{
   return (Array.BinarySearch(this.storedNames, name, StringComparer.Ordinal) >= 0);  // Line B.
}
```

```vb
' Correct.
Dim storedNames() As String

Public Sub StoreNames(names() As String)
   Dim index As Integer = 0
   ReDim storedNames(names.Length - 1)

   For Each name As String In names
      Me.storedNames(index) = name
      index+= 1
   Next

   Array.Sort(names, StringComparer.Ordinal)           ' Line A.
End Sub

Public Function DoesNameExist(name As String) As Boolean
   Return Array.BinarySearch(Me.storedNames, name, StringComparer.Ordinal) >= 0      ' Line B.
End Function
```

If this data is persisted and moved across cultures, and sorting is used to present this data to the user, you might consider using StringComparison.InvariantCulture, which operates linguistically for better user output but is unaffected by changes in culture. The following example modifies the two previous examples to use the invariant culture for sorting and searching the array.

```csharp
// Correct.
string []storedNames;

public void StoreNames(string [] names)
{
   int index = 0;
   storedNames = new string[names.Length];

   foreach (string name in names)
   {
      this.storedNames[index++] = name;
   }

   Array.Sort(names, StringComparer.InvariantCulture);  // Line A.
}

public bool DoesNameExist(string name)
{
   return (Array.BinarySearch(this.storedNames, name, StringComparer.InvariantCulture) >= 0);  // Line B.
}
```

```vb
' Correct.
Dim storedNames() As String

Public Sub StoreNames(names() As String)
   Dim index As Integer = 0
   ReDim storedNames(names.Length - 1)

   For Each name As String In names
      Me.storedNames(index) = name
      index+= 1
   Next

   Array.Sort(names, StringComparer.InvariantCulture)           ' Line A.
End Sub

Public Function DoesNameExist(name As String) As Boolean
   Return Array.BinarySearch(Me.storedNames, name, StringComparer.InvariantCulture) >= 0      ' Line B.
End Function
```

Collections Example: Hashtable Constructor

Hashing strings provides a second example of an operation that is affected by the way in which strings are compared.

The following example instantiates a Hashtable object by passing it the StringComparer object that is returned by the StringComparer.OrdinalIgnoreCase property. Because a class StringComparer that is derived from StringComparer implements the IEqualityComparer interface, its GetHashCode method is used to compute the hash code of strings in the hash table.

```
const int initialTableCapacity = 100;
Hashtable h;

public void PopulateFileTable(string directory)
{
   h = new Hashtable(initialTableCapacity,
                     StringComparer.OrdinalIgnoreCase);

   foreach (string file in Directory.GetFiles(directory))
         h.Add(file, File.GetCreationTime(file));
}

public void PrintCreationTime(string targetFile)
{
   Object dt = h[targetFile];
   if (dt != null)
   {
      Console.WriteLine("File {0} was created at time {1}.",
         targetFile,
         (DateTime) dt);
   }
   else
   {
      Console.WriteLine("File {0} does not exist.", targetFile);
   }
}
```

```
Const initialTableCapacity As Integer = 100
Dim h As Hashtable

Public Sub PopulateFileTable(dir As String)
   h = New Hashtable(initialTableCapacity, _
                     StringComparer.OrdinalIgnoreCase)

   For Each filename As String In Directory.GetFiles(dir)
      h.Add(filename, File.GetCreationTime(filename))
   Next
End Sub

Public Sub PrintCreationTime(targetFile As String)
   Dim dt As Object = h(targetFile)
   If dt IsNot Nothing Then
      Console.WriteLine("File {0} was created at {1}.", _
         targetFile, _
         CDate(dt))
   Else
      Console.WriteLine("File {0} does not exist.", targetFile)
   End If
End Sub
```

Back to top

Displaying and Persisting Formatted Data

When you display non-string data such as numbers and dates and times to users, format them by using the user's cultural settings. By default, the String.Format method and the `ToString` methods of the numeric types and the date and time types use the current thread culture for formatting operations. To explicitly specify that the formatting method should use the current culture, you can call an overload of a formatting method that has a `provider` parameter, such as String.Format(IFormatProvider, String, Object[]) or DateTime.ToString(IFormatProvider), and pass it the CultureInfo.CurrentCulture property.

You can persist non-string data either as binary data or as formatted data. If you choose to save it as formatted data, you should call a formatting method overload that includes a `provider` parameter and pass it the CultureInfo.InvariantCulture property. The invariant culture provides a consistent format for formatted data that is independent of culture and machine. In contrast, persisting data that is formatted by using cultures other than the invariant culture has a number of limitations:

- The data is likely to be unusable if it is retrieved on a system that has a different culture, or if the user of the current system changes the current culture and tries to retrieve the data.
- The properties of a culture on a specific computer can differ from standard values. At any time, a user can customize culture-sensitive display settings. Because of this, formatted data that is saved on a system may not be readable after the user customizes cultural settings. The portability of formatted data across computers is likely to be even more limited.
- International, regional, or national standards that govern the formatting of numbers or dates and times change over time, and these changes are incorporated into Windows operating system updates. When formatting conventions change, data that was formatted by using the previous conventions may become unreadable.

The following example illustrates the limited portability that results from using culture-sensitive formatting to persist data. The example saves an array of date and time values to a file. These are formatted by using the conventions of the English (United States) culture. After the application changes the current thread culture to French (Switzerland), it tries to read the saved values by using the formatting conventions of the current culture. The attempt to read two of the data items throws a FormatException exception, and the array of dates now contains two incorrect elements that are equal to MinValue.

```
using System;
using System.Globalization;
using System.IO;
using System.Text;
using System.Threading;

public class Example
{
   private static string filename = @".\dates.dat";

   public static void Main()
   {
      DateTime[] dates = { new DateTime(1758, 5, 6, 21, 26, 0),
                           new DateTime(1818, 5, 5, 7, 19, 0),
                           new DateTime(1870, 4, 22, 23, 54, 0),
                           new DateTime(1890, 9, 8, 6, 47, 0),
                           new DateTime(1905, 2, 18, 15, 12, 0) };
      // Write the data to a file using the current culture.
      WriteData(dates);
      // Change the current culture.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-CH");
      // Read the data using the current culture.
      DateTime[] newDates = ReadData();
      foreach (var newDate in newDates)
         Console.WriteLine(newDate.ToString("g"));
   }

   private static void WriteData(DateTime[] dates)
   {
      StreamWriter sw = new StreamWriter(filename, false, Encoding.UTF8);
      for (int ctr = 0; ctr < dates.Length; ctr++) {
         sw.Write("{0}", dates[ctr].ToString("g", CultureInfo.CurrentCulture));
         if (ctr < dates.Length - 1) sw.Write("|");
      }
      sw.Close();
   }

   private static DateTime[] ReadData()
```

```
   {
      bool exceptionOccurred = false;

      // Read file contents as a single string, then split it.
      StreamReader sr = new StreamReader(filename, Encoding.UTF8);
      string output = sr.ReadToEnd();
      sr.Close();

      string[] values = output.Split( new char[] { '|' } );
      DateTime[] newDates = new DateTime[values.Length];
      for (int ctr = 0; ctr < values.Length; ctr++) {
         try {
            newDates[ctr] = DateTime.Parse(values[ctr], CultureInfo.CurrentCulture);
         }
         catch (FormatException) {
            Console.WriteLine("Failed to parse {0}", values[ctr]);
            exceptionOccurred = true;
         }
      }
      if (exceptionOccurred) Console.WriteLine();
      return newDates;
   }
}
// The example displays the following output:
//       Failed to parse 4/22/1870 11:54 PM
//       Failed to parse 2/18/1905 3:12 PM
//
//       05.06.1758 21:26
//       05.05.1818 07:19
//       01.01.0001 00:00
//       09.08.1890 06:47
//       01.01.0001 00:00
//       01.01.0001 00:00
```

```
Imports System.Globalization
Imports System.IO
Imports System.Text
Imports System.Threading

Module Example
   Private filename As String = ".\dates.dat"

   Public Sub Main()
      Dim dates() As Date = { #5/6/1758 9:26PM#, #5/5/1818 7:19AM#, _
                              #4/22/1870 11:54PM#, #9/8/1890 6:47AM#, _
                              #2/18/1905 3:12PM# }
      ' Write the data to a file using the current culture.
      WriteData(dates)
      ' Change the current culture.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-CH")
      ' Read the data using the current culture.
      Dim newDates() As Date = ReadData()
      For Each newDate In newDates
         Console.WriteLine(newDate.ToString("g"))
      Next
   End Sub

   Private Sub WriteData(dates() As Date)
      Dim sw As New StreamWriter(filename, False, Encoding.Utf8)
      For ctr As Integer = 0 To dates.Length - 1
         sw.Write("{0}", dates(ctr).ToString("g", CultureInfo.CurrentCulture))
         If ctr < dates.Length - 1 Then sw.Write("|")
      Next
      sw.Close()
   End Sub

   Private Function ReadData() As Date()
      Dim exceptionOccurred As Boolean = False

      ' Read file contents as a single string, then split it.
      Dim sr As New StreamReader(filename, Encoding.Utf8)
      Dim output As String = sr.ReadToEnd()
      sr.Close()

      Dim values() As String = output.Split( {"|"c } )
      Dim newDates(values.Length - 1) As Date
      For ctr As Integer = 0 To values.Length - 1
         Try
            newDates(ctr) = DateTime.Parse(values(ctr), CultureInfo.CurrentCulture)
         Catch e As FormatException
            Console.WriteLine("Failed to parse {0}", values(ctr))
            exceptionOccurred = True
         End Try
      Next
      If exceptionOccurred Then Console.WriteLine()
      Return newDates
   End Function
End Module
' The example displays the following output:
'       Failed to parse 4/22/1870 11:54 PM
'       Failed to parse 2/18/1905 3:12 PM
'
'       05.06.1758 21:26
'       05.05.1818 07:19
'       01.01.0001 00:00
'       09.08.1890 06:47
'       01.01.0001 00:00
'       01.01.0001 00:00
'
```

However, if you replace the CultureInfo.CurrentCulture property with CultureInfo.InvariantCulture in the calls to DateTime.ToString(String, IFormatProvider) and DateTime.Parse(String, IFormatProvider), the persisted date and time data is successfully restored, as the following output shows.

```
06.05.1758 21:26
05.05.1818 07:19
22.04.1870 23:54
08.09.1890 06:47
18.02.1905 15:12
```

See also

- Manipulating Strings

Basic String Operations in .NET

5/2/2018 • 2 minutes to read • Edit Online

Applications often respond to users by constructing messages based on user input. For example, it is not uncommon for Web sites to respond to a newly logged-on user with a specialized greeting that includes the user's name. Several methods in the System.String and System.Text.StringBuilder classes allow you to dynamically construct custom strings to display in your user interface. These methods also help you perform a number of basic string operations like creating new strings from arrays of bytes, comparing the values of strings, and modifying existing strings.

In This Section

Creating New Strings
Describes basic ways to convert objects into strings and to combine strings.

Trimming and Removing Characters
Describes how to trim or remove characters in a string.

Padding Strings
Describes how to insert characters or empty spaces into a string.

Comparing Strings
Describes how to compare the contents of two or more strings.

Changing Case
Describes how to change the case of characters within a string.

Using the StringBuilder Class
Describes how to create and modify dynamic string objects with the StringBuilder class.

How to: Perform Basic String Manipulations
Demonstrates the use of basic string operations.

Related Sections

Type Conversion in .NET
Describes how to convert one type into another type.

Formatting Types
Describes how to format strings using format specifiers.

Creating New Strings in .NET

9/6/2018 • 4 minutes to read • Edit Online

The .NET Framework allows strings to be created using simple assignment, and also overloads a class constructor to support string creation using a number of different parameters. The .NET Framework also provides several methods in the System.String class that create new string objects by combining several strings, arrays of strings, or objects.

Creating Strings Using Assignment

The easiest way to create a new String object is simply to assign a string literal to a String object.

Creating Strings Using a Class Constructor

You can use overloads of the String class constructor to create strings from character arrays. You can also create a new string by duplicating a particular character a specified number of times.

Methods that Return Strings

The following table lists several useful methods that return new string objects.

METHOD NAME	USE
String.Format	Builds a formatted string from a set of input objects.
String.Concat	Builds strings from two or more strings.
String.Join	Builds a new string by combining an array of strings.
String.Insert	Builds a new string by inserting a string into the specified index of an existing string.
String.CopyTo	Copies specified characters in a string into a specified position in an array of characters.

Format

You can use the **String.Format** method to create formatted strings and concatenate strings representing multiple objects. This method automatically converts any passed object into a string. For example, if your application must display an **Int32** value and a **DateTime** value to the user, you can easily construct a string to represent these values using the **Format** method. For information about formatting conventions used with this method, see the section on composite formatting.

The following example uses the **Format** method to create a string that uses an integer variable.

```
int numberOfFleas = 12;
string miscInfo = String.Format("Your dog has {0} fleas. " +
                                "It is time to get a flea collar. " +
                                "The current universal date is: {1:u}.",
                                numberOfFleas, DateTime.Now);
Console.WriteLine(miscInfo);
// The example displays the following output:
//       Your dog has 12 fleas. It is time to get a flea collar.
//       The current universal date is: 2008-03-28 13:31:40Z.
```

```
Dim numberOfFleas As Integer = 12
Dim miscInfo As String = String.Format("Your dog has {0} fleas. " & _
                                       "It is time to get a flea collar. " & _
                                       "The current universal date is: {1:u}.", _
                                       numberOfFleas, Date.Now)
Console.WriteLine(miscInfo)
' The example displays the following output:
'       Your dog has 12 fleas. It is time to get a flea collar.
'       The current universal date is: 2008-03-28 13:31:40Z.
```

In this example,DateTime.Now displays the current date and time in a manner specified by the culture associated with the current thread.

Concat

The **String.Concat** method can be used to easily create a new string object from two or more existing objects. It provides a language-independent way to concatenate strings. This method accepts any class that derives from **System.Object**. The following example creates a string from two existing string objects and a separating character.

```
string helloString1 = "Hello";
string helloString2 = "World!";
Console.WriteLine(String.Concat(helloString1, ' ', helloString2));
// The example displays the following output:
//      Hello World!
```

```
Dim helloString1 As String = "Hello"
Dim helloString2 As String = "World!"
Console.WriteLine(String.Concat(helloString1, " "c, helloString2))
' The example displays the following output:
'      Hello World!
```

Join

The **String.Join** method creates a new string from an array of strings and a separator string. This method is useful if you want to concatenate multiple strings together, making a list perhaps separated by a comma.

The following example uses a space to bind a string array.

```
string[] words = {"Hello", "and", "welcome", "to", "my" , "world!"};
Console.WriteLine(String.Join(" ", words));
// The example displays the following output:
//      Hello and welcome to my world!
```

```
Dim words() As String = {"Hello", "and", "welcome", "to", "my" , "world!"}
Console.WriteLine(String.Join(" ", words))
' The example displays the following output:
'      Hello and welcome to my world!
```

Insert

The **String.Insert** method creates a new string by inserting a string into a specified position in another string. This method uses a zero-based index. The following example inserts a string into the fifth index position of `MyString` and creates a new string with this value.

```
string sentence = "Once a time.";
 Console.WriteLine(sentence.Insert(4, " upon"));
 // The example displays the following output:
 //      Once upon a time.
```

```
Dim sentence As String = "Once a time."
 Console.WriteLine(sentence.Insert(4, " upon"))
 ' The example displays the following output:
 '      Once upon a time.
```

CopyTo

The **String.CopyTo** method copies portions of a string into an array of characters. You can specify both the beginning index of the string and the number of characters to be copied. This method takes the source index, an array of characters, the destination index, and the number of characters to copy. All indexes are zero-based.

The following example uses the **CopyTo** method to copy the characters of the word "Hello" from a string object to the first index position of an array of characters.

```
string greeting = "Hello World!";
char[] charArray = {'W','h','e','r','e'};
Console.WriteLine("The original character array: {0}", new string(charArray));
greeting.CopyTo(0, charArray,0 ,5);
Console.WriteLine("The new character array: {0}", new string(charArray));
// The example displays the following output:
//       The original character array: Where
//       The new character array: Hello
```

```
Dim greeting As String = "Hello World!"
Dim charArray() As Char = {"W"c, "h"c, "e"c, "r"c, "e"c}
Console.WriteLine("The original character array: {0}", New String(charArray))
greeting.CopyTo(0, charArray,0 ,5)
Console.WriteLine("The new character array: {0}", New String(charArray))
' The example displays the following output:
'       The original character array: Where
'       The new character array: Hello
```

See also

- Basic String Operations
- Composite Formatting

Trimming and Removing Characters from Strings in .NET

9/6/2018 • 5 minutes to read • Edit Online

If you are parsing a sentence into individual words, you might end up with words that have blank spaces (also called white spaces) on either end of the word. In this situation, you can use one of the trim methods in the **System.String** class to remove any number of spaces or other characters from a specified position in the string. The following table describes the available trim methods.

METHOD NAME	USE
String.Trim	Removes white spaces or characters specified in an array of characters from the beginning and end of a string.
String.TrimEnd	Removes characters specified in an array of characters from the end of a string.
String.TrimStart	Removes characters specified in an array of characters from the beginning of a string.
String.Remove	Removes a specified number of characters from a specified index position in a string.

Trim

You can easily remove white spaces from both ends of a string by using the String.Trim method, as shown in the following example.

```
String^ MyString = " Big   ";
Console::WriteLine("Hello{0}World!", MyString);
String^ TrimString = MyString->Trim();
Console::WriteLine("Hello{0}World!", TrimString);
// The example displays the following output:
//       Hello Big   World!
//       HelloBigWorld!
```

```
string MyString = " Big   ";
Console.WriteLine("Hello{0}World!", MyString);
string TrimString = MyString.Trim();
Console.WriteLine("Hello{0}World!", TrimString);
//       The example displays the following output:
//             Hello Big   World!
//             HelloBigWorld!
```

```
Dim MyString As String = " Big   "
Console.WriteLine("Hello{0}World!", MyString)
Dim TrimString As String = MyString.Trim()
Console.WriteLine("Hello{0}World!", TrimString)
' The example displays the following output:
'       Hello Big   World!
'       HelloBigWorld!
```

You can also remove characters that you specify in a character array from the beginning and end of a string. The following example removes white-space characters, periods, and asterisks.

```
using System;

public class Example
{
   public static void Main()
   {
      String header = "* A Short String. *";
      Console.WriteLine(header);
      Console.WriteLine(header.Trim( new Char[] { ' ', '*', '.' } ));
   }
}
// The example displays the following output:
//       * A Short String. *
//       A Short String
```

```
Module Example
   Public Sub Main()
      Dim header As String = "* A Short String. *"
      Console.WriteLine(header)
      Console.WriteLine(header.Trim( { " "c, "*"c, "."c } ))
   End Sub
End Module
' The example displays the following output:
'       * A Short String. *
'       A Short String
```

TrimEnd

The **String.TrimEnd** method removes characters from the end of a string, creating a new string object. An array of characters is passed to this method to specify the characters to be removed. The order of the elements in the character array does not affect the trim operation. The trim stops when a character not specified in the array is found.

The following example removes the last letters of a string using the **TrimEnd** method. In this example, the position of the `'r'` character and the `'W'` character are reversed to illustrate that the order of characters in the array does not matter. Notice that this code removes the last word of `MyString` plus part of the first.

```
String^ MyString = "Hello World!";
array<Char>^ MyChar = {'r','o','W','l','d','!',' '};
String^ NewString = MyString->TrimEnd(MyChar);
Console::WriteLine(NewString);
```

```
string MyString = "Hello World!";
char[] MyChar = {'r','o','W','l','d','!',' '};
string NewString = MyString.TrimEnd(MyChar);
Console.WriteLine(NewString);
```

```
Dim MyString As String = "Hello World!"
Dim MyChar() As Char = {"r","o","W","l","d","!"," "}
Dim NewString As String = MyString.TrimEnd(MyChar)
Console.WriteLine(NewString)
```

This code displays `He` to the console.

The following example removes the last word of a string using the **TrimEnd** method. In this code, a comma follows the word `Hello` and, because the comma is not specified in the array of characters to trim, the trim ends at the comma.

```
String^ MyString = "Hello, World!";
array<Char>^ MyChar = {'r','o','W','l','d','!',' '};
String^ NewString = MyString->TrimEnd(MyChar);
Console::WriteLine(NewString);
```

```
string MyString = "Hello, World!";
char[] MyChar = {'r','o','W','l','d','!',' '};
string NewString = MyString.TrimEnd(MyChar);
Console.WriteLine(NewString);
```

```
Dim MyString As String = "Hello, World!"
Dim MyChar() As Char = {"r","o","W","l","d","!"," "}
Dim NewString As String = MyString.TrimEnd(MyChar)
Console.WriteLine(NewString)
```

This code displays `Hello,` to the console.

TrimStart

The **String.TrimStart** method is similar to the **String.TrimEnd** method except that it creates a new string by removing characters from the beginning of an existing string object. An array of characters is passed to the **TrimStart** method to specify the characters to be removed. As with the **TrimEnd** method, the order of the elements in the character array does not affect the trim operation. The trim stops when a character not specified in the array is found.

The following example removes the first word of a string. In this example, the position of the `'l'` character and the `'H'` character are reversed to illustrate that the order of characters in the array does not matter.

```
String^ MyString = "Hello World!";
array<Char>^ MyChar = {'e', 'H','l','o',' ' };
String^ NewString = MyString->TrimStart(MyChar);
Console::WriteLine(NewString);
```

```
string MyString = "Hello World!";
char[] MyChar = {'e', 'H','l','o',' ' };
string NewString = MyString.TrimStart(MyChar);
Console.WriteLine(NewString);
```

```
Dim MyString As String = "Hello World!"
Dim MyChar() As Char = {"e","H","l","o"," " }
Dim NewString As String = MyString.TrimStart(MyChar)
Console.WriteLine(NewString)
```

This code displays `World!` to the console.

Remove

The String.Remove method removes a specified number of characters that begin at a specified position in an existing string. This method assumes a zero-based index.

The following example removes ten characters from a string beginning at position five of a zero-based index of the string.

```
String^ MyString = "Hello Beautiful World!";
Console::WriteLine(MyString->Remove(5,10));
// The example displays the following output:
//         Hello World!
```

```
string MyString = "Hello Beautiful World!";
Console.WriteLine(MyString.Remove(5,10));
// The example displays the following output:
//         Hello World!
```

```
Dim MyString As String = "Hello Beautiful World!"
Console.WriteLine(MyString.Remove(5,10))
' The example displays the following output:
'         Hello World!
```

You can also remove a specified character or substring from a string by calling the String.Replace(String, String) method and specifying an empty string (String.Empty) as the replacement. The following example removes all commas from a string.

```
using System;

public class Example
{
   public static void Main()
   {
      String phrase = "a cold, dark night";
      Console.WriteLine("Before: {0}", phrase);
      phrase = phrase.Replace(",", "");
      Console.WriteLine("After: {0}", phrase);
   }
}
// The example displays the following output:
//       Before: a cold, dark night
//       After: a cold dark night
```

```
Module Example
   Public Sub Main()
      Dim phrase As String = "a cold, dark night"
      Console.WriteLine("Before: {0}", phrase)
      phrase = phrase.Replace(",", "")
      Console.WriteLine("After: {0}", phrase)
   End Sub
End Module
' The example displays the following output:
'       Before: a cold, dark night
'       After: a cold dark night
```

See also

- Basic String Operations

Padding Strings in .NET

9/6/2018 • 2 minutes to read • Edit Online

Use one of the following String methods to create a new string that consists of an original string that is padded with leading or trailing characters to a specified total length. The padding character can be a space or a specified character. The resulting string appears to be either right-aligned or left-aligned. If the original string's length is already equal to or greater than the desired total length, the padding methods return the original string unchanged; for more information, see the **Returns** sections of the two overloads of the String.PadLeft and String.PadRight methods.

METHOD NAME	USE
String.PadLeft	Pads a string with leading characters to a specified total length.
String.PadRight	Pads a string with trailing characters to a specified total length.

PadLeft

The String.PadLeft method creates a new string by concatenating enough leading pad characters to an original string to achieve a specified total length. The String.PadLeft(Int32) method uses white space as the padding character and the String.PadLeft(Int32, Char) method enables you to specify your own padding character.

The following code example uses the PadLeft method to create a new string that is twenty characters long. The example displays "`--------Hello World!`" to the console.

```
String^ MyString = "Hello World!";
Console::WriteLine(MyString->PadLeft(20, '-'));
```

```
string MyString = "Hello World!";
Console.WriteLine(MyString.PadLeft(20, '-'));
```

```
Dim MyString As String = "Hello World!"
Console.WriteLine(MyString.PadLeft(20, "-"c))
```

PadRight

The String.PadRight method creates a new string by concatenating enough trailing pad characters to an original string to achieve a specified total length. The String.PadRight(Int32) method uses white space as the padding character and the String.PadRight(Int32, Char) method enables you to specify your own padding character.

The following code example uses the PadRight method to create a new string that is twenty characters long. The example displays "`Hello World!--------`" to the console.

```
String^ MyString = "Hello World!";
Console::WriteLine(MyString->PadRight(20, '-'));
```

```csharp
string MyString = "Hello World!";
Console.WriteLine(MyString.PadRight(20, '-'));
```

```vb
Dim MyString As String = "Hello World!"
Console.WriteLine(MyString.PadRight(20, "-"c))
```

See also

- Basic String Operations

Comparing Strings in .NET

9/14/2018 • 6 minutes to read • Edit Online

.NET provides several methods to compare the values of strings. The following table lists and describes the value-comparison methods.

METHOD NAME	USE
String.Compare	Compares the values of two strings. Returns an integer value.
String.CompareOrdinal	Compares two strings without regard to local culture. Returns an integer value.
String.CompareTo	Compares the current string object to another string. Returns an integer value.
String.StartsWith	Determines whether a string begins with the string passed. Returns a Boolean value.
String.EndsWith	Determines whether a string ends with the string passed. Returns a Boolean value.
String.Equals	Determines whether two strings are the same. Returns a Boolean value.
String.IndexOf	Returns the index position of a character or string, starting from the beginning of the string you are examining. Returns an integer value.
String.LastIndexOf	Returns the index position of a character or string, starting from the end of the string you are examining. Returns an integer value.

Compare

The static String.Compare method provides a thorough way of comparing two strings. This method is culturally aware. You can use this function to compare two strings or substrings of two strings. Additionally, overloads are provided that regard or disregard case and cultural variance. The following table shows the three integer values that this method might return.

RETURN VALUE	CONDITION
A negative integer	The first string precedes the second string in the sort order. -or- The first string is `null`.

RETURN VALUE	CONDITION
0	The first string and the second string are equal. -or- Both strings are `null`.
A positive integer -or- 1	The first string follows the second string in the sort order. -or- The second string is `null`.

> **IMPORTANT**
>
> The String.Compare method is primarily intended for use when ordering or sorting strings. You should not use the String.Compare method to test for equality (that is, to explicitly look for a return value of 0 with no regard for whether one string is less than or greater than the other). Instead, to determine whether two strings are equal, use the String.Equals(String, String, StringComparison) method.

The following example uses the String.Compare method to determine the relative values of two strings.

```
String^ string1 = "Hello World!";
Console::WriteLine(String::Compare(string1, "Hello World?"));
```

```
string string1 = "Hello World!";
Console.WriteLine(String.Compare(string1, "Hello World?"));
```

```
Dim string1 As String = "Hello World!"
Console.WriteLine(String.Compare(string1, "Hello World?"))
```

This example displays `-1` to the console.

The preceding example is culture-sensitive by default. To perform a culture-insensitive string comparison, use an overload of the String.Compare method that allows you to specify the culture to use by supplying a *culture* parameter. For an example that demonstrates how to use the String.Compare method to perform a culture-insensitive comparison, see Performing Culture-Insensitive String Comparisons.

CompareOrdinal

The String.CompareOrdinal method compares two string objects without considering the local culture. The return values of this method are identical to the values returned by the **Compare** method in the previous table.

> **IMPORTANT**
>
> The String.CompareOrdinal method is primarily intended for use when ordering or sorting strings. You should not use the String.CompareOrdinal method to test for equality (that is, to explicitly look for a return value of 0 with no regard for whether one string is less than or greater than the other). Instead, to determine whether two strings are equal, use the String.Equals(String, String, StringComparison) method.

The following example uses the **CompareOrdinal** method to compare the values of two strings.

```
String^ string1 = "Hello World!";
Console::WriteLine(String::CompareOrdinal(string1, "hello world!"));
```

```
string string1 = "Hello World!";
Console.WriteLine(String.CompareOrdinal(string1, "hello world!"));
```

```
Dim string1 As String = "Hello World!"
Console.WriteLine(String.CompareOrdinal(string1, "hello world!"))
```

This example displays `-32` to the console.

CompareTo

The String.CompareTo method compares the string that the current string object encapsulates to another string or object. The return values of this method are identical to the values returned by the String.Compare method in the previous table.

> **IMPORTANT**
>
> The String.CompareTo method is primarily intended for use when ordering or sorting strings. You should not use the String.CompareTo method to test for equality (that is, to explicitly look for a return value of 0 with no regard for whether one string is less than or greater than the other). Instead, to determine whether two strings are equal, use the String.Equals(String, String, StringComparison) method.

The following example uses the String.CompareTo method to compare the `string1` object to the `string2` object.

```
String^ string1 = "Hello World";
String^ string2 = "Hello World!";
int MyInt = string1->CompareTo(string2);
Console::WriteLine( MyInt );
```

```
string string1 = "Hello World";
string string2 = "Hello World!";
int MyInt = string1.CompareTo(string2);
Console.WriteLine( MyInt );
```

```
Dim string1 As String = "Hello World"
Dim string2 As String = "Hello World!"
Dim MyInt As Integer = string1.CompareTo(string2)
Console.WriteLine(MyInt)
```

This example displays `-1` to the console.

All overloads of the String.CompareTo method perform culture-sensitive and case-sensitive comparisons by default. No overloads of this method are provided that allow you to perform a culture-insensitive comparison. For code clarity, we recommend that you use the **String.Compare** method instead, specifying CultureInfo.CurrentCulture for culture-sensitive operations or CultureInfo.InvariantCulture for culture-insensitive operations. For examples that demonstrate how to use the **String.Compare** method to perform both culture-sensitive and culture-insensitive comparisons, see Performing Culture-Insensitive String Comparisons.

Equals

The **String.Equals** method can easily determine if two strings are the same. This case-sensitive method returns a **true** or **false** Boolean value. It can be used from an existing class, as illustrated in the next example. The following example uses the **Equals** method to determine whether a string object contains the phrase "Hello World".

```
String^ string1 = "Hello World";
Console::WriteLine(string1->Equals("Hello World"));
```

```
string string1 = "Hello World";
Console.WriteLine(string1.Equals("Hello World"));
```

```
Dim string1 As String = "Hello World"
Console.WriteLine(string1.Equals("Hello World"))
```

This example displays `True` to the console.

This method can also be used as a static method. The following example compares two string objects using a static method.

```
String^ string1 = "Hello World";
String^ string2 = "Hello World";
Console::WriteLine(String::Equals(string1, string2));
```

```
string string1 = "Hello World";
string string2 = "Hello World";
Console.WriteLine(String.Equals(string1, string2));
```

```
Dim string1 As String = "Hello World"
Dim string2 As String = "Hello World"
Console.WriteLine(String.Equals(string1, string2))
```

This example displays `True` to the console.

StartsWith and EndsWith

You can use the **String.StartsWith** method to determine whether a string object begins with the same characters that encompass another string. This case-sensitive method returns **true** if the current string object begins with the passed string and **false** if it does not. The following example uses this method to determine if a string object begins with "Hello".

```
String^ string1 = "Hello World";
Console::WriteLine(string1->StartsWith("Hello"));
```

```
string string1 = "Hello World";
Console.WriteLine(string1.StartsWith("Hello"));
```

```
Dim string1 As String = "Hello World!"
Console.WriteLine(string1.StartsWith("Hello"))
```

This example displays `True` to the console.

The **String.EndsWith** method compares a passed string to the characters that exist at the end of the current string object. It also returns a Boolean value. The following example checks the end of a string using the **EndsWith** method.

```
String^ string1 = "Hello World";
Console::WriteLine(string1->EndsWith("Hello"));
```

```
string string1 = "Hello World";
Console.WriteLine(string1.EndsWith("Hello"));
```

```
Dim string1 As String = "Hello World!"
Console.WriteLine(string1.EndsWith("Hello"))
```

This example displays `False` to the console.

IndexOf and LastIndexOf

You can use the **String.IndexOf** method to determine the position of the first occurrence of a particular character within a string. This case-sensitive method starts counting from the beginning of a string and returns the position of a passed character using a zero-based index. If the character cannot be found, a value of –1 is returned.

The following example uses the **IndexOf** method to search for the first occurrence of the '`l`' character in a string.

```
String^ string1 = "Hello World";
Console::WriteLine(string1->IndexOf('l'));
```

```
string string1 = "Hello World";
Console.WriteLine(string1.IndexOf('l'));
```

```
Dim string1 As String = "Hello World!"
Console.WriteLine(string1.IndexOf("l"))
```

This example displays `2` to the console.

The **String.LastIndexOf** method is similar to the **String.IndexOf** method except that it returns the position of the last occurrence of a particular character within a string. It is case-sensitive and uses a zero-based index.

The following example uses the **LastIndexOf** method to search for the last occurrence of the '`l`' character in a string.

```
String^ string1 = "Hello World";
Console::WriteLine(string1->LastIndexOf('l'));
```

```
string string1 = "Hello World";
Console.WriteLine(string1.LastIndexOf('l'));
```

```
Dim string1 As String = "Hello World!"
Console.WriteLine(string1.LastIndexOf("l"))
```

This example displays `9` to the console.

Both methods are useful when used in conjunction with the **String.Remove** method. You can use either the **IndexOf** or **LastIndexOf** methods to retrieve the position of a character, and then supply that position to the **Remove** method in order to remove a character or a word that begins with that character.

See also

- Basic String Operations
- Performing Culture-Insensitive String Operations
- Sorting Weight Tables (for .NET on Windows)
- Default Unicode Collation Element Table (for .NET Core on Linux and macOS)

Changing Case in .NET

9/6/2018 • 4 minutes to read • Edit Online

If you write an application that accepts input from a user, you can never be sure what case he or she will use to enter the data. Often, you want strings to be cased consistently, particularly if you are displaying them in the user interface. The following table describes three case-changing methods. The first two methods provide an overload that accepts a culture.

METHOD NAME	USE
String.ToUpper	Converts all characters in a string to uppercase.
String.ToLower	Converts all characters in a string to lowercase.
TextInfo.ToTitleCase	Converts a string to title case.

> **WARNING**
> Note that the String.ToUpper and String.ToLower methods should not be used to convert strings in order to compare them or test them for equality. For more information, see the Comparing strings of mixed case section.

Comparing strings of mixed case

To compare strings of mixed case to determine their ordering, call one of the overloads of the String.CompareTo method with a `comparisonType` parameter, and provide a value of either StringComparison.CurrentCultureIgnoreCase, StringComparison.InvariantCultureIgnoreCase, or StringComparison.OrdinalIgnoreCase for the `comparisonType` argument. For a comparison using a specific culture other than the current culture, call an overload of the String.CompareTo method with both a `culture` and `options` parameter, and provide a value of CompareOptions.IgnoreCase as the `options` argument.

To compare strings of mixed case to determine whether they are equal, their, call one of the overloads of the String.Equals method with a `comparisonType` parameter, and provide a value of either StringComparison.CurrentCultureIgnoreCase, StringComparison.InvariantCultureIgnoreCase, or StringComparison.OrdinalIgnoreCase for the `comparisonType` argument.

For more information, see Best Practices for Using Strings.

ToUpper

The String.ToUpper method changes all characters in a string to uppercase. The following example converts the string "Hello World!" from mixed case to uppercase.

```
string properString = "Hello World!";
Console.WriteLine(properString.ToUpper());
// This example displays the following output:
//       HELLO WORLD!
```

```
Dim MyString As String = "Hello World!"
Console.WriteLine(MyString.ToUpper())
' This example displays the following output:
'       HELLO WORLD!
```

The preceding example is culture-sensitive by default; it applies the casing conventions of the current culture. To perform a culture-insensitive case change or to apply the casing conventions of a particular culture, use the String.ToUpper(CultureInfo) method overload and supply a value of CultureInfo.InvariantCulture or a System.Globalization.CultureInfo object that represents the specified culture to the *culture* parameter. For an example that demonstrates how to use the ToUpper method to perform a culture-insensitive case change, see Performing Culture-Insensitive Case Changes.

ToLower

The String.ToLower method is similar to the previous method, but instead converts all the characters in a string to lowercase. The following example converts the string "Hello World!" to lowercase.

```
string properString = "Hello World!";
Console.WriteLine(properString.ToLower());
// This example displays the following output:
//       hello world!
```

```
Dim MyString As String = "Hello World!"
Console.WriteLine(MyString.ToLower())
' This example displays the following output:
'       hello world!
```

The preceding example is culture-sensitive by default; it applies the casing conventions of the current culture. To perform a culture-insensitive case change or to apply the casing conventions of a particular culture, use the String.ToLower(CultureInfo) method overload and supply a value of CultureInfo.InvariantCulture or a System.Globalization.CultureInfo object that represents the specified culture to the *culture* parameter. For an example that demonstrates how to use the ToLower(CultureInfo) method to perform a culture-insensitive case change, see Performing Culture-Insensitive Case Changes.

ToTitleCase

The TextInfo.ToTitleCase converts the first character of each word to uppercase and the remaining characters to lowercase. However, words that are entirely uppercase are assumed to be acronyms and are not converted.

The TextInfo.ToTitleCase method is culture-sensitive; that is, it uses the casing conventions of a particular culture. In order to call the method, you first retrieve the TextInfo object that represents the casing conventions of the particular culture from the CultureInfo.TextInfo property of a particular culture.

The following example passes each string in an array to the TextInfo.ToTitleCase method. The strings include proper title strings as well as acronyms. The strings are converted to title case by using the casing conventions of the English (United States) culture.

```csharp
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string[] values = { "a tale of two cities", "gROWL to the rescue",
                          "inside the US government", "sports and MLB baseball",
                          "The Return of Sherlock Holmes", "UNICEF and children"};

      TextInfo ti = CultureInfo.CurrentCulture.TextInfo;
      foreach (var value in values)
         Console.WriteLine("{0} --> {1}", value, ti.ToTitleCase(value));
   }
}
// The example displays the following output:
//    a tale of two cities --> A Tale Of Two Cities
//    gROWL to the rescue --> Growl To The Rescue
//    inside the US government --> Inside The US Government
//    sports and MLB baseball --> Sports And MLB Baseball
//    The Return of Sherlock Holmes --> The Return Of Sherlock Holmes
//    UNICEF and children --> UNICEF And Children
```

```vb
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim values() As String = { "a tale of two cities", "gROWL to the rescue",
                                 "inside the US government", "sports and MLB baseball",
                                 "The Return of Sherlock Holmes", "UNICEF and children"}

      Dim ti As TextInfo = CultureInfo.CurrentCulture.TextInfo
      For Each value In values
         Console.WriteLine("{0} --> {1}", value, ti.ToTitleCase(value))
      Next
   End Sub
End Module
' The example displays the following output:
'    a tale of two cities --> A Tale Of Two Cities
'    gROWL to the rescue --> Growl To The Rescue
'    inside the US government --> Inside The US Government
'    sports and MLB baseball --> Sports And MLB Baseball
'    The Return of Sherlock Holmes --> The Return Of Sherlock Holmes
'    UNICEF and children --> UNICEF And Children
```

Note that although it is culture-sensitive, the TextInfo.ToTitleCase method does not provide linguistically correct casing rules. For instance, in the previous example, the method converts "a tale of two cities" to "A Tale Of Two Cities". However, the linguistically correct title casing for the en-US culture is "A Tale of Two Cities."

See also

- Basic String Operations
- Performing Culture-Insensitive String Operations

Using the StringBuilder Class in .NET

9/6/2018 • 7 minutes to read • Edit Online

The String object is immutable. Every time you use one of the methods in the System.String class, you create a new string object in memory, which requires a new allocation of space for that new object. In situations where you need to perform repeated modifications to a string, the overhead associated with creating a new String object can be costly. The System.Text.StringBuilder class can be used when you want to modify a string without creating a new object. For example, using the StringBuilder class can boost performance when concatenating many strings together in a loop.

Importing the System.Text Namespace

The StringBuilder class is found in the System.Text namespace. To avoid having to provide a fully qualified type name in your code, you can import the System.Text namespace:

```
using namespace System;
using namespace System::Text;
```

```
using System;
using System.Text;
```

```
Imports System.Text
```

Instantiating a StringBuilder Object

You can create a new instance of the StringBuilder class by initializing your variable with one of the overloaded constructor methods, as illustrated in the following example.

```
StringBuilder^ myStringBuilder = gcnew StringBuilder("Hello World!");
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
```

```
Dim myStringBuilder As New StringBuilder("Hello World!")
```

Setting the Capacity and Length

Although the StringBuilder is a dynamic object that allows you to expand the number of characters in the string that it encapsulates, you can specify a value for the maximum number of characters that it can hold. This value is called the capacity of the object and should not be confused with the length of the string that the current StringBuilder holds. For example, you might create a new instance of the StringBuilder class with the string "Hello", which has a length of 5, and you might specify that the object has a maximum capacity of 25. When you modify the StringBuilder, it does not reallocate size for itself until the capacity is reached. When this occurs, the new space is allocated automatically and the capacity is doubled. You can specify the capacity of the StringBuilder class using one of the overloaded constructors. The following example specifies that the `myStringBuilder` object can be expanded to a maximum of 25 spaces.

```
StringBuilder^ myStringBuilder = gcnew StringBuilder("Hello World!", 25);
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!", 25);
```

```
Dim myStringBuilder As New StringBuilder("Hello World!", 25)
```

Additionally, you can use the read/write Capacity property to set the maximum length of your object. The following example uses the **Capacity** property to define the maximum object length.

```
myStringBuilder->Capacity = 25;
```

```
myStringBuilder.Capacity = 25;
```

```
myStringBuilder.Capacity = 25
```

The EnsureCapacity method can be used to check the capacity of the current **StringBuilder**. If the capacity is greater than the passed value, no change is made; however, if the capacity is smaller than the passed value, the current capacity is changed to match the passed value.

The Length property can also be viewed or set. If you set the **Length** property to a value that is greater than the **Capacity** property, the **Capacity** property is automatically changed to the same value as the **Length** property. Setting the **Length** property to a value that is less than the length of the string within the current **StringBuilder** shortens the string.

Modifying the StringBuilder String

The following table lists the methods you can use to modify the contents of a **StringBuilder**.

METHOD NAME	USE
StringBuilder.Append	Appends information to the end of the current **StringBuilder**.
StringBuilder.AppendFormat	Replaces a format specifier passed in a string with formatted text.
StringBuilder.Insert	Inserts a string or object into the specified index of the current **StringBuilder**.
StringBuilder.Remove	Removes a specified number of characters from the current **StringBuilder**.
StringBuilder.Replace	Replaces a specified character at a specified index.

Append

The **Append** method can be used to add text or a string representation of an object to the end of a string represented by the current **StringBuilder**. The following example initializes a **StringBuilder** to "Hello World" and then appends some text to the end of the object. Space is allocated automatically as needed.

```
StringBuilder^ myStringBuilder = gcnew StringBuilder("Hello World!");
myStringBuilder->Append(" What a beautiful day.");
Console::WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello World! What a beautiful day.
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Append(" What a beautiful day.");
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello World! What a beautiful day.
```

```
Dim myStringBuilder As New StringBuilder("Hello World!")
myStringBuilder.Append(" What a beautiful day.")
Console.WriteLine(myStringBuilder)
' The example displays the following output:
'       Hello World! What a beautiful day.
```

AppendFormat

The StringBuilder.AppendFormat method adds text to the end of the StringBuilder object. It supports the composite formatting feature (for more information, see Composite Formatting) by calling the IFormattable implementation of the object or objects to be formatted. Therefore, it accepts the standard format strings for numeric, date and time, and enumeration values, the custom format strings for numeric and date and time values, and the format strings defined for custom types. (For information about formatting, see Formatting Types.) You can use this method to customize the format of variables and append those values to a StringBuilder. The following example uses the AppendFormat method to place an integer value formatted as a currency value at the end of a StringBuilder object.

```
int MyInt = 25;
StringBuilder^ myStringBuilder = gcnew StringBuilder("Your total is ");
myStringBuilder->AppendFormat("{0:C} ", MyInt);
Console::WriteLine(myStringBuilder);
// The example displays the following output:
//       Your total is $25.00
```

```
int MyInt = 25;
StringBuilder myStringBuilder = new StringBuilder("Your total is ");
myStringBuilder.AppendFormat("{0:C} ", MyInt);
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Your total is $25.00
```

```
Dim MyInt As Integer = 25
Dim myStringBuilder As New StringBuilder("Your total is ")
myStringBuilder.AppendFormat("{0:C} ", MyInt)
Console.WriteLine(myStringBuilder)
' The example displays the following output:
'     Your total is $25.00
```

Insert

The Insert method adds a string or object to a specified position in the current StringBuilder object. The following example uses this method to insert a word into the sixth position of a StringBuilder object.

```
StringBuilder^ myStringBuilder = gcnew StringBuilder("Hello World!");
myStringBuilder->Insert(6,"Beautiful ");
Console::WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello Beautiful World!
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Insert(6,"Beautiful ");
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello Beautiful World!
```

```
Dim myStringBuilder As New StringBuilder("Hello World!")
myStringBuilder.Insert(6, "Beautiful ")
Console.WriteLine(myStringBuilder)
' The example displays the following output:
'      Hello Beautiful World!
```

Remove

You can use the **Remove** method to remove a specified number of characters from the current StringBuilder object, beginning at a specified zero-based index. The following example uses the **Remove** method to shorten a StringBuilder object.

```
StringBuilder^ myStringBuilder = gcnew StringBuilder("Hello World!");
myStringBuilder->Remove(5,7);
Console::WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Remove(5,7);
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello
```

```
Dim myStringBuilder As New StringBuilder("Hello World!")
myStringBuilder.Remove(5, 7)
Console.WriteLine(myStringBuilder)
' The example displays the following output:
'       Hello
```

Replace

The **Replace** method can be used to replace characters within the StringBuilder object with another specified character. The following example uses the **Replace** method to search a StringBuilder object for all instances of the exclamation point character (!) and replace them with the question mark character (?).

```
StringBuilder^ myStringBuilder = gcnew StringBuilder("Hello World!");
myStringBuilder->Replace('!', '?');
Console::WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello World?
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Replace('!', '?');
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello World?
```

```
Dim myStringBuilder As New StringBuilder("Hello World!")
myStringBuilder.Replace("!"c, "?"c)
Console.WriteLine(myStringBuilder)
' The example displays the following output:
'       Hello World?
```

Converting a StringBuilder Object to a String

You must convert the StringBuilder object to a String object before you can pass the string represented by the StringBuilder object to a method that has a String parameter or display it in the user interface. You do this conversion by calling the StringBuilder.ToString method. The following example calls a number of StringBuilder methods and then calls the StringBuilder.ToString() method to display the string.

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      StringBuilder sb = new StringBuilder();
      bool flag = true;
      string[] spellings = { "recieve", "receeve", "receive" };
      sb.AppendFormat("Which of the following spellings is {0}:", flag);
      sb.AppendLine();
      for (int ctr = 0; ctr <= spellings.GetUpperBound(0); ctr++) {
         sb.AppendFormat("   {0}. {1}", ctr, spellings[ctr]);
         sb.AppendLine();
      }
      sb.AppendLine();
      Console.WriteLine(sb.ToString());
   }
}
// The example displays the following output:
//       Which of the following spellings is True:
//          0. recieve
//          1. receeve
//          2. receive
```

```vb
Imports System.Text

Module Example
   Public Sub Main()
      Dim sb As New StringBuilder()
      Dim flag As Boolean = True
      Dim spellings() As String = { "recieve", "receeve", "receive" }
      sb.AppendFormat("Which of the following spellings is {0}:", flag)
      sb.AppendLine()
      For ctr As Integer = 0 To spellings.GetUpperBound(0)
         sb.AppendFormat("   {0}. {1}", ctr, spellings(ctr))
         sb.AppendLine()
      Next
      sb.AppendLine()
      Console.WriteLine(sb.ToString())
   End Sub
End Module
' The example displays the following output:
'       Which of the following spellings is True:
'          0. recieve
'          1. receeve
'          2. receive
```

See also

- System.Text.StringBuilder
- Basic String Operations
- Formatting Types

How to: Perform Basic String Manipulations in .NET

9/6/2018 • 4 minutes to read • Edit Online

The following example uses some of the methods discussed in the Basic String Operations topics to construct a class that performs string manipulations in a manner that might be found in a real-world application. The `MailToData` class stores the name and address of an individual in separate properties and provides a way to combine the `City`, `State`, and `Zip` fields into a single string for display to the user. Furthermore, the class allows the user to enter the city, state, and ZIP Code information as a single string; the application automatically parses the single string and enters the proper information into the corresponding property.

For simplicity, this example uses a console application with a command-line interface.

Example

```
using System;

class MainClass
{
   static void Main()
   {
      MailToData MyData = new MailToData();

      Console.Write("Enter Your Name: ");
      MyData.Name = Console.ReadLine();
      Console.Write("Enter Your Address: ");
      MyData.Address = Console.ReadLine();
      Console.Write("Enter Your City, State, and ZIP Code separated by spaces: ");
      MyData.CityStateZip = Console.ReadLine();
      Console.WriteLine();

      if (MyData.Validated) {
         Console.WriteLine("Name: {0}", MyData.Name);
         Console.WriteLine("Address: {0}", MyData.Address);
         Console.WriteLine("City: {0}", MyData.City);
         Console.WriteLine("State: {0}", MyData.State);
         Console.WriteLine("Zip: {0}", MyData.Zip);

         Console.WriteLine("\nThe following address will be used:");
         Console.WriteLine(MyData.Address);
         Console.WriteLine(MyData.CityStateZip);
      }
   }
}

public class MailToData
{
   string name = "";
   string address = "";
   string citystatezip = "";
   string city = "";
   string state = "";
   string zip = "";
   bool parseSucceeded = false;

   public string Name
   {
      get{return name;}
      set{name = value;}
   }
```

```
   public string Address
   {
      get{return address;}
      set{address = value;}
   }

   public string CityStateZip
   {
      get {
         return String.Format("{0}, {1} {2}", city, state, zip);
      }
      set {
         citystatezip = value.Trim();
         ParseCityStateZip();
      }
   }

   public string City
   {
      get{return city;}
      set{city = value;}
   }

   public string State
   {
      get{return state;}
      set{state = value;}
   }

   public string Zip
   {
      get{return zip;}
      set{zip = value;}
   }

   public bool Validated
   {
      get { return parseSucceeded; }
   }

   private void ParseCityStateZip()
   {
      string msg = "";
      const string msgEnd = "\nYou must enter spaces between city, state, and zip code.\n";

      // Throw a FormatException if the user did not enter the necessary spaces
      // between elements.
      try
      {
         // City may consist of multiple words, so we'll have to parse the
         // string from right to left starting with the zip code.
         int zipIndex = citystatezip.LastIndexOf(" ");
         if (zipIndex == -1) {
            msg = "\nCannot identify a zip code." + msgEnd;
            throw new FormatException(msg);
         }
         zip = citystatezip.Substring(zipIndex + 1);

         int stateIndex = citystatezip.LastIndexOf(" ", zipIndex - 1);
         if (stateIndex == -1) {
            msg = "\nCannot identify a state." + msgEnd;
            throw new FormatException(msg);
         }
         state = citystatezip.Substring(stateIndex + 1, zipIndex - stateIndex - 1);
         state = state.ToUpper();

         city = citystatezip.Substring(0, stateIndex);
         if (city.Length == 0) {
```

```
            msg = "\nCannot identify a city." + msgEnd;
            throw new FormatException(msg);
         }
         parseSucceeded = true;
      }
      catch (FormatException ex)
      {
         Console.WriteLine(ex.Message);
      }
   }

   private string ReturnCityStateZip()
    {
        // Make state uppercase.
        state = state.ToUpper();

        // Put the value of city, state, and zip together in the proper manner.
        string MyCityStateZip = String.Concat(city, ", ", state, " ", zip);

        return MyCityStateZip;
    }
}
```

```
Class MainClass
   Public Shared Sub Main()
      Dim MyData As New MailToData()

      Console.Write("Enter Your Name: ")
      MyData.Name = Console.ReadLine()
      Console.Write("Enter Your Address: ")
      MyData.Address = Console.ReadLine()
      Console.Write("Enter Your City, State, and ZIP Code separated by spaces: ")
      MyData.CityStateZip = Console.ReadLine()
      Console.WriteLine()

      If MyData.Validated Then
         Console.WriteLine("Name: {0}", MyData.Name)
         Console.WriteLine("Address: {0}", MyData.Address)
         Console.WriteLine("City: {0}", MyData.City)
         Console.WriteLine("State: {0}", MyData.State)
         Console.WriteLine("ZIP Code: {0}", MyData.Zip)

         Console.WriteLine("The following address will be used:")
         Console.WriteLine(MyData.Address)
         Console.WriteLine(MyData.CityStateZip)
      End If
   End Sub
End Class

Public Class MailToData
   Private strName As String = ""
   Private strAddress As String = ""
   Private strCityStateZip As String = ""
   Private strCity As String = ""
   Private strState As String = ""
   Private strZip As String = ""
   Private parseSucceeded As Boolean = False

   Public Property Name() As String
      Get
         Return strName
      End Get
      Set
         strName = value
      End Set
   End Property

   Public Property Address() As String
```

```
   Public Property Address() As String
      Get
         Return strAddress
      End Get
      Set
         strAddress = value
      End Set
   End Property

   Public Property CityStateZip() As String
      Get
         Return String.Format("{0}, {1} {2}", strCity, strState, strZip)
      End Get
      Set
         strCityStateZip = value.Trim()
         ParseCityStateZip()
      End Set
   End Property

   Public Property City() As String
      Get
         Return strCity
      End Get
      Set
         strCity = value
      End Set
   End Property

   Public Property State() As String
      Get
         Return strState
      End Get
      Set
         strState = value
      End Set
   End Property

   Public Property Zip() As String
      Get
         Return strZip
      End Get
      Set
         strZip = value
      End Set
   End Property

   Public ReadOnly Property Validated As Boolean
      Get
         Return parseSucceeded
      End Get
   End Property

   Private Sub ParseCityStateZip()
      Dim msg As String = Nothing
      Const msgEnd As String = vbCrLf +
                               "You must enter spaces between city, state, and zip code." +
                               vbCrLf

      ' Throw a FormatException if the user did not enter the necessary spaces
      ' between elements.
      Try
         ' City may consist of multiple words, so we'll have to parse the
         ' string from right to left starting with the zip code.
         Dim zipIndex As Integer = strCityStateZip.LastIndexOf(" ")
         If zipIndex = -1 Then
            msg = vbCrLf + "Cannot identify a zip code." + msgEnd
            Throw New FormatException(msg)
         End If
         strZip = strCityStateZip.Substring(zipIndex + 1)
```

```
        Dim stateIndex As Integer = strCityStateZip.LastIndexOf(" ", zipIndex - 1)
        If stateIndex = -1 Then
           msg = vbCrLf + "Cannot identify a state." + msgEnd
           Throw New FormatException(msg)
        End If
        strState = strCityStateZip.Substring(stateIndex + 1, zipIndex - stateIndex - 1)
        strState = strState.ToUpper()

        strCity = strCityStateZip.Substring(0, stateIndex)
        If strCity.Length = 0 Then
           msg = vbCrLf + "Cannot identify a city." + msgEnd
           Throw New FormatException(msg)
        End If
        parseSucceeded = True
     Catch ex As FormatException
        Console.WriteLine(ex.Message)
     End Try
  End Sub
End Class
```

When the preceding code is executed, the user is asked to enter his or her name and address. The application places the information in the appropriate properties and displays the information back to the user, creating a single string that displays the city, state, and ZIP Code information.

See also

- Basic String Operations

.NET Regular Expressions

5/2/2018 • 10 minutes to read • Edit Online

Regular expressions provide a powerful, flexible, and efficient method for processing text. The extensive pattern-matching notation of regular expressions enables you to quickly parse large amounts of text to find specific character patterns; to validate text to ensure that it matches a predefined pattern (such as an email address); to extract, edit, replace, or delete text substrings; and to add the extracted strings to a collection in order to generate a report. For many applications that deal with strings or that parse large blocks of text, regular expressions are an indispensable tool.

How Regular Expressions Work

The centerpiece of text processing with regular expressions is the regular expression engine, which is represented by the System.Text.RegularExpressions.Regex object in .NET. At a minimum, processing text using regular expressions requires that the regular expression engine be provided with the following two items of information:

- The regular expression pattern to identify in the text.

  In .NET, regular expression patterns are defined by a special syntax or language, which is compatible with Perl 5 regular expressions and adds some additional features such as right-to-left matching. For more information, see Regular Expression Language - Quick Reference.

- The text to parse for the regular expression pattern.

The methods of the Regex class let you perform the following operations:

- Determine whether the regular expression pattern occurs in the input text by calling the Regex.IsMatch method. For an example that uses the IsMatch method for validating text, see How to: Verify that Strings Are in Valid Email Format.

- Retrieve one or all occurrences of text that matches the regular expression pattern by calling the Regex.Match or Regex.Matches method. The former method returns a System.Text.RegularExpressions.Match object that provides information about the matching text. The latter returns a MatchCollection object that contains one System.Text.RegularExpressions.Match object for each match found in the parsed text.

- Replace text that matches the regular expression pattern by calling the Regex.Replace method. For examples that use the Replace method to change date formats and remove invalid characters from a string, see How to: Strip Invalid Characters from a String and Example: Changing Date Formats.

For an overview of the regular expression object model, see The Regular Expression Object Model.

For more information about the regular expression language, see Regular Expression Language - Quick Reference or download and print one of these brochures:

Quick Reference in Word (.docx) format
Quick Reference in PDF (.pdf) format

Regular Expression Examples

The String class includes a number of string search and replacement methods that you can use when you want to locate literal strings in a larger string. Regular expressions are most useful either when you want to locate one of several substrings in a larger string, or when you want to identify patterns in a string, as the following examples illustrate.

Example 1: Replacing Substrings

Assume that a mailing list contains names that sometimes include a title (Mr., Mrs., Miss, or Ms.) along with a first and last name. If you do not want to include the titles when you generate envelope labels from the list, you can use a regular expression to remove the titles, as the following example illustrates.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "(Mr\\.? |Mrs\\.? |Miss |Ms\\.? )";
      string[] names = { "Mr. Henry Hunt", "Ms. Sara Samuels",
                         "Abraham Adams", "Ms. Nicole Norris" };
      foreach (string name in names)
         Console.WriteLine(Regex.Replace(name, pattern, String.Empty));
   }
}
// The example displays the following output:
//    Henry Hunt
//    Sara Samuels
//    Abraham Adams
//    Nicole Norris
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(Mr\.? |Mrs\.? |Miss |Ms\.? )"
      Dim names() As String = { "Mr. Henry Hunt", "Ms. Sara Samuels", _
                                "Abraham Adams", "Ms. Nicole Norris" }
      For Each name As String In names
         Console.WriteLine(Regex.Replace(name, pattern, String.Empty))
      Next
   End Sub
End Module
' The example displays the following output:
'    Henry Hunt
'    Sara Samuels
'    Abraham Adams
'    Nicole Norris
```

The regular expression pattern `(Mr\.? |Mrs\.? |Miss |Ms\.? )` matches any occurrence of "Mr ", "Mr. ", "Mrs ", "Mrs. ", "Miss ", "Ms or "Ms. ". The call to the Regex.Replace method replaces the matched string with String.Empty; in other words, it removes it from the original string.

Example 2: Identifying Duplicated Words

Accidentally duplicating words is a common error that writers make. A regular expression can be used to identify duplicated words, as the following example shows.

```
using System;
using System.Text.RegularExpressions;

public class Class1
{
   public static void Main()
   {
      string pattern = @"\b(\w+?)\s\1\b";
      string input = "This this is a nice day. What about this? This tastes good. I saw a a dog.";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine("{0} (duplicates '{1}') at position {2}",
                           match.Value, match.Groups[1].Value, match.Index);
   }
}
// The example displays the following output:
//       This this (duplicates 'This') at position 0
//       a a (duplicates 'a') at position 66
```

```
Imports System.Text.RegularExpressions

Module modMain
   Public Sub Main()
      Dim pattern As String = "\b(\w+?)\s\1\b"
      Dim input As String = "This this is a nice day. What about this? This tastes good. I saw a a dog."
      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
         Console.WriteLine("{0} (duplicates '{1}') at position {2}", _
                           match.Value, match.Groups(1).Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       This this (duplicates 'This') at position 0
'       a a (duplicates 'a') at position 66
```

The regular expression pattern `\b(\w+?)\s\1\b` can be interpreted as follows:

`\b`	Start at a word boundary.
(\w+?)	Match one or more word characters, but as few characters as possible. Together, they form a group that can be referred to as `\1`.
`\s`	Match a white-space character.
`\1`	Match the substring that is equal to the group named `\1`.
`\b`	Match a word boundary.

The Regex.Matches method is called with regular expression options set to RegexOptions.IgnoreCase. Therefore, the match operation is case-insensitive, and the example identifies the substring "This this" as a duplication.

Note that the input string includes the substring "this? This". However, because of the intervening punctuation mark, it is not identified as a duplication.

Example 3: Dynamically Building a Culture-Sensitive Regular Expression

The following example illustrates the power of regular expressions combined with the flexibility offered by .NET's globalization features. It uses the NumberFormatInfo object to determine the format of currency values in the system's current culture. It then uses that information to dynamically construct a regular expression that extracts currency values from the text. For each match, it extracts the subgroup that contains the numeric string only, converts it to a Decimal value, and calculates a running total.

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      // Define text to be parsed.
      string input = "Office expenses on 2/13/2008:\n" +
                     "Paper (500 sheets)                      $3.95\n" +
                     "Pencils (box of 10)                     $1.00\n" +
                     "Pens (box of 10)                        $4.49\n" +
                     "Erasers                                 $2.19\n" +
                     "Ink jet printer                        $69.95\n\n" +
                     "Total Expenses                        $ 81.58\n";

      // Get current culture's NumberFormatInfo object.
      NumberFormatInfo nfi = CultureInfo.CurrentCulture.NumberFormat;
      // Assign needed property values to variables.
      string currencySymbol = nfi.CurrencySymbol;
      bool symbolPrecedesIfPositive = nfi.CurrencyPositivePattern % 2 == 0;
      string groupSeparator = nfi.CurrencyGroupSeparator;
      string decimalSeparator = nfi.CurrencyDecimalSeparator;

      // Form regular expression pattern.
      string pattern = Regex.Escape( symbolPrecedesIfPositive ? currencySymbol : "") +
                       @"\s*[-+]?" + "([0-9]{0,3}(" + groupSeparator + "[0-9]{3})*(" +
                       Regex.Escape(decimalSeparator) + "[0-9]+)?)" +
                       (! symbolPrecedesIfPositive ? currencySymbol : "");
      Console.WriteLine( "The regular expression pattern is:");
      Console.WriteLine("   " + pattern);

      // Get text that matches regular expression pattern.
      MatchCollection matches = Regex.Matches(input, pattern,
                                              RegexOptions.IgnorePatternWhitespace);
      Console.WriteLine("Found {0} matches.", matches.Count);

      // Get numeric string, convert it to a value, and add it to List object.
      List<decimal> expenses = new List<Decimal>();

      foreach (Match match in matches)
         expenses.Add(Decimal.Parse(match.Groups[1].Value));

      // Determine whether total is present and if present, whether it is correct.
      decimal total = 0;
      foreach (decimal value in expenses)
         total += value;

      if (total / 2 == expenses[expenses.Count - 1])
         Console.WriteLine("The expenses total {0:C2}.", expenses[expenses.Count - 1]);
      else
         Console.WriteLine("The expenses total {0:C2}.", total);
   }
}
// The example displays the following output:
//       The regular expression pattern is:
//          \$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)
//       Found 6 matches.
//       The expenses total $81.58.
```

```vb
Imports System.Collections.Generic
Imports System.Globalization
Imports System.Text.RegularExpressions

Public Module Example
   Public Sub Main()
      ' Define text to be parsed.
      Dim input As String = "Office expenses on 2/13/2008:" + vbCrLf + _
                            "Paper (500 sheets)                      $3.95" + vbCrLf + _
                            "Pencils (box of 10)                     $1.00" + vbCrLf + _
                            "Pens (box of 10)                        $4.49" + vbCrLf + _
                            "Erasers                                 $2.19" + vbCrLf + _
                            "Ink jet printer                        $69.95" + vbCrLf + vbCrLf + _
                            "Total Expenses                        $ 81.58" + vbCrLf
      ' Get current culture's NumberFormatInfo object.
      Dim nfi As NumberFormatInfo = CultureInfo.CurrentCulture.NumberFormat
      ' Assign needed property values to variables.
      Dim currencySymbol As String = nfi.CurrencySymbol
      Dim symbolPrecedesIfPositive As Boolean = CBool(nfi.CurrencyPositivePattern Mod 2 = 0)
      Dim groupSeparator As String = nfi.CurrencyGroupSeparator
      Dim decimalSeparator As String = nfi.CurrencyDecimalSeparator

      ' Form regular expression pattern.
      Dim pattern As String = Regex.Escape(CStr(IIf(symbolPrecedesIfPositive, currencySymbol, ""))) + _
                              "\s*[-+]?" + "([0-9]{0,3}(" + groupSeparator + "[0-9]{3})*(" + _
                              Regex.Escape(decimalSeparator) + "[0-9]+)?)" + _
                              CStr(IIf(Not symbolPrecedesIfPositive, currencySymbol, ""))
      Console.WriteLine("The regular expression pattern is: ")
      Console.WriteLine("   " + pattern)

      ' Get text that matches regular expression pattern.
      Dim matches As MatchCollection = Regex.Matches(input, pattern, RegexOptions.IgnorePatternWhitespace)
      Console.WriteLine("Found {0} matches. ", matches.Count)

      ' Get numeric string, convert it to a value, and add it to List object.
      Dim expenses As New List(Of Decimal)

      For Each match As Match In matches
         expenses.Add(Decimal.Parse(match.Groups.Item(1).Value))
      Next

      ' Determine whether total is present and if present, whether it is correct.
      Dim total As Decimal
      For Each value As Decimal In expenses
         total += value
      Next

      If total / 2 = expenses(expenses.Count - 1) Then
         Console.WriteLine("The expenses total {0:C2}.", expenses(expenses.Count - 1))
      Else
         Console.WriteLine("The expenses total {0:C2}.", total)
      End If
   End Sub
End Module
' The example displays the following output:
'       The regular expression pattern is:
'          \$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)
'       Found 6 matches.
'       The expenses total $81.58.
```

On a computer whose current culture is English - United States (en-US), the example dynamically builds the regular expression `\$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)`. This regular expression pattern can be interpreted as follows:

`\$`	Look for a single occurrence of the dollar symbol ($) in the input string. The regular expression pattern string includes a backslash to indicate that the dollar symbol is to be interpreted literally rather than as a regular expression anchor. (The $ symbol alone would indicate that the regular expression engine should try to begin its match at the end of a string.) To ensure that the current culture's currency symbol is not misinterpreted as a regular expression symbol, the example calls the Escape method to escape the character.
`\s*`	Look for zero or more occurrences of a white-space character.
`[-+]?`	Look for zero or one occurrence of either a positive sign or a negative sign.
`([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)`	The outer parentheses around this expression define it as a capturing group or a subexpression. If a match is found, information about this part of the matching string can be retrieved from the second Group object in the GroupCollection object returned by the Match.Groups property. (The first element in the collection represents the entire match.)
`[0-9]{0,3}`	Look for zero to three occurrences of the decimal digits 0 through 9.
`(,[0-9]{3})*`	Look for zero or more occurrences of a group separator followed by three decimal digits.
`\.`	Look for a single occurrence of the decimal separator.
`[0-9]+`	Look for one or more decimal digits.
`(\.[0-9]+)?`	Look for zero or one occurrence of the decimal separator followed by at least one decimal digit.

If each of these subpatterns is found in the input string, the match succeeds, and a Match object that contains information about the match is added to the MatchCollection object.

Related Topics

TITLE	DESCRIPTION
Regular Expression Language - Quick Reference	Provides information on the set of characters, operators, and constructs that you can use to define regular expressions.
The Regular Expression Object Model	Provides information and code examples that illustrate how to use the regular expression classes.
Details of Regular Expression Behavior	Provides information about the capabilities and behavior of .NET regular expressions.
Regular Expression Examples	Provides code examples that illustrate typical uses of regular expressions.

Reference

System.Text.RegularExpressions
System.Text.RegularExpressions.Regex
Regular Expressions - Quick Reference (download in Word format)
Regular Expressions - Quick Reference (download in PDF format)

Regular Expression Language - Quick Reference

9/6/2018 • 11 minutes to read • Edit Online

A regular expression is a pattern that the regular expression engine attempts to match in input text. A pattern consists of one or more character literals, operators, or constructs. For a brief introduction, see .NET Regular Expressions.

Each section in this quick reference lists a particular category of characters, operators, and constructs that you can use to define regular expressions:

We've also provided this information in two formats that you can download and print for easy reference:

Download in Word (.docx) format
Download in PDF (.pdf) format

Character Escapes

The backslash character (\) in a regular expression indicates that the character that follows it either is a special character (as shown in the following table), or should be interpreted literally. For more information, see Character Escapes.

ESCAPED CHARACTER	DESCRIPTION	PATTERN	MATCHES
`\a`	Matches a bell character, \u0007.	`\a`	"\u0007" in "Error!" + '\u0007'
`\b`	In a character class, matches a backspace, \u0008.	`[\b]{3,}`	"\b\b\b\b" in "\b\b\b\b"
`\t`	Matches a tab, \u0009.	`(\w+)\t`	"item1\t", "item2\t" in "item1\titem2\t"
`\r`	Matches a carriage return, \u000D. (`\r` is not equivalent to the newline character, `\n`.)	`\r\n(\w+)`	"\r\nThese" in "\r\nThese are\ntwo lines."
`\v`	Matches a vertical tab, \u000B.	`[\v]{2,}`	"\v\v\v" in "\v\v\v"

ESCAPED CHARACTER	DESCRIPTION	PATTERN	MATCHES
`\f`	Matches a form feed, \u000C.	`[\f]{2,}`	"\f\f\f" in "\f\f\f"
`\n`	Matches a new line, \u000A.	`\r\n(\w+)`	"\r\nThese" in "\r\nThese are\ntwo lines."
`\e`	Matches an escape, \u001B.	`\e`	"\x001B" in "\x001B"
`\` *nnn*	Uses octal representation to specify a character (*nnn* consists of two or three digits).	`\w\040\w`	"a b", "c d" in "a bc d"
`\x` *nn*	Uses hexadecimal representation to specify a character (*nn* consists of exactly two digits).	`\w\x20\w`	"a b", "c d" in "a bc d"
`\c` *X* `\c` *x*	Matches the ASCII control character that is specified by *X* or *x*, where *X* or *x* is the letter of the control character.	`\cC`	"\x0003" in "\x0003" (Ctrl-C)
`\u` *nnnn*	Matches a Unicode character by using hexadecimal representation (exactly four digits, as represented by *nnnn*).	`\w\u0020\w`	"a b", "c d" in "a bc d"
`\`	When followed by a character that is not recognized as an escaped character in this and other tables in this topic, matches that character. For example, `\*` is the same as `\x2A`, and `\.` is the same as `\x2E`. This allows the regular expression engine to disambiguate language elements (such as * or ?) and character literals (represented by `\*` or `\?`).	`\d+[\+-x\*]\d+`	"2+2" and "3*9" in "(2+2) * 3*9"

Back to top

Character Classes

A character class matches any one of a set of characters. Character classes include the language elements listed in the following table. For more information, see Character Classes.

CHARACTER CLASS	DESCRIPTION	PATTERN	MATCHES
`[` *character_group* `]`	Matches any single character in *character_group*. By default, the match is case-sensitive.	`[ae]`	"a" in "gray" "a", "e" in "lane"
`[^` *character_group* `]`	Negation: Matches any single character that is not in *character_group*. By default, characters in *character_group* are case-sensitive.	`[^aei]`	"r", "g", "n" in "reign"
`[` *first* `-` *last* `]`	Character range: Matches any single character in the range from *first* to *last*.	`[A-Z]`	"A", "B" in "AB123"
`.`	Wildcard: Matches any single character except \n. To match a literal period character (. or `\u002E`), you must precede it with the escape character (`\.`).	`a.e`	"ave" in "nave" "ate" in "water"

`\p{` *name* `}`	Matches any single character in the Unicode general category or named block specified by *name*.	`\p{Lu}`
		
		`\p{IsCyrillic}`
			"C", "L" in "City Lights"
			
			"Д", "Ж" in "ДЖem"

`\P{` *name* `}`	Matches any single character that is not in the Unicode general category or named block specified by *name*.	`\P{Lu}`
		
		`\P{IsCyrillic}`
			"i", "t", "y" in "City"
			
			"e", "m" in "ДЖem"

`\w`	Matches any word character.	`\w`	"I", "D", "A", "1", "3" in "ID A1.3"
`\W`	Matches any non-word character.	`\W`	" ", "." in "ID A1.3"
`\s`	Matches any white-space character.	`\w\s`	"D " in "ID A1.3"
`\S`	Matches any non-white-space character.	`\s\S`	" _" in "int __ctr"
`\d`	Matches any decimal digit.	`\d`	"4" in "4 = IV"
`\D`	Matches any character other than a decimal digit.	`\D`	" ", "=", " ", "I", "V" in "4 = IV"

Back to top

Anchors

Anchors, or atomic zero-width assertions, cause a match to succeed or fail depending on the current position in the string, but they do not cause the engine to advance through the string or consume characters. The metacharacters listed in the following table are anchors. For more information, see Anchors.

ASSERTION	DESCRIPTION	PATTERN	MATCHES
`^`	By default, the match must start at the beginning of the string; in multiline mode, it must start at the beginning of the line.	`^\d{3}`	"901" in "901-333-"
`$`	By default, the match must occur at the end of the string or before `\n` at the end of the string; in multiline mode, it must occur before the end of the line or before `\n` at the end of the line.	`-\d{3}$`	"-333" in "-901-333"
`\A`	The match must occur at the start of the string.	`\A\d{3}`	"901" in "901-333-"
`\Z`	The match must occur at the end of the string or before `\n` at the end of the string.	`-\d{3}\Z`	"-333" in "-901-333"
`\z`	The match must occur at the end of the string.	`-\d{3}\z`	"-333" in "-901-333"
`\G`	The match must occur at the point where the previous match ended.	`\G\(\d\)`	"(1)", "(3)", "(5)" in "(1)(3)(5)[7](9)"
`\b`	The match must occur on a boundary between a `\w` (alphanumeric) and a `\W` (nonalphanumeric) character.	`\b\w+\s\w+\b`	"them theme", "them them" in "them theme them them"
`\B`	The match must not occur on a `\b` boundary.	`\Bend\w*\b`	"ends", "ender" in "end sends endure lender"

Back to top

Grouping Constructs

Grouping constructs delineate subexpressions of a regular expression and typically capture substrings of an input string. Grouping constructs include the language elements listed in the following table. For more information, see Grouping Constructs.

GROUPING CONSTRUCT	DESCRIPTION	PATTERN	MATCHES
`(` *subexpression* `)`	Captures the matched subexpression and assigns it a one-based ordinal number.	`(\w)\1`	"ee" in "deep"
`(?<` *name* `>` *subexpression* `)`	Captures the matched subexpression into a named group.	`(?<double>\w)\k<double>`	"ee" in "deep"
`(?<` *name1* `-` *name2* `>` *subexpression* `)`	Defines a balancing group definition. For more information, see the "Balancing Group Definition" section in Grouping Constructs.	`(((?'Open'\()[^\(\)]*)+((?'Close-Open'\))[^\(\)]*)+)*(?(Open)(?!))$`	"((1-3)*(3-1))" in "3+2^((1-3)*(3-1))"
`(?:` *subexpression* `)`	Defines a noncapturing group.	`Write(?:Line)?`	"WriteLine" in "Console.WriteLine()" "Write" in "Console.Write(value)"
`(?imnsx-imnsx:` *subexpression* `)`	Applies or disables the specified options within *subexpression*. For more information, see Regular Expression Options.	`A\d{2}(?i:\w+)\b`	"A12xl", "A12XL" in "A12xl A12XL a12xl"
`(?=` *subexpression* `)`	Zero-width positive lookahead assertion.	`\w+(?=\.)`	"is", "ran", and "out" in "He is. The dog ran. The sun is out."
`(?!` *subexpression* `)`	Zero-width negative lookahead assertion.	`\b(?!un)\w+\b`	"sure", "used" in "unsure sure unity used"
`(?<=` *subexpression* `)`	Zero-width positive lookbehind assertion.	`(?<=19)\d{2}\b`	"99", "50", "05" in "1851 1999 1950 1905 2003"
`(?<!` *subexpression* `)`	Zero-width negative lookbehind assertion.	`(?<!19)\d{2}\b`	"51", "03" in "1851 1999 1950 1905 2003"
`(?>` *subexpression* `)`	Nonbacktracking (or "greedy") subexpression.	`[13579](?>A+B+)`	"1ABB", "3ABB", and "5AB" in "1ABB 3ABBC 5AB 5AC"

Back to top

Quantifiers

A quantifier specifies how many instances of the previous element (which can be a character, a group, or a character class) must be present in the input string for a match to occur. Quantifiers include the language elements listed in the following table. For more information, see Quantifiers.

QUANTIFIER	DESCRIPTION	PATTERN	MATCHES

QUANTIFIER	DESCRIPTION	PATTERN	MATCHES
`*`	Matches the previous element zero or more times.	`\d*\.\d`	".0", "19.9", "219.9"
`+`	Matches the previous element one or more times.	`"be+"`	"bee" in "been", "be" in "bent"
`?`	Matches the previous element zero or one time.	`"rai?n"`	"ran", "rain"
`{` *n* `}`	Matches the previous element exactly *n* times.	`",\d{3}"`	",043" in "1,043.6", ",876", ",543", and ",210" in "9,876,543,210"
`{` *n* `,}`	Matches the previous element at least *n* times.	`"\d{2,}"`	"166", "29", "1930"
`{` *n* `,` *m* `}`	Matches the previous element at least *n* times, but no more than *m* times.	`"\d{3,5}"`	"166", "17668" "19302" in "193024"
`*?`	Matches the previous element zero or more times, but as few times as possible.	`\d*?\.\d`	".0", "19.9", "219.9"
`+?`	Matches the previous element one or more times, but as few times as possible.	`"be+?"`	"be" in "been", "be" in "bent"
`??`	Matches the previous element zero or one time, but as few times as possible.	`"rai??n"`	"ran", "rain"
`{` *n* `}?`	Matches the preceding element exactly *n* times.	`",\d{3}?"`	",043" in "1,043.6", ",876", ",543", and ",210" in "9,876,543,210"
`{` *n* `,}?`	Matches the previous element at least *n* times, but as few times as possible.	`"\d{2,}?"`	"166", "29", "1930"
`{` *n* `,` *m* `}?`	Matches the previous element between *n* and *m* times, but as few times as possible.	`"\d{3,5}?"`	"166", "17668" "193", "024" in "193024"

Back to top

Backreference Constructs

A backreference allows a previously matched subexpression to be identified subsequently in the same regular expression. The following table lists the backreference constructs supported by regular expressions in .NET. For more information, see Backreference Constructs.

BACKREFERENCE CONSTRUCT	DESCRIPTION	PATTERN	MATCHES
`\` *number*	Backreference. Matches the value of a numbered subexpression.	`(\w)\1`	"ee" in "seek"
`\k<` *name* `>`	Named backreference. Matches the value of a named expression.	`(?<char>\w)\k<char>`	"ee" in "seek"

Back to top

Alternation Constructs

Alternation constructs modify a regular expression to enable either/or matching. These constructs include the language elements listed in the following table. For more information, see Alternation Constructs.

ALTERNATION CONSTRUCT	DESCRIPTION	PATTERN	MATCHES
`|`	Matches any one element separated by the vertical bar (|) character.	`th(e|is|at)`	"the", "this" in "this is the day. "
`(?(` *expression* `)` *yes* `|` *no* `)`	Matches *yes* if the regular expression pattern designated by *expression* matches; otherwise, matches the optional *no* part. *expression* is interpreted as a zero-width assertion.	`(?(A)A\d{2}\b|\b\d{3}\b)`	"A10", "910" in "A10 C103 910"
`(?(` *name* `)` *yes* `|` *no* `)`	Matches *yes* if *name*, a named or numbered capturing group, has a match; otherwise, matches the optional *no*.	`(?<quoted>")?(?(quoted).+?"|\S+\s)`	Dogs.jpg, "Yiska playing.jpg" in "Dogs.jpg "Yiska playing.jpg""

Back to top

Substitutions

Substitutions are regular expression language elements that are supported in replacement patterns. For more information, see Substitutions. The metacharacters listed in the following table are atomic zero-width assertions.

CHARACTER	DESCRIPTION	PATTERN	REPLACEMENT PATTERN	INPUT STRING	RESULT STRING
`$` *number*	Substitutes the substring matched by group *number*.	`\b(\w+)(\s)(\w+)\b`	`$3$2$1`	"one two"	"two one"

CHARACTER	DESCRIPTION	PATTERN	REPLACEMENT PATTERN	INPUT STRING	RESULT STRING
`${` *name* `}`	Substitutes the substring matched by the named group *name*.	`\b(?<word1>\w+)(\s)(?<word2>\w+)\b`	`${word2} ${word1}`	"one two"	"two one"
`$$`	Substitutes a literal "$".	`\b(\d+)\s?USD`	`$$$1`	"103 USD"	"$103"
`$&`	Substitutes a copy of the whole match.	`\$?\d*\.?\d+`	`**$&**`	"$1.30"	"**$1.30**"
`` $` ``	Substitutes all the text of the input string before the match.	`B+`	`` $` ``	"AABBCC"	"AAAACC"
`$'`	Substitutes all the text of the input string after the match.	`B+`	`$'`	"AABBCC"	"AACCCC"
`$+`	Substitutes the last group that was captured.	`B+(C+)`	`$+`	"AABBCCDD"	AACCDD
`$_`	Substitutes the entire input string.	`B+`	`$_`	"AABBCC"	"AAAABBCCCC"

Back to top

Regular Expression Options

You can specify options that control how the regular expression engine interprets a regular expression pattern. Many of these options can be specified either inline (in the regular expression pattern) or as one or more RegexOptions constants. This quick reference lists only inline options. For more information about inline and RegexOptions options, see the article Regular Expression Options.

You can specify an inline option in two ways:

- By using the miscellaneous construct `(?imnsx-imnsx)`, where a minus sign (-) before an option or set of options turns those options off. For example, `(?i-mn)` turns case-insensitive matching (`i`) on, turns multiline mode (`m`) off, and turns unnamed group captures (`n`) off. The option applies to the regular expression pattern from the point at which the option is defined, and is effective either to the end of the pattern or to the point where another construct reverses the option.
- By using the grouping construct `(?imnsx-imnsx:` *subexpression* `)`, which defines options for the specified group only.

The .NET regular expression engine supports the following inline options.

OPTION	DESCRIPTION	PATTERN	MATCHES
`i`	Use case-insensitive matching.	`\b(?i)a(?-i)a\w+\b`	"aardvark", "aaaAuto" in "aardvark AAAuto aaaAuto Adam breakfast"
`m`	Use multiline mode. `^` and `$` match the beginning and end of a line, instead of the beginning and end of a string.	For an example, see the "Multiline Mode" section in Regular Expression Options.	
`n`	Do not capture unnamed groups.	For an example, see the "Explicit Captures Only" section in Regular Expression Options.	
`s`	Use single-line mode.	For an example, see the "Single-line Mode" section in Regular Expression Options.	
`x`	Ignore unescaped white space in the regular expression pattern.	`\b(?x) \d+ \s \w+`	"1 aardvark", "2 cats" in "1 aardvark 2 cats IV centurions"

Back to top

Miscellaneous Constructs

Miscellaneous constructs either modify a regular expression pattern or provide information about it. The following table lists the miscellaneous constructs supported by .NET. For more information, see Miscellaneous Constructs.

CONSTRUCT	DEFINITION	EXAMPLE
`(?imnsx-imnsx)`	Sets or disables options such as case insensitivity in the middle of a pattern.For more information, see Regular Expression Options.	`\bA(?i)b\w+\b` matches "ABA", "Able" in "ABA Able Act"
`(?#` *comment* `)`	Inline comment. The comment ends at the first closing parenthesis.	`\bA(?#Matches words starting with A)\w+\b`
`#` [to end of line]	X-mode comment. The comment starts at an unescaped `#` and continues to the end of the line.	`(?x)\bA\w+\b#Matches words starting with A`

See also

- System.Text.RegularExpressions
- Regex
- Regular Expressions
- Regular Expression Classes
- Regular Expression Examples

- Regular Expressions - Quick Reference (download in Word format)
- Regular Expressions - Quick Reference (download in PDF format)

Character Escapes in Regular Expressions

9/6/2018 • 4 minutes to read • Edit Online

The backslash (\) in a regular expression indicates one of the following:

- The character that follows it is a special character, as shown in the table in the following section. For example, `\b` is an anchor that indicates that a regular expression match should begin on a word boundary, `\t` represents a tab, and `\x020` represents a space.
- A character that otherwise would be interpreted as an unescaped language construct should be interpreted literally. For example, a brace (`{`) begins the definition of a quantifier, but a backslash followed by a brace (`\{`) indicates that the regular expression engine should match the brace. Similarly, a single backslash marks the beginning of an escaped language construct, but two backslashes (`\\`) indicate that the regular expression engine should match the backslash.

> **NOTE**
> Character escapes are recognized in regular expression patterns but not in replacement patterns.

Character Escapes in .NET

The following table lists the character escapes supported by regular expressions in .NET.

CHARACTER OR SEQUENCE	DESCRIPTION
All characters except for the following: . $ ^ { [(|) * + ? \	Characters other than those listed in the **Character or sequence** column have no special meaning in regular expressions; they match themselves. The characters included in the **Character or sequence** column are special regular expression language elements. To match them in a regular expression, they must be escaped or included in a positive character group. For example, the regular expression `\$\d+` or `[$]\d+` matches "$1200".
`\a`	Matches a bell (alarm) character, `\u0007`.
`\b`	In a `[` *character_group* `]` character class, matches a backspace, `\u0008`. (See Character Classes.) Outside a character class, `\b` is an anchor that matches a word boundary. (See Anchors.)
`\t`	Matches a tab, `\u0009`.
`\r`	Matches a carriage return, `\u000D`. Note that `\r` is not equivalent to the newline character, `\n`.
`\v`	Matches a vertical tab, `\u000B`.
`\f`	Matches a form feed, `\u000C`.

CHARACTER OR SEQUENCE	DESCRIPTION
`\n`	Matches a new line, `\u000A`.
`\e`	Matches an escape, `\u001B`.
`\` *nnn*	Matches an ASCII character, where *nnn* consists of two or three digits that represent the octal character code. For example, `\040` represents a space character. This construct is interpreted as a backreference if it has only one digit (for example, `\2`) or if it corresponds to the number of a capturing group. (See Backreference Constructs.)
`\x` *nn*	Matches an ASCII character, where *nn* is a two-digit hexadecimal character code.
`\c` *X*	Matches an ASCII control character, where X is the letter of the control character. For example, `\cC` is CTRL-C.
`\u` *nnnn*	Matches a UTF-16 code unit whose value is *nnnn* hexadecimal. **Note:** The Perl 5 character escape that is used to specify Unicode is not supported by .NET. The Perl 5 character escape has the form `\x{` *####* `…}`, where *####* `…` is a series of hexadecimal digits. Instead, use `\u` *nnnn*.
`\`	When followed by a character that is not recognized as an escaped character, matches that character. For example, `\*` matches an asterisk (*) and is the same as `\x2A`.

An Example

The following example illustrates the use of character escapes in a regular expression. It parses a string that contains the names of the world's largest cities and their populations in 2009. Each city name is separated from its population by a tab (`\t`) or a vertical bar (| or `\u007c`). Individual cities and their populations are separated from each other by a carriage return and line feed.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string delimited = @"\G(.+)[\t\u007c](.+)\r?\n";
      string input = "Mumbai, India|13,922,125\t\n" +
                            "Shanghai, China\t13,831,900\n" +
                            "Karachi, Pakistan|12,991,000\n" +
                            "Delhi, India\t12,259,230\n" +
                            "Istanbul, Turkey|11,372,613\n";
      Console.WriteLine("Population of the World's Largest Cities, 2009");
      Console.WriteLine();
      Console.WriteLine("{0,-20} {1,10}", "City", "Population");
      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, delimited))
         Console.WriteLine("{0,-20} {1,10}", match.Groups[1].Value,
                                            match.Groups[2].Value);
   }
}
// The example displyas the following output:
//       Population of the World's Largest Cities, 2009
//
//       City                 Population
//
//       Mumbai, India        13,922,125
//       Shanghai, China      13,831,900
//       Karachi, Pakistan    12,991,000
//       Delhi, India         12,259,230
//       Istanbul, Turkey     11,372,613
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim delimited As String = "\G(.+)[\t\u007c](.+)\r?\n"
      Dim input As String = "Mumbai, India|13,922,125" + vbCrLf + _
                            "Shanghai, China" + vbTab + "13,831,900" + vbCrLf + _
                            "Karachi, Pakistan|12,991,000" + vbCrLf + _
                            "Delhi, India" + vbTab + "12,259,230" + vbCrLf + _
                            "Istanbul, Turkey|11,372,613" + vbCrLf
      Console.WriteLine("Population of the World's Largest Cities, 2009")
      Console.WriteLine()
      Console.WriteLine("{0,-20} {1,10}", "City", "Population")
      Console.WriteLine()
      For Each match As Match In Regex.Matches(input, delimited)
         Console.WriteLine("{0,-20} {1,10}", match.Groups(1).Value, _
                                            match.Groups(2).Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       Population of the World's Largest Cities, 2009
'
'       City                 Population
'
'       Mumbai, India        13,922,125
'       Shanghai, China      13,831,900
'       Karachi, Pakistan    12,991,000
'       Delhi, India         12,259,230
'       Istanbul, Turkey     11,372,613
```

The regular expression `\G(.+)[\t|\u007c](.+)\r?\n` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\G`	Begin the match where the last match ended.
`(.+)`	Match any character one or more times. This is the first capturing group.
`[\t\u007c]`	Match a tab (`\t`) or a vertical bar (|).
`(.+)`	Match any character one or more times. This is the second capturing group.
`\r?\n`	Match zero or one occurrence of a carriage return followed by a new line.

See also

- Regular Expression Language - Quick Reference

Character Classes in Regular Expressions

9/6/2018 • 33 minutes to read • Edit Online

A character class defines a set of characters, any one of which can occur in an input string for a match to succeed. The regular expression language in .NET supports the following character classes:

- Positive character groups. A character in the input string must match one of a specified set of characters. For more information, see Positive Character Group.
- Negative character groups. A character in the input string must not match one of a specified set of characters. For more information, see Negative Character Group.
- Any character. The `.` (dot or period) character in a regular expression is a wildcard character that matches any character except `\n`. For more information, see Any Character.
- A general Unicode category or named block. A character in the input string must be a member of a particular Unicode category or must fall within a contiguous range of Unicode characters for a match to succeed. For more information, see Unicode Category or Unicode Block.
- A negative general Unicode category or named block. A character in the input string must not be a member of a particular Unicode category or must not fall within a contiguous range of Unicode characters for a match to succeed. For more information, see Negative Unicode Category or Unicode Block.
- A word character. A character in the input string can belong to any of the Unicode categories that are appropriate for characters in words. For more information, see Word Character.
- A non-word character. A character in the input string can belong to any Unicode category that is not a word character. For more information, see Non-Word Character.
- A white-space character. A character in the input string can be any Unicode separator character, as well as any one of a number of control characters. For more information, see White-Space Character.
- A non-white-space character. A character in the input string can be any character that is not a white-space character. For more information, see Non-White-Space Character.
- A decimal digit. A character in the input string can be any of a number of characters classified as Unicode decimal digits. For more information, see Decimal Digit Character.
- A non-decimal digit. A character in the input string can be anything other than a Unicode decimal digit. For more information, see Decimal Digit Character.

.NET supports character class subtraction expressions, which enables you to define a set of characters as the result of excluding one character class from another character class. For more information, see Character Class Subtraction.

> **NOTE**
>
> Character classes that match characters by category, such as \w to match word characters or \p{} to match a Unicode category, rely on the CharUnicodeInfo class to provide information about character categories. Starting with the .NET Framework 4.6.2, character categories are based on The Unicode Standard, Version 8.0.0. In the .NET Framework 4 through the .NET Framework 4.6.1, they are based on The Unicode Standard, Version 6.3.0.

Positive Character Group: [ ]

A positive character group specifies a list of characters, any one of which may appear in an input string for a match to occur. This list of characters may be specified individually, as a range, or both.

The syntax for specifying a list of individual characters is as follows:

[*character_group*]

where *character_group* is a list of the individual characters that can appear in the input string for a match to succeed. *character_group* can consist of any combination of one or more literal characters, escape characters, or character classes.

The syntax for specifying a range of characters is as follows:

```
[firstCharacter-lastCharacter]
```

where *firstCharacter* is the character that begins the range and *lastCharacter* is the character that ends the range. A character range is a contiguous series of characters defined by specifying the first character in the series, a hyphen (-), and then the last character in the series. Two characters are contiguous if they have adjacent Unicode code points.

Some common regular expression patterns that contain positive character classes are listed in the following table.

PATTERN	DESCRIPTION
`[aeiou]`	Match all vowels.
`[\p{P}\d]`	Match all punctuation and decimal digit characters.
`[\s\p{P}]`	Match all white space and punctuation.

The following example defines a positive character group that contains the characters "a" and "e" so that the input string must contain the words "grey" or "gray" followed by another word for a match to occur.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"gr[ae]y\s\S+?[\s\p{P}]";
      string input = "The gray wolf jumped over the grey wall.";
      MatchCollection matches = Regex.Matches(input, pattern);
      foreach (Match match in matches)
         Console.WriteLine($"'{match.Value}'");
   }
}
// The example displays the following output:
//       'gray wolf '
//       'grey wall.'
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "gr[ae]y\s\S+?[\s\p{P}]"
      Dim input As String = "The gray wolf jumped over the grey wall."
      Dim matches As MatchCollection = Regex.Matches(input, pattern)
      For Each match As Match In matches
         Console.WriteLine($"'{match.Value}'")
      Next
   End Sub
End Module
' The example displays the following output:
'       'gray wolf '
'       'grey wall.'
```

The regular expression `gr[ae]y\s\S+?[\s|\p{P}]` is defined as follows:

PATTERN	DESCRIPTION
`gr`	Match the literal characters "gr".
`[ae]`	Match either an "a" or an "e".
`y\s`	Match the literal character "y" followed by a white-space character.
`\S+?`	Match one or more non-white-space characters, but as few as possible.
`[\s\p{P}]`	Match either a white-space character or a punctuation mark.

The following example matches words that begin with any capital letter. It uses the subexpression `[A-Z]` to represent the range of capital letters from A to Z.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b[A-Z]\w*\b";
      string input = "A city Albany Zulu maritime Marseilles";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       A
//       Albany
//       Zulu
//       Marseilles
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b[A-Z]\w*\b"
      Dim input As String = "A city Albany Zulu maritime Marseilles"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Value)
      Next
   End Sub
End Module
```

The regular expression `\b[A-Z]\w*\b` is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`[A-Z]`	Match any uppercase character from A to Z.
`\w*`	Match zero or more word characters.
`\b`	Match a word boundary.

Back to Top

Negative Character Group: [^]

A negative character group specifies a list of characters that must not appear in an input string for a match to occur. The list of characters may be specified individually, as a range, or both.

The syntax for specifying a list of individual characters is as follows:

[^*character_group*]

where *character_group* is a list of the individual characters that cannot appear in the input string for a match to succeed. *character_group* can consist of any combination of one or more literal characters, escape characters, or character classes.

The syntax for specifying a range of characters is as follows:

[^*firstCharacter*-*lastCharacter*]

where *firstCharacter* is the character that begins the range, and *lastCharacter* is the character that ends the range. A character range is a contiguous series of characters defined by specifying the first character in the series, a hyphen (-), and then the last character in the series. Two characters are contiguous if they have adjacent Unicode code points.

Two or more character ranges can be concatenated. For example, to specify the range of decimal digits from "0" through "9", the range of lowercase letters from "a" through "f", and the range of uppercase letters from "A" through "F", use `[0-9a-fA-F]`.

The leading carat character (`^`) in a negative character group is mandatory and indicates the character group is a negative character group instead of a positive character group.

> **IMPORTANT**
> A negative character group in a larger regular expression pattern is not a zero-width assertion. That is, after evaluating the negative character group, the regular expression engine advances one character in the input string.

Some common regular expression patterns that contain negative character groups are listed in the following table.

PATTERN	DESCRIPTION
`[^aeiou]`	Match all characters except vowels.
`[^\p{P}\d]`	Match all characters except punctuation and decimal digit characters.

The following example matches any word that begins with the characters "th" and is not followed by an "o".

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\bth[^o]\w+\b";
      string input = "thought thing though them through thus thorough this";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       thing
//       them
//       through
//       thus
//       this
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\bth[^o]\w+\b"
      Dim input As String = "thought thing though them through thus " + _
                            "thorough this"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       thing
'       them
'       through
'       thus
'       this
```

The regular expression `\bth[^o]\w+\b` is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`th`	Match the literal characters "th".
`[^o]`	Match any character that is not an "o".
`\w+`	Match one or more word characters.
`\b`	End at a word boundary.

Back to Top

Any Character: .

The period character (.) matches any character except `\n` (the newline character, \u000A), with the following two qualifications:

- If a regular expression pattern is modified by the RegexOptions.Singleline option, or if the portion of the pattern that contains the `.` character class is modified by the `s` option, `.` matches any character. For more information, see Regular Expression Options.

  The following example illustrates the different behavior of the `.` character class by default and with the RegexOptions.Singleline option. The regular expression `^.+` starts at the beginning of the string and matches every character. By default, the match ends at the end of the first line; the regular expression pattern matches the carriage return character, `\r` or \u000D, but it does not match `\n`. Because the RegexOptions.Singleline option interprets the entire input string as a single line, it matches every character in the input string, including `\n`.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "^.+";
      string input = "This is one line and" + Environment.NewLine + "this is the second.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(Regex.Escape(match.Value));

      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Singleline))
         Console.WriteLine(Regex.Escape(match.Value));
   }
}
// The example displays the following output:
//       This\ is\ one\ line\ and\r
//
//       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "^.+"
      Dim input As String = "This is one line and" + vbCrLf + "this is the second."
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(Regex.Escape(match.Value))
      Next
      Console.WriteLine()
      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.SingleLine)
         Console.WriteLine(Regex.Escape(match.Value))
      Next
   End Sub
End Module
' The example displays the following output:
'       This\ is\ one\ line\ and\r
'
'       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

NOTE

Because it matches any character except `\n`, the `.` character class also matches `\r` (the carriage return character, \u000D).

- In a positive or negative character group, a period is treated as a literal period character, and not as a character class. For more information, see Positive Character Group and Negative Character Group earlier in this topic. The following example provides an illustration by defining a regular expression that includes the period character (`.`) both as a character class and as a member of a positive character group. The regular expression `\b.*[.?!;:](\s|\z)` begins at a word boundary, matches any character until it encounters one of five punctuation marks, including a period, and then matches either a white-space character or the end of the string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b.*[.?!;:](\s|\z)";
      string input = "this. what: is? go, thing.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       this. what: is? go, thing.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As STring = "\b.*[.?!;:](\s|\z)"
      Dim input As String = "this. what: is? go, thing."
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       this. what: is? go, thing.
```

NOTE

Because it matches any character, the `.` language element is often used with a lazy quantifier if a regular expression pattern attempts to match any character multiple times. For more information, see Quantifiers.

Back to Top

Unicode Category or Unicode Block: \p{}

The Unicode standard assigns each character a general category. For example, a particular character can be an uppercase letter (represented by the `Lu` category), a decimal digit (the `Nd` category), a math symbol (the `Sm` category), or a paragraph separator (the `Zl` category). Specific character sets in the Unicode standard also occupy a specific range or block of consecutive code points. For example, the basic Latin character set is found from \u0000 through \u007F, while the Arabic character set is found from \u0600 through \u06FF.

The regular expression construct

`\p{` *name* `}`

matches any character that belongs to a Unicode general category or named block, where *name* is the category abbreviation or named block name. For a list of category abbreviations, see the Supported Unicode General Categories section later in this topic. For a list of named blocks, see the Supported Named Blocks section later in this topic.

The following example uses the `\p{` *name* `}` construct to match both a Unicode general category (in this case, the `Pd`, or Punctuation, Dash category) and a named block (the `IsGreek` and `IsBasicLatin` named blocks).

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\p{IsGreek}+(\s)?)+\p{Pd}\s(\p{IsBasicLatin}+(\s)?)+";
      string input = "Κατα Μαθθαίον - The Gospel of Matthew";

      Console.WriteLine(Regex.IsMatch(input, pattern));        // Displays True.
   }
}
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b(\p{IsGreek}+(\s)?)+\p{Pd}\s(\p{IsBasicLatin}+(\s)?)+"
      Dim input As String = "Κατα Μαθθαίον - The Gospel of Matthew"

      Console.WriteLine(Regex.IsMatch(input, pattern))         ' Displays True.
   End Sub
End Module
```

The regular expression `\b(\p{IsGreek}+(\s)?)+\p{Pd}\s(\p{IsBasicLatin}+(\s)?)+` is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`\p{IsGreek}+`	Match one or more Greek characters.
`(\s)?`	Match zero or one white-space character.
`(\p{IsGreek}+(\s)?)+`	Match the pattern of one or more Greek characters followed by zero or one white-space characters one or more times.
`\p{Pd}`	Match a Punctuation, Dash character.
`\s`	Match a white-space character.
`\p{IsBasicLatin}+`	Match one or more basic Latin characters.
`(\s)?`	Match zero or one white-space character.
`(\p{IsBasicLatin}+(\s)?)+`	Match the pattern of one or more basic Latin characters followed by zero or one white-space characters one or more times.

Back to Top

Negative Unicode Category or Unicode Block: \P{}

The Unicode standard assigns each character a general category. For example, a particular character can be an uppercase letter (represented by the `Lu` category), a decimal digit (the `Nd` category), a math symbol (the `Sm` category), or a paragraph separator (the `Zl` category). Specific character sets in the Unicode standard also occupy a specific range or block of consecutive code points. For example, the basic Latin character set is found from \u0000 through \u007F, while the Arabic character set is found from \u0600 through \u06FF.

The regular expression construct

`\P{` *name* `}`

matches any character that does not belong to a Unicode general category or named block, where *name* is the category abbreviation or named block name. For a list of category abbreviations, see the Supported Unicode General Categories section later in this topic. For a list of named blocks, see the Supported Named Blocks section later in this topic.

The following example uses the `\P{ name }` construct to remove any currency symbols (in this case, the `Sc`, or Symbol, Currency category) from numeric strings.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\P{Sc})+";

      string[] values = { "$164,091.78", "£1,073,142.68", "73¢", "€120" };
      foreach (string value in values)
         Console.WriteLine(Regex.Match(value, pattern).Value);
   }
}
// The example displays the following output:
//       164,091.78
//       1,073,142.68
//       73
//       120
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(\P{Sc})+"

      Dim values() As String = { "$164,091.78", "£1,073,142.68", "73¢", "€120"}
      For Each value As String In values
         Console.WriteLine(Regex.Match(value, pattern).Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       164,091.78
'       1,073,142.68
'       73
'       120
```

The regular expression pattern `(\P{Sc})+` matches one or more characters that are not currency symbols; it effectively strips any currency symbol from the result string.

Back to Top

Word Character: \w

`\w` matches any word character. A word character is a member of any of the Unicode categories listed in the following table.

CATEGORY	DESCRIPTION
Ll	Letter, Lowercase
Lu	Letter, Uppercase
Lt	Letter, Titlecase

CATEGORY	DESCRIPTION
Lo	Letter, Other
Lm	Letter, Modifier
Mn	Mark, Nonspacing
Nd	Number, Decimal Digit
Pc	Punctuation, Connector. This category includes ten characters, the most commonly used of which is the LOWLINE character (_), u+005F.

If ECMAScript-compliant behavior is specified, `\w` is equivalent to `[a-zA-Z_0-9]`. For information on ECMAScript regular expressions, see the "ECMAScript Matching Behavior" section in Regular Expression Options.

> **NOTE**
>
> Because it matches any word character, the `\w` language element is often used with a lazy quantifier if a regular expression pattern attempts to match any word character multiple times, followed by a specific word character. For more information, see Quantifiers.

The following example uses the `\w` language element to match duplicate characters in a word. The example defines a regular expression pattern, `(\w)\1`, which can be interpreted as follows.

ELEMENT	DESCRIPTION
(\w)	Match a word character. This is the first capturing group.
\1	Match the value of the first capture.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\w)\1";
      string[] words = { "trellis", "seer", "latter", "summer",
                         "hoarse", "lesser", "aardvark", "stunned" };
      foreach (string word in words)
      {
         Match match = Regex.Match(word, pattern);
         if (match.Success)
            Console.WriteLine("'{0}' found in '{1}' at position {2}.",
                              match.Value, word, match.Index);
         else
            Console.WriteLine("No double characters in '{0}'.", word);
      }
   }
}
// The example displays the following output:
//       'll' found in 'trellis' at position 3.
//       'ee' found in 'seer' at position 1.
//       'tt' found in 'latter' at position 2.
//       'mm' found in 'summer' at position 2.
//       No double characters in 'hoarse'.
//       'ss' found in 'lesser' at position 2.
//       'aa' found in 'aardvark' at position 0.
//       'nn' found in 'stunned' at position 3.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(\w)\1"
      Dim words() As String = { "trellis", "seer", "latter", "summer", _
                                "hoarse", "lesser", "aardvark", "stunned" }
      For Each word As String In words
         Dim match As Match = Regex.Match(word, pattern)
         If match.Success Then
            Console.WriteLine("'{0}' found in '{1}' at position {2}.", _
                              match.Value, word, match.Index)
         Else
            Console.WriteLine("No double characters in '{0}'.", word)
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'       'll' found in 'trellis' at position 3.
'       'ee' found in 'seer' at position 1.
'       'tt' found in 'latter' at position 2.
'       'mm' found in 'summer' at position 2.
'       No double characters in 'hoarse'.
'       'ss' found in 'lesser' at position 2.
'       'aa' found in 'aardvark' at position 0.
'       'nn' found in 'stunned' at position 3.
```

Back to Top

Non-Word Character: \W

`\W` matches any non-word character. The \W language element is equivalent to the following character class:

```
[^\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]
```

In other words, it matches any character except for those in the Unicode categories listed in the following table.

CATEGORY	DESCRIPTION
Ll	Letter, Lowercase
Lu	Letter, Uppercase
Lt	Letter, Titlecase
Lo	Letter, Other
Lm	Letter, Modifier
Mn	Mark, Nonspacing
Nd	Number, Decimal Digit
Pc	Punctuation, Connector. This category includes ten characters, the most commonly used of which is the LOWLINE character (_), u+005F.

If ECMAScript-compliant behavior is specified, `\W` is equivalent to `[^a-zA-Z_0-9]`. For information on ECMAScript regular expressions, see the "ECMAScript Matching Behavior" section in Regular Expression Options.

> **NOTE**
>
> Because it matches any non-word character, the `\W` language element is often used with a lazy quantifier if a regular expression pattern attempts to match any non-word character multiple times followed by a specific non-word character. For more information, see Quantifiers.

The following example illustrates the `\W` character class. It defines a regular expression pattern, `\b(\w+)(\W){1,2}`, that matches a word followed by one or two non-word characters, such as white space or punctuation. The regular expression is interpreted as shown in the following table.

ELEMENT	DESCRIPTION
\b	Begin the match at a word boundary.
(\w+)	Match one or more word characters. This is the first capturing group.
(\W){1,2}	Match a non-word character either one or two times. This is the second capturing group.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+)(\W){1,2}";
      string input = "The old, grey mare slowly walked across the narrow, green pasture.";
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine(match.Value);
         Console.Write("   Non-word character(s):");
         CaptureCollection captures = match.Groups[2].Captures;
         for (int ctr = 0; ctr < captures.Count; ctr++)
             Console.Write(@"'{0}' (\u{1}){2}", captures[ctr].Value,
                           Convert.ToUInt16(captures[ctr].Value[0]).ToString("X4"),
                           ctr < captures.Count - 1 ? ", " : "");
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       The
//          Non-word character(s):' ' (\u0020)
//       old,
//          Non-word character(s):',' (\u002C), ' ' (\u0020)
//       grey
//          Non-word character(s):' ' (\u0020)
//       mare
//          Non-word character(s):' ' (\u0020)
//       slowly
//          Non-word character(s):' ' (\u0020)
//       walked
//          Non-word character(s):' ' (\u0020)
//       across
//          Non-word character(s):' ' (\u0020)
//       the
//          Non-word character(s):' ' (\u0020)
//       narrow,
//          Non-word character(s):',' (\u002C), ' ' (\u0020)
//       green
//          Non-word character(s):' ' (\u0020)
//       pasture.
//          Non-word character(s):'.' (\u002E)
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b(\w+)(\W){1,2}"
      Dim input As String = "The old, grey mare slowly walked across the narrow, green pasture."
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Value)
         Console.Write("   Non-word character(s):")
         Dim captures As CaptureCollection = match.Groups(2).Captures
         For ctr As Integer = 0 To captures.Count - 1
             Console.Write("'{0}' (\u{1}){2}", captures(ctr).Value, _
                           Convert.ToUInt16(captures(ctr).Value.Chars(0)).ToString("X4"), _
                           If(ctr < captures.Count - 1, ", ", ""))
         Next
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays the following output:
'       The
'          Non-word character(s):' ' (\u0020)
'       old,
'          Non-word character(s):',' (\u002C), ' ' (\u0020)
'       grey
'          Non-word character(s):' ' (\u0020)
'       mare
'          Non-word character(s):' ' (\u0020)
'       slowly
'          Non-word character(s):' ' (\u0020)
'       walked
'          Non-word character(s):' ' (\u0020)
'       across
'          Non-word character(s):' ' (\u0020)
'       the
'          Non-word character(s):' ' (\u0020)
'       narrow,
'          Non-word character(s):',' (\u002C), ' ' (\u0020)
'       green
'          Non-word character(s):' ' (\u0020)
'       pasture.
'          Non-word character(s):'.' (\u002E)
```

Because the Group object for the second capturing group contains only a single captured non-word character, the example retrieves all captured non-word characters from the CaptureCollection object that is returned by the Group.Captures property.

Back to Top

White-Space Character: \s

`\s` matches any white-space character. It is equivalent to the escape sequences and Unicode categories listed in the following table.

CATEGORY	DESCRIPTION
`\f`	The form feed character, \u000C.
`\n`	The newline character, \u000A.
`\r`	The carriage return character, \u000D.

CATEGORY	DESCRIPTION
`\t`	The tab character, \u0009.
`\v`	The vertical tab character, \u000B.
`\x85`	The ellipsis or NEXT LINE (NEL) character (…), \u0085.
`\p{Z}`	Matches any separator character.

If ECMAScript-compliant behavior is specified, `\s` is equivalent to `[ \f\n\r\t\v]`. For information on ECMAScript regular expressions, see the "ECMAScript Matching Behavior" section in Regular Expression Options.

The following example illustrates the `\s` character class. It defines a regular expression pattern, `\b\w+(e)?s(\s|$)`, that matches a word ending in either "s" or "es" followed by either a white-space character or the end of the input string. The regular expression is interpreted as shown in the following table.

ELEMENT	DESCRIPTION
\b	Begin the match at a word boundary.
\w+	Match one or more word characters.
(e)?	Match an "e" either zero or one time.
s	Match an "s".
(\s|$)	Match either a white-space character or the end of the input string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\w+(e)?s(\s|$)";
      string input = "matches stores stops leave leaves";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       matches
//       stores
//       stops
//       leaves
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b\w+(e)?s(\s|$)"
      Dim input As String = "matches stores stops leave leaves"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       matches
'       stores
'       stops
'       leaves
```

Back to Top

Non-White-Space Character: \S

`\S` matches any non-white-space character. It is equivalent to the `[^\f\n\r\t\v\x85\p{Z}]` regular expression pattern, or the opposite of the regular expression pattern that is equivalent to `\s`, which matches white-space characters. For more information, see White-Space Character: \s.

If ECMAScript-compliant behavior is specified, `\S` is equivalent to `[^ \f\n\r\t\v]`. For information on ECMAScript regular expressions, see the "ECMAScript Matching Behavior" section in Regular Expression Options.

The following example illustrates the `\S` language element. The regular expression pattern `\b(\S+)\s?` matches strings that are delimited by white-space characters. The second element in the match's GroupCollection object contains the matched string. The regular expression can be interpreted as shown in the following table.

ELEMENT	DESCRIPTION
`\b`	Begin the match at a word boundary.
`(\S+)`	Match one or more non-white-space characters. This is the first capturing group.
`\s?`	Match zero or one white-space character.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\S+)\s?";
      string input = "This is the first sentence of the first paragraph. " +
                            "This is the second sentence.\n" +
                            "This is the only sentence of the second paragraph.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Groups[1]);
   }
}
// The example displays the following output:
//    This
//    is
//    the
//    first
//    sentence
//    of
//    the
//    first
//    paragraph.
//    This
//    is
//    the
//    second
//    sentence.
//    This
//    is
//    the
//    only
//    sentence
//    of
//    the
//    second
//    paragraph.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b(\S+)\s?"
      Dim input As String = "This is the first sentence of the first paragraph. " + _
                            "This is the second sentence." + vbCrLf + _
                            "This is the only sentence of the second paragraph."
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Groups(1))
      Next
   End Sub
End Module
' The example displays the following output:
'    This
'    is
'    the
'    first
'    sentence
'    of
'    the
'    first
'    paragraph.
'    This
'    is
'    the
'    second
'    sentence.
'    This
'    is
'    the
'    only
'    sentence
'    of
'    the
'    second
'    paragraph.
```

Back to Top

Decimal Digit Character: \d

`\d` matches any decimal digit. It is equivalent to the `\p{Nd}` regular expression pattern, which includes the standard decimal digits 0-9 as well as the decimal digits of a number of other character sets.

If ECMAScript-compliant behavior is specified, `\d` is equivalent to `[0-9]`. For information on ECMAScript regular expressions, see the "ECMAScript Matching Behavior" section in Regular Expression Options.

The following example illustrates the `\d` language element. It tests whether an input string represents a valid telephone number in the United States and Canada. The regular expression pattern `^(\(?\d{3}\)?[\s-])?\d{3}-\d{4}$` is defined as shown in the following table.

ELEMENT	DESCRIPTION
`^`	Begin the match at the beginning of the input string.
`\(?`	Match zero or one literal "(" character.
`\d{3}`	Match three decimal digits.

ELEMENT	DESCRIPTION
`\)?`	Match zero or one literal ")" character.
`[\s-]`	Match a hyphen or a white-space character.
`(\(?\d{3}\)?[\s-])?`	Match an optional opening parenthesis followed by three decimal digits, an optional closing parenthesis, and either a white-space character or a hyphen zero or one time. This is the first capturing group.
`\d{3}-\d{4}`	Match three decimal digits followed by a hyphen and four more decimal digits.
`$`	Match the end of the input string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"^(\(?\d{3}\)?[\s-])?\d{3}-\d{4}$";
      string[] inputs = { "111 111-1111", "222-2222", "222 333-444",
                          "(212) 111-1111", "111-AB1-1111",
                          "212-111-1111", "01 999-9999" };

      foreach (string input in inputs)
      {
         if (Regex.IsMatch(input, pattern))
            Console.WriteLine(input + ": matched");
         else
            Console.WriteLine(input + ": match failed");
      }
   }
}
// The example displays the following output:
//       111 111-1111: matched
//       222-2222: matched
//       222 333-444: match failed
//       (212) 111-1111: matched
//       111-AB1-1111: match failed
//       212-111-1111: matched
//       01 999-9999: match failed
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "^(\(?\d{3}\)?[\s-])?\d{3}-\d{4}$"
      Dim inputs() As String = { "111 111-1111", "222-2222", "222 333-444", _
                                 "(212) 111-1111", "111-AB1-1111", _
                                 "212-111-1111", "01 999-9999" }

      For Each input As String In inputs
         If Regex.IsMatch(input, pattern) Then
            Console.WriteLine(input + ": matched")
         Else
            Console.WriteLine(input + ": match failed")
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'       111 111-1111: matched
'       222-2222: matched
'       222 333-444: match failed
'       (212) 111-1111: matched
'       111-AB1-1111: match failed
'       212-111-1111: matched
'       01 999-9999: match failed
```

Back to Top

Non-Digit Character: \D

`\D` matches any non-digit character. It is equivalent to the `\P{Nd}` regular expression pattern.

If ECMAScript-compliant behavior is specified, `\D` is equivalent to `[^0-9]`. For information on ECMAScript regular expressions, see the "ECMAScript Matching Behavior" section in Regular Expression Options.

The following example illustrates the \D language element. It tests whether a string such as a part number consists of the appropriate combination of decimal and non-decimal characters. The regular expression pattern `^\D\d{1,5}\D*$` is defined as shown in the following table.

ELEMENT	DESCRIPTION
`^`	Begin the match at the beginning of the input string.
`\D`	Match a non-digit character.
`\d{1,5}`	Match from one to five decimal digits.
`\D*`	Match zero, one, or more non-decimal characters.
`$`	Match the end of the input string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"^\D\d{1,5}\D*$";
      string[] inputs = { "A1039C", "AA0001", "C18A", "Y938518" };

      foreach (string input in inputs)
      {
         if (Regex.IsMatch(input, pattern))
            Console.WriteLine(input + ": matched");
         else
            Console.WriteLine(input + ": match failed");
      }
   }
}
// The example displays the following output:
//       A1039C: matched
//       AA0001: match failed
//       C18A: matched
//       Y938518: match failed
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "^\D\d{1,5}\D*$"
      Dim inputs() As String = { "A1039C", "AA0001", "C18A", "Y938518" }

      For Each input As String In inputs
         If Regex.IsMatch(input, pattern) Then
            Console.WriteLine(input + ": matched")
         Else
            Console.WriteLine(input + ": match failed")
         End If
      Next
   End Sub
End Module
' The example displays the following output:
```

Back to Top

Supported Unicode General Categories

Unicode defines the general categories listed in the following table. For more information, see the "UCD File Format" and "General Category Values" subtopics at the Unicode Character Database.

CATEGORY	DESCRIPTION
`Lu`	Letter, Uppercase
`Ll`	Letter, Lowercase
`Lt`	Letter, Titlecase
`Lm`	Letter, Modifier

CATEGORY	DESCRIPTION
`Lo`	Letter, Other
`L`	All letter characters. This includes the `Lu`, `Ll`, `Lt`, `Lm`, and `Lo` characters.
`Mn`	Mark, Nonspacing
`Mc`	Mark, Spacing Combining
`Me`	Mark, Enclosing
`M`	All diacritic marks. This includes the `Mn`, `Mc`, and `Me` categories.
`Nd`	Number, Decimal Digit
`Nl`	Number, Letter
`No`	Number, Other
`N`	All numbers. This includes the `Nd`, `Nl`, and `No` categories.
`Pc`	Punctuation, Connector
`Pd`	Punctuation, Dash
`Ps`	Punctuation, Open
`Pe`	Punctuation, Close
`Pi`	Punctuation, Initial quote (may behave like Ps or Pe depending on usage)
`Pf`	Punctuation, Final quote (may behave like Ps or Pe depending on usage)
`Po`	Punctuation, Other
`P`	All punctuation characters. This includes the `Pc`, `Pd`, `Ps`, `Pe`, `Pi`, `Pf`, and `Po` categories.
`Sm`	Symbol, Math
`Sc`	Symbol, Currency
`Sk`	Symbol, Modifier
`So`	Symbol, Other

CATEGORY	DESCRIPTION
`S`	All symbols. This includes the `Sm`, `Sc`, `Sk`, and `So` categories.
`Zs`	Separator, Space
`Zl`	Separator, Line
`Zp`	Separator, Paragraph
`Z`	All separator characters. This includes the `Zs`, `Zl`, and `Zp` categories.
`Cc`	Other, Control
`Cf`	Other, Format
`Cs`	Other, Surrogate
`Co`	Other, Private Use
`Cn`	Other, Not Assigned (no characters have this property)
`C`	All control characters. This includes the `Cc`, `Cf`, `Cs`, `Co`, and `Cn` categories.

You can determine the Unicode category of any particular character by passing that character to the GetUnicodeCategory method. The following example uses the GetUnicodeCategory method to determine the category of each element in an array that contains selected Latin characters.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      char[] chars = { 'a', 'X', '8', ',', ' ', '\u0009', '!' };

      foreach (char ch in chars)
         Console.WriteLine("'{0}': {1}", Regex.Escape(ch.ToString()),
                           Char.GetUnicodeCategory(ch));
   }
}
// The example displays the following output:
//       'a': LowercaseLetter
//       'X': UppercaseLetter
//       '8': DecimalDigitNumber
//       ',': OtherPunctuation
//       '\ ': SpaceSeparator
//       '\t': Control
//       '!': OtherPunctuation
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim chars() As Char = { "a"c, "X"c, "8"c, ","c, " "c, ChrW(9), "!"c }

      For Each ch As Char In chars
         Console.WriteLine("'{0}': {1}", Regex.Escape(ch.ToString()), _
                           Char.GetUnicodeCategory(ch))
      Next
   End Sub
End Module
' The example displays the following output:
'       'a': LowercaseLetter
'       'X': UppercaseLetter
'       '8': DecimalDigitNumber
'       ',': OtherPunctuation
'       '\ ': SpaceSeparator
'       '\t': Control
'       '!': OtherPunctuation
```

Back to Top

Supported Named Blocks

.NET provides the named blocks listed in the following table. The set of supported named blocks is based on Unicode 4.0 and Perl 5.6.

CODE POINT RANGE	BLOCK NAME
0000 - 007F	`IsBasicLatin`
0080 - 00FF	`IsLatin-1Supplement`
0100 - 017F	`IsLatinExtended-A`
0180 - 024F	`IsLatinExtended-B`
0250 - 02AF	`IsIPAExtensions`
02B0 - 02FF	`IsSpacingModifierLetters`
0300 - 036F	`IsCombiningDiacriticalMarks`

0370 - 03FF	`IsGreek`
	-or-
	`IsGreekandCoptic`

0400 - 04FF	`IsCyrillic`
0500 - 052F	`IsCyrillicSupplement`
0530 - 058F	`IsArmenian`

CODE POINT RANGE	BLOCK NAME
0590 - 05FF	`IsHebrew`
0600 - 06FF	`IsArabic`
0700 - 074F	`IsSyriac`
0780 - 07BF	`IsThaana`
0900 - 097F	`IsDevanagari`
0980 - 09FF	`IsBengali`
0A00 - 0A7F	`IsGurmukhi`
0A80 - 0AFF	`IsGujarati`
0B00 - 0B7F	`IsOriya`
0B80 - 0BFF	`IsTamil`
0C00 - 0C7F	`IsTelugu`
0C80 - 0CFF	`IsKannada`
0D00 - 0D7F	`IsMalayalam`
0D80 - 0DFF	`IsSinhala`
0E00 - 0E7F	`IsThai`
0E80 - 0EFF	`IsLao`
0F00 - 0FFF	`IsTibetan`
1000 - 109F	`IsMyanmar`
10A0 - 10FF	`IsGeorgian`
1100 - 11FF	`IsHangulJamo`
1200 - 137F	`IsEthiopic`
13A0 - 13FF	`IsCherokee`
1400 - 167F	`IsUnifiedCanadianAboriginalSyllabics`
1680 - 169F	`IsOgham`

CODE POINT RANGE	BLOCK NAME
16A0 - 16FF	`IsRunic`
1700 - 171F	`IsTagalog`
1720 - 173F	`IsHanunoo`
1740 - 175F	`IsBuhid`
1760 - 177F	`IsTagbanwa`
1780 - 17FF	`IsKhmer`
1800 - 18AF	`IsMongolian`
1900 - 194F	`IsLimbu`
1950 - 197F	`IsTaiLe`
19E0 - 19FF	`IsKhmerSymbols`
1D00 - 1D7F	`IsPhoneticExtensions`
1E00 - 1EFF	`IsLatinExtendedAdditional`
1F00 - 1FFF	`IsGreekExtended`
2000 - 206F	`IsGeneralPunctuation`
2070 - 209F	`IsSuperscriptsandSubscripts`
20A0 - 20CF	`IsCurrencySymbols`

20D0 - 20FF	`IsCombiningDiacriticalMarksforSymbols`
	
	-or-
	
	`IsCombiningMarksforSymbols`

2100 - 214F	`IsLetterlikeSymbols`
2150 - 218F	`IsNumberForms`
2190 - 21FF	`IsArrows`
2200 - 22FF	`IsMathematicalOperators`
2300 - 23FF	`IsMiscellaneousTechnical`
2400 - 243F	`IsControlPictures`

CODE POINT RANGE	BLOCK NAME
2440 - 245F	`IsOpticalCharacterRecognition`
2460 - 24FF	`IsEnclosedAlphanumerics`
2500 - 257F	`IsBoxDrawing`
2580 - 259F	`IsBlockElements`
25A0 - 25FF	`IsGeometricShapes`
2600 - 26FF	`IsMiscellaneousSymbols`
2700 - 27BF	`IsDingbats`
27C0 - 27EF	`IsMiscellaneousMathematicalSymbols-A`
27F0 - 27FF	`IsSupplementalArrows-A`
2800 - 28FF	`IsBraillePatterns`
2900 - 297F	`IsSupplementalArrows-B`
2980 - 29FF	`IsMiscellaneousMathematicalSymbols-B`
2A00 - 2AFF	`IsSupplementalMathematicalOperators`
2B00 - 2BFF	`IsMiscellaneousSymbolsandArrows`
2E80 - 2EFF	`IsCJKRadicalsSupplement`
2F00 - 2FDF	`IsKangxiRadicals`
2FF0 - 2FFF	`IsIdeographicDescriptionCharacters`
3000 - 303F	`IsCJKSymbolsandPunctuation`
3040 - 309F	`IsHiragana`
30A0 - 30FF	`IsKatakana`
3100 - 312F	`IsBopomofo`
3130 - 318F	`IsHangulCompatibilityJamo`
3190 - 319F	`IsKanbun`
31A0 - 31BF	`IsBopomofoExtended`

CODE POINT RANGE	BLOCK NAME
31F0 - 31FF	`IsKatakanaPhoneticExtensions`
3200 - 32FF	`IsEnclosedCJKLettersandMonths`
3300 - 33FF	`IsCJKCompatibility`
3400 - 4DBF	`IsCJKUnifiedIdeographsExtensionA`
4DC0 - 4DFF	`IsYijingHexagramSymbols`
4E00 - 9FFF	`IsCJKUnifiedIdeographs`
A000 - A48F	`IsYiSyllables`
A490 - A4CF	`IsYiRadicals`
AC00 - D7AF	`IsHangulSyllables`
D800 - DB7F	`IsHighSurrogates`
DB80 - DBFF	`IsHighPrivateUseSurrogates`
DC00 - DFFF	`IsLowSurrogates`
E000 - F8FF	`IsPrivateUse` or `IsPrivateUseArea`
F900 - FAFF	`IsCJKCompatibilityIdeographs`
FB00 - FB4F	`IsAlphabeticPresentationForms`
FB50 - FDFF	`IsArabicPresentationForms-A`
FE00 - FE0F	`IsVariationSelectors`
FE20 - FE2F	`IsCombiningHalfMarks`
FE30 - FE4F	`IsCJKCompatibilityForms`
FE50 - FE6F	`IsSmallFormVariants`
FE70 - FEFF	`IsArabicPresentationForms-B`
FF00 - FFEF	`IsHalfwidthandFullwidthForms`
FFF0 - FFFF	`IsSpecials`

Back to Top

Character Class Subtraction: [base_group - [excluded_group]]

A character class defines a set of characters. Character class subtraction yields a set of characters that is the result of excluding the characters in one character class from another character class.

A character class subtraction expression has the following form:

`[` *base_group* `-[` *excluded_group* `]]`

The square brackets (`[]`) and hyphen (`-`) are mandatory. The *base_group* is a positive character group or a negative character group. The *excluded_group* component is another positive or negative character group, or another character class subtraction expression (that is, you can nest character class subtraction expressions).

For example, suppose you have a base group that consists of the character range from "a" through "z". To define the set of characters that consists of the base group except for the character "m", use `[a-z-[m]]`. To define the set of characters that consists of the base group except for the set of characters "d", "j", and "p", use `[a-z-[djp]]`. To define the set of characters that consists of the base group except for the character range from "m" through "p", use `[a-z-[m-p]]`.

Consider the nested character class subtraction expression, `[a-z-[d-w-[m-o]]]`. The expression is evaluated from the innermost character range outward. First, the character range from "m" through "o" is subtracted from the character range "d" through "w", which yields the set of characters from "d" through "l" and "p" through "w". That set is then subtracted from the character range from "a" through "z", which yields the set of characters `[abcmnoxyz]`.

You can use any character class with character class subtraction. To define the set of characters that consists of all Unicode characters from \u0000 through \uFFFF except white-space characters (`\s`), the characters in the punctuation general category (`\p{P}`), the characters in the `IsGreek` named block (`\p{IsGreek}`), and the Unicode NEXT LINE control character (\x85), use `[\u0000-\uFFFF-[\s\p{P}\p{IsGreek}\x85]]`.

Choose character classes for a character class subtraction expression that will yield useful results. Avoid an expression that yields an empty set of characters, which cannot match anything, or an expression that is equivalent to the original base group. For example, the empty set is the result of the expression `[\p{IsBasicLatin}-[\x00-\x7F]]`, which subtracts all characters in the `IsBasicLatin` character range from the `IsBasicLatin` general category. Similarly, the original base group is the result of the expression `[a-z-[0-9]]`. This is because the base group, which is the character range of letters from "a" through "z", does not contain any characters in the excluded group, which is the character range of decimal digits from "0" through "9".

The following example defines a regular expression, `^[0-9-[2468]]+$`, that matches zero and odd digits in an input string. The regular expression is interpreted as shown in the following table.

ELEMENT	DESCRIPTION
^	Begin the match at the start of the input string.
`[0-9-[2468]]+`	Match one or more occurrences of any character from 0 to 9 except for 2, 4, 6, and 8. In other words, match one or more occurrences of zero or an odd digit.
$	End the match at the end of the input string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "123", "13579753", "3557798", "335599901" };
      string pattern = @"^[0-9-[2468]]+$";

      foreach (string input in inputs)
      {
         Match match = Regex.Match(input, pattern);
         if (match.Success)
            Console.WriteLine(match.Value);
      }
   }
}
// The example displays the following output:
//       13579753
//       335599901
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim inputs() As String = { "123", "13579753", "3557798", "335599901" }
      Dim pattern As String = "^[0-9-[2468]]+$"

      For Each input As String In inputs
         Dim match As Match = Regex.Match(input, pattern)
         If match.Success Then Console.WriteLine(match.Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       13579753
'       335599901
```

See also

- GetUnicodeCategory
- Regular Expression Language - Quick Reference
- Regular Expression Options

Anchors in Regular Expressions

9/6/2018 • 23 minutes to read • Edit Online

Anchors, or atomic zero-width assertions, specify a position in the string where a match must occur. When you use an anchor in your search expression, the regular expression engine does not advance through the string or consume characters; it looks for a match in the specified position only. For example, `^` specifies that the match must start at the beginning of a line or string. Therefore, the regular expression `^http:` matches "http:" only when it occurs at the beginning of a line. The following table lists the anchors supported by the regular expressions in .NET.

ANCHOR	DESCRIPTION
`^`	By default, the match must occur at the beginning of the string; in multiline mode, it must occur at the beginning of the line. For more information, see Start of String or Line.
`$`	By default, the match must occur at the end of the string or before `\n` at the end of the string; in multiline mode, it must occur at the end of the line or before `\n` at the end of the line. For more information, see End of String or Line.
`\A`	The match must occur at the beginning of the string only (no multiline support). For more information, see Start of String Only.
`\Z`	The match must occur at the end of the string, or before `\n` at the end of the string. For more information, see End of String or Before Ending Newline.
`\z`	The match must occur at the end of the string only. For more information, see End of String Only.
`\G`	The match must start at the position where the previous match ended. For more information, see Contiguous Matches.
`\b`	The match must occur on a word boundary. For more information, see Word Boundary.
`\B`	The match must not occur on a word boundary. For more information, see Non-Word Boundary.

Start of String or Line: ^

By default, the `^` anchor specifies that the following pattern must begin at the first character position of the string. If you use `^` with the RegexOptions.Multiline option (see Regular Expression Options), the match must occur at the beginning of each line.

The following example uses the `^` anchor in a regular expression that extracts information about the years during which some professional baseball teams existed. The example calls two overloads of the Regex.Matches method:

- The call to the Matches(String, String) overload finds only the first substring in the input string that matches the regular expression pattern.

- The call to the Matches(String, String, RegexOptions) overload with the `options` parameter set to RegexOptions.Multiline finds all five substrings.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      int startPos = 0, endPos = 70;
      string input = "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957\n" +
                     "Chicago Cubs, National League, 1903-present\n" +
                     "Detroit Tigers, American League, 1901-present\n" +
                     "New York Giants, National League, 1885-1957\n" +
                     "Washington Senators, American League, 1901-1960\n";
      string pattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+";
      Match match;

      if (input.Substring(startPos, endPos).Contains(",")) {
         match = Regex.Match(input, pattern);
         while (match.Success) {
            Console.Write("The {0} played in the {1} in",
                              match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
               Console.Write(capture.Value);

            Console.WriteLine(".");
            startPos = match.Index + match.Length;
            endPos = startPos + 70 <= input.Length ? 70 : input.Length - startPos;
            if (! input.Substring(startPos, endPos).Contains(",")) break;
            match = match.NextMatch();
         }
         Console.WriteLine();
      }

      if (input.Substring(startPos, endPos).Contains(",")) {
         match = Regex.Match(input, pattern, RegexOptions.Multiline);
         while (match.Success) {
            Console.Write("The {0} played in the {1} in",
                              match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
               Console.Write(capture.Value);

            Console.WriteLine(".");
            startPos = match.Index + match.Length;
            endPos = startPos + 70 <= input.Length ? 70 : input.Length - startPos;
            if (! input.Substring(startPos, endPos).Contains(",")) break;
            match = match.NextMatch();
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
//
//    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
//    The Chicago Cubs played in the National League in 1903-present.
//    The Detroit Tigers played in the American League in 1901-present.
//    The New York Giants played in the National League in 1885-1957.
//    The Washington Senators played in the American League in 1901-1960.
```

```
Imports System.Text.RegularExpressions

Module Example
```

```
Module Example
   Public Sub Main()
      Dim startPos As Integer = 0
      Dim endPos As Integer = 70
      Dim input As String = "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957" + vbCrLf + _
                            "Chicago Cubs, National League, 1903-present" + vbCrLf + _
                            "Detroit Tigers, American League, 1901-present" + vbCrLf + _
                            "New York Giants, National League, 1885-1957" + vbCrLf + _
                            "Washington Senators, American League, 1901-1960" + vbCrLf

      Dim pattern As String = "^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+"
      Dim match As Match

      ' Provide minimal validation in the event the input is invalid.
      If input.Substring(startPos, endPos).Contains(",") Then
         match = Regex.Match(input, pattern)
         Do While match.Success
            Console.Write("The {0} played in the {1} in", _
                              match.Groups(1).Value, match.Groups(4).Value)
            For Each capture As Capture In match.Groups(5).Captures
               Console.Write(capture.Value)
            Next
            Console.WriteLine(".")
            startPos = match.Index + match.Length
            endPos = CInt(IIf(startPos + 70 <= input.Length, 70, input.Length - startPos))
            If Not input.Substring(startPos, endPos).Contains(",") Then Exit Do
            match = match.NextMatch()
         Loop
         Console.WriteLine()
      End If

      startPos = 0
      endPos = 70
      If input.Substring(startPos, endPos).Contains(",") Then
         match = Regex.Match(input, pattern, RegexOptions.Multiline)
         Do While match.Success
            Console.Write("The {0} played in the {1} in", _
                              match.Groups(1).Value, match.Groups(4).Value)
            For Each capture As Capture In match.Groups(5).Captures
               Console.Write(capture.Value)
            Next
            Console.WriteLine(".")
            startPos = match.Index + match.Length
            endPos = CInt(IIf(startPos + 70 <= input.Length, 70, input.Length - startPos))
            If Not input.Substring(startPos, endPos).Contains(",") Then Exit Do
            match = match.NextMatch()
         Loop
         Console.WriteLine()
      End If


'       For Each match As Match in Regex.Matches(input, pattern, RegexOptions.Multiline)
'          Console.Write("The {0} played in the {1} in", _
'                            match.Groups(1).Value, match.Groups(4).Value)
'          For Each capture As Capture In match.Groups(5).Captures
'             Console.Write(capture.Value)
'          Next
'          Console.WriteLine(".")
'       Next
   End Sub
End Module
' The example displays the following output:
'    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
'
'    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
'    The Chicago Cubs played in the National League in 1903-present.
'    The Detroit Tigers played in the American League in 1901-present.
'    The New York Giants played in the National League in 1885-1957.
'    The Washington Senators played in the American League in 1901-1960.
```

The regular expression pattern `^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+` is defined as shown in the following table.

PATTERN	DESCRIPTION
`^`	Begin the match at the beginning of the input string (or the beginning of the line if the method is called with the RegexOptions.Multiline option).
`((\w+(\s?)){2,}`	Match one or more word characters followed either by zero or by one space exactly two times. This is the first capturing group. This expression also defines a second and third capturing group: The second consists of the captured word, and the third consists of the captured spaces.
`,\s`	Match a comma followed by a white-space character.
`(\w+\s\w+)`	Match one or more word characters followed by a space, followed by one or more word characters. This is the fourth capturing group.
`,`	Match a comma.
`\s\d{4}`	Match a space followed by four decimal digits.
`(-(\d{4}|present))?`	Match zero or one occurrence of a hyphen followed by four decimal digits or the string "present". This is the sixth capturing group. It also includes a seventh capturing group.
`,?`	Match zero or one occurrence of a comma.
`(\s\d{4}(-(\d{4}|present))?,?)+`	Match one or more occurrences of the following: a space, four decimal digits, zero or one occurrence of a hyphen followed by four decimal digits or the string "present", and zero or one comma. This is the fifth capturing group.

Back to top

End of String or Line: $

The `$` anchor specifies that the preceding pattern must occur at the end of the input string, or before `\n` at the end of the input string.

If you use `$` with the RegexOptions.Multiline option, the match can also occur at the end of a line. Note that `$` matches `\n` but does not match `\r\n` (the combination of carriage return and newline characters, or CR/LF). To match the CR/LF character combination, include `\r?$` in the regular expression pattern.

The following example adds the `$` anchor to the regular expression pattern used in the example in the Start of String or Line section. When used with the original input string, which includes five lines of text, the Regex.Matches(String, String) method is unable to find a match, because the end of the first line does not match the `$` pattern. When the original input string is split into a string array, the Regex.Matches(String, String) method succeeds in matching each of the five lines. When the Regex.Matches(String, String, RegexOptions) method is called with the `options` parameter set to RegexOptions.Multiline, no matches are found because the regular expression pattern does not account for the carriage return element (\u+000D). However, when the regular expression pattern is modified by replacing `$` with `\r?$`, calling the Regex.Matches(String, String, RegexOptions) method with the `options` parameter set to RegexOptions.Multiline again finds five matches.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      int startPos = 0, endPos = 70;
      string cr = Environment.NewLine;
      string input = "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957" + cr +
                     "Chicago Cubs, National League, 1903-present" + cr +
                     "Detroit Tigers, American League, 1901-present" + cr +
                     "New York Giants, National League, 1885-1957" + cr +
                     "Washington Senators, American League, 1901-1960" + cr;
      Match match;

      string basePattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+";
      string pattern = basePattern + "$";
      Console.WriteLine("Attempting to match the entire input string:");
      if (input.Substring(startPos, endPos).Contains(",")) {
         match = Regex.Match(input, pattern);
         while (match.Success) {
            Console.Write("The {0} played in the {1} in",
                              match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
               Console.Write(capture.Value);

            Console.WriteLine(".");
            startPos = match.Index + match.Length;
            endPos = startPos + 70 <= input.Length ? 70 : input.Length - startPos;
            if (! input.Substring(startPos, endPos).Contains(",")) break;
            match = match.NextMatch();
         }
         Console.WriteLine();
      }

      string[] teams = input.Split(new String[] { cr }, StringSplitOptions.RemoveEmptyEntries);
      Console.WriteLine("Attempting to match each element in a string array:");
      foreach (string team in teams)
      {
         if (team.Length > 70) continue;

         match = Regex.Match(team, pattern);
         if (match.Success)
         {
            Console.Write("The {0} played in the {1} in",
                          match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
               Console.Write(capture.Value);
            Console.WriteLine(".");
         }
      }
      Console.WriteLine();

      startPos = 0;
      endPos = 70;
      Console.WriteLine("Attempting to match each line of an input string with '$':");
      if (input.Substring(startPos, endPos).Contains(",")) {
         match = Regex.Match(input, pattern, RegexOptions.Multiline);
         while (match.Success) {
            Console.Write("The {0} played in the {1} in",
                              match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
               Console.Write(capture.Value);
```

```
            Console.WriteLine(".");
            startPos = match.Index + match.Length;
            endPos = startPos + 70 <= input.Length ? 70 : input.Length - startPos;
            if (! input.Substring(startPos, endPos).Contains(",")) break;
            match = match.NextMatch();
         }
         Console.WriteLine();
      }

      startPos = 0;
      endPos = 70;
      pattern = basePattern + "\r?$";
      Console.WriteLine(@"Attempting to match each line of an input string with '\r?$':");
      if (input.Substring(startPos, endPos).Contains(",")) {
         match = Regex.Match(input, pattern, RegexOptions.Multiline);
         while (match.Success) {
            Console.Write("The {0} played in the {1} in",
                              match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
               Console.Write(capture.Value);

            Console.WriteLine(".");
            startPos = match.Index + match.Length;
            endPos = startPos + 70 <= input.Length ? 70 : input.Length - startPos;
            if (! input.Substring(startPos, endPos).Contains(",")) break;
            match = match.NextMatch();
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//    Attempting to match the entire input string:
//
//    Attempting to match each element in a string array:
//    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
//    The Chicago Cubs played in the National League in 1903-present.
//    The Detroit Tigers played in the American League in 1901-present.
//    The New York Giants played in the National League in 1885-1957.
//    The Washington Senators played in the American League in 1901-1960.
//
//    Attempting to match each line of an input string with '$':
//
//    Attempting to match each line of an input string with '\r+$':
//    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
//    The Chicago Cubs played in the National League in 1903-present.
//    The Detroit Tigers played in the American League in 1901-present.
//    The New York Giants played in the National League in 1885-1957.
//    The Washington Senators played in the American League in 1901-1960.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim startPos As Integer = 0
      Dim endPos As Integer = 70
      Dim input As String = "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957" + vbCrLf + _
                            "Chicago Cubs, National League, 1903-present" + vbCrLf + _
                            "Detroit Tigers, American League, 1901-present" + vbCrLf + _
                            "New York Giants, National League, 1885-1957" + vbCrLf + _
                            "Washington Senators, American League, 1901-1960" + vbCrLf

      Dim basePattern As String = "^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+"
      Dim match As Match

      Dim pattern As String = basePattern + "$"
      Console.WriteLine("Attempting to match the entire input string:")
```

```
Console.WriteLine("Attempting to match the entire input string:")
' Provide minimal validation in the event the input is invalid.
If input.Substring(startPos, endPos).Contains(",") Then
   match = Regex.Match(input, pattern)
   Do While match.Success
      Console.Write("The {0} played in the {1} in", _
                        match.Groups(1).Value, match.Groups(4).Value)
      For Each capture As Capture In match.Groups(5).Captures
         Console.Write(capture.Value)
      Next
      Console.WriteLine(".")
      startPos = match.Index + match.Length
      endPos = CInt(IIf(startPos + 70 <= input.Length, 70, input.Length - startPos))
      If Not input.Substring(startPos, endPos).Contains(",") Then Exit Do
      match = match.NextMatch()
   Loop
   Console.WriteLine()
End If

Dim teams() As String = input.Split(New String() { vbCrLf }, StringSplitOptions.RemoveEmptyEntries)
Console.WriteLine("Attempting to match each element in a string array:")
For Each team As String In teams
   If team.Length > 70 Then Continue For
   match = Regex.Match(team, pattern)
   If match.Success Then
      Console.Write("The {0} played in the {1} in", _
                     match.Groups(1).Value, match.Groups(4).Value)
      For Each capture As Capture In match.Groups(5).Captures
         Console.Write(capture.Value)
      Next
      Console.WriteLine(".")
   End If
Next
Console.WriteLine()

startPos = 0
endPos = 70
Console.WriteLine("Attempting to match each line of an input string with '$':")
' Provide minimal validation in the event the input is invalid.
If input.Substring(startPos, endPos).Contains(",") Then
   match = Regex.Match(input, pattern, RegexOptions.Multiline)
   Do While match.Success
      Console.Write("The {0} played in the {1} in", _
                        match.Groups(1).Value, match.Groups(4).Value)
      For Each capture As Capture In match.Groups(5).Captures
         Console.Write(capture.Value)
      Next
      Console.WriteLine(".")
      startPos = match.Index + match.Length
      endPos = CInt(IIf(startPos + 70 <= input.Length, 70, input.Length - startPos))
      If Not input.Substring(startPos, endPos).Contains(",") Then Exit Do
      match = match.NextMatch()
   Loop
   Console.WriteLine()
End If


startPos = 0
endPos = 70
pattern = basePattern + "\r?$"
Console.WriteLine("Attempting to match each line of an input string with '\r?$':")
' Provide minimal validation in the event the input is invalid.
If input.Substring(startPos, endPos).Contains(",") Then
   match = Regex.Match(input, pattern, RegexOptions.Multiline)
   Do While match.Success
      Console.Write("The {0} played in the {1} in", _
                        match.Groups(1).Value, match.Groups(4).Value)
      For Each capture As Capture In match.Groups(5).Captures
         Console.Write(capture.Value)
      Next
```

```
            Next
            Console.WriteLine(".")
            startPos = match.Index + match.Length
            endPos = CInt(IIf(startPos + 70 <= input.Length, 70, input.Length - startPos))
            If Not input.Substring(startPos, endPos).Contains(",") Then Exit Do
            match = match.NextMatch()
         Loop
         Console.WriteLine()
      End If
   End Sub
End Module
' The example displays the following output:
'    Attempting to match the entire input string:
'
'    Attempting to match each element in a string array:
'    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
'    The Chicago Cubs played in the National League in 1903-present.
'    The Detroit Tigers played in the American League in 1901-present.
'    The New York Giants played in the National League in 1885-1957.
'    The Washington Senators played in the American League in 1901-1960.
'
'    Attempting to match each line of an input string with '$':
'
'    Attempting to match each line of an input string with '\r+$':
'    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
'    The Chicago Cubs played in the National League in 1903-present.
'    The Detroit Tigers played in the American League in 1901-present.
'    The New York Giants played in the National League in 1885-1957.
'    The Washington Senators played in the American League in 1901-1960.
```

Back to top

Start of String Only: \A

The `\A` anchor specifies that a match must occur at the beginning of the input string. It is identical to the `^` anchor, except that `\A` ignores the RegexOptions.Multiline option. Therefore, it can only match the start of the first line in a multiline input string.

The following example is similar to the examples for the `^` and `$` anchors. It uses the `\A` anchor in a regular expression that extracts information about the years during which some professional baseball teams existed. The input string includes five lines. The call to the Regex.Matches(String, String, RegexOptions) method finds only the first substring in the input string that matches the regular expression pattern. As the example shows, the Multiline option has no effect.

```csharp
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      int startPos = 0, endPos = 70;
      string input = "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957\n" +
                     "Chicago Cubs, National League, 1903-present\n" +
                     "Detroit Tigers, American League, 1901-present\n" +
                     "New York Giants, National League, 1885-1957\n" +
                     "Washington Senators, American League, 1901-1960\n";

      string pattern = @"\A((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+";
      Match match;

      if (input.Substring(startPos, endPos).Contains(",")) {
         match = Regex.Match(input, pattern, RegexOptions.Multiline);
         while (match.Success) {
            Console.Write("The {0} played in the {1} in",
                              match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
               Console.Write(capture.Value);

            Console.WriteLine(".");
            startPos = match.Index + match.Length;
            endPos = startPos + 70 <= input.Length ? 70 : input.Length - startPos;
            if (! input.Substring(startPos, endPos).Contains(",")) break;
            match = match.NextMatch();
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim startPos As Integer = 0
      Dim endPos As Integer = 70
      Dim input As String = "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957" + vbCrLf + _
                            "Chicago Cubs, National League, 1903-present" + vbCrLf + _
                            "Detroit Tigers, American League, 1901-present" + vbCrLf + _
                            "New York Giants, National League, 1885-1957" + vbCrLf + _
                            "Washington Senators, American League, 1901-1960" + vbCrLf

      Dim pattern As String = "\A((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+"
      Dim match As Match

      ' Provide minimal validation in the event the input is invalid.
      If input.Substring(startPos, endPos).Contains(",") Then
         match = Regex.Match(input, pattern, RegexOptions.Multiline)
         Do While match.Success
            Console.Write("The {0} played in the {1} in", _
                              match.Groups(1).Value, match.Groups(4).Value)
            For Each capture As Capture In match.Groups(5).Captures
               Console.Write(capture.Value)
            Next
            Console.WriteLine(".")
            startPos = match.Index + match.Length
            endPos = CInt(IIf(startPos + 70 <= input.Length, 70, input.Length - startPos))
            If Not input.Substring(startPos, endPos).Contains(",") Then Exit Do
            match = match.NextMatch()
         Loop
         Console.WriteLine()
      End If
   End Sub
End Module
' The example displays the following output:
'    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
```

Back to top

End of String or Before Ending Newline: \Z

The `\Z` anchor specifies that a match must occur at the end of the input string, or before `\n` at the end of the input string. It is identical to the `$` anchor, except that `\Z` ignores the RegexOptions.Multiline option. Therefore, in a multiline string, it can only match the end of the last line, or the last line before `\n`.

Note that `\Z` matches `\n` but does not match `\r\n` (the CR/LF character combination). To match CR/LF, include `\r?\Z` in the regular expression pattern.

The following example uses the `\Z` anchor in a regular expression that is similar to the example in the Start of String or Line section, which extracts information about the years during which some professional baseball teams existed. The subexpression `\r?\Z` in the regular expression `^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\Z` matches the end of a string, and also matches a string that ends with `\n` or `\r\n`. As a result, each element in the array matches the regular expression pattern.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957",
                          "Chicago Cubs, National League, 1903-present" + Environment.NewLine,
                          "Detroit Tigers, American League, 1901-present" + Regex.Unescape(@"\n"),
                          "New York Giants, National League, 1885-1957",
                          "Washington Senators, American League, 1901-1960" + Environment.NewLine};
      string pattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\Z";

      foreach (string input in inputs)
      {
         if (input.Length > 70 || ! input.Contains(",")) continue;

         Console.WriteLine(Regex.Escape(input));
         Match match = Regex.Match(input, pattern);
         if (match.Success)
            Console.WriteLine("   Match succeeded.");
         else
            Console.WriteLine("   Match failed.");
      }
   }
}
// The example displays the following output:
//    Brooklyn\ Dodgers,\ National\ League,\ 1911,\ 1912,\ 1932-1957
//       Match succeeded.
//    Chicago\ Cubs,\ National\ League,\ 1903-present\r\n
//       Match succeeded.
//    Detroit\ Tigers,\ American\ League,\ 1901-present\n
//       Match succeeded.
//    New\ York\ Giants,\ National\ League,\ 1885-1957
//       Match succeeded.
//    Washington\ Senators,\ American\ League,\ 1901-1960\r\n
//       Match succeeded.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim inputs() As String = { "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957",  _
                                 "Chicago Cubs, National League, 1903-present" + vbCrLf, _
                                 "Detroit Tigers, American League, 1901-present" + vbLf, _
                                 "New York Giants, National League, 1885-1957", _
                                 "Washington Senators, American League, 1901-1960" + vbCrLf }
      Dim pattern As String = "^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\Z"

      For Each input As String In inputs
         If input.Length > 70 Or Not input.Contains(",") Then Continue For

         Console.WriteLine(Regex.Escape(input))
         Dim match As Match = Regex.Match(input, pattern)
         If match.Success Then
            Console.WriteLine("   Match succeeded.")
         Else
            Console.WriteLine("   Match failed.")
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'    Brooklyn\ Dodgers,\ National\ League,\ 1911,\ 1912,\ 1932-1957
'       Match succeeded.
'    Chicago\ Cubs,\ National\ League,\ 1903-present\r\n
'       Match succeeded.
'    Detroit\ Tigers,\ American\ League,\ 1901-present\n
'       Match succeeded.
'    New\ York\ Giants,\ National\ League,\ 1885-1957
'       Match succeeded.
'    Washington\ Senators,\ American\ League,\ 1901-1960\r\n
'       Match succeeded.
```

Back to top

End of String Only: \z

The `\z` anchor specifies that a match must occur at the end of the input string. Like the `$` language element, `\z` ignores the RegexOptions.Multiline option. Unlike the `\Z` language element, `\z` does not match a `\n` character at the end of a string. Therefore, it can only match the last line of the input string.

The following example uses the `\z` anchor in a regular expression that is otherwise identical to the example in the previous section, which extracts information about the years during which some professional baseball teams existed. The example tries to match each of five elements in a string array with the regular expression pattern `^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\z`. Two of the strings end with carriage return and line feed characters, one ends with a line feed character, and two end with neither a carriage return nor a line feed character. As the output shows, only the strings without a carriage return or line feed character match the pattern.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957",
                          "Chicago Cubs, National League, 1903-present" + Environment.NewLine,
                          "Detroit Tigers, American League, 1901-present\\r",
                          "New York Giants, National League, 1885-1957",
                          "Washington Senators, American League, 1901-1960" + Environment.NewLine };
      string pattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\z";

      foreach (string input in inputs)
      {
         if (input.Length > 70 || ! input.Contains(",")) continue;

         Console.WriteLine(Regex.Escape(input));
         Match match = Regex.Match(input, pattern);
         if (match.Success)
            Console.WriteLine("   Match succeeded.");
         else
            Console.WriteLine("   Match failed.");
      }
   }
}
// The example displays the following output:
//    Brooklyn\ Dodgers,\ National\ League,\ 1911,\ 1912,\ 1932-1957
//       Match succeeded.
//    Chicago\ Cubs,\ National\ League,\ 1903-present\r\n
//       Match failed.
//    Detroit\ Tigers,\ American\ League,\ 1901-present\n
//       Match failed.
//    New\ York\ Giants,\ National\ League,\ 1885-1957
//       Match succeeded.
//    Washington\ Senators,\ American\ League,\ 1901-1960\r\n
//       Match failed.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim inputs() As String = { "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957",  _
                              "Chicago Cubs, National League, 1903-present" + vbCrLf, _
                              "Detroit Tigers, American League, 1901-present" + vbLf, _
                              "New York Giants, National League, 1885-1957", _
                              "Washington Senators, American League, 1901-1960" + vbCrLf }
      Dim pattern As String = "^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\z"

      For Each input As String In inputs
         If input.Length > 70 Or Not input.Contains(",") Then Continue For

         Console.WriteLine(Regex.Escape(input))
         Dim match As Match = Regex.Match(input, pattern)
         If match.Success Then
            Console.WriteLine("   Match succeeded.")
         Else
            Console.WriteLine("   Match failed.")
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'    Brooklyn\ Dodgers,\ National\ League,\ 1911,\ 1912,\ 1932-1957
'       Match succeeded.
'    Chicago\ Cubs,\ National\ League,\ 1903-present\r\n
'       Match failed.
'    Detroit\ Tigers,\ American\ League,\ 1901-present\n
'       Match failed.
'    New\ York\ Giants,\ National\ League,\ 1885-1957
'       Match succeeded.
'    Washington\ Senators,\ American\ League,\ 1901-1960\r\n
'       Match failed.
```

Back to top

Contiguous Matches: \G

The `\G` anchor specifies that a match must occur at the point where the previous match ended. When you use this anchor with the Regex.Matches or Match.NextMatch method, it ensures that all matches are contiguous.

The following example uses a regular expression to extract the names of rodent species from a comma-delimited string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "capybara,squirrel,chipmunk,porcupine,gopher," +
                     "beaver,groundhog,hamster,guinea pig,gerbil," +
                     "chinchilla,prairie dog,mouse,rat";
      string pattern = @"\G(\w+\s?\w*),?";
      Match match = Regex.Match(input, pattern);
      while (match.Success)
      {
         Console.WriteLine(match.Groups[1].Value);
         match = match.NextMatch();
      }
   }
}
// The example displays the following output:
//       capybara
//       squirrel
//       chipmunk
//       porcupine
//       gopher
//       beaver
//       groundhog
//       hamster
//       guinea pig
//       gerbil
//       chinchilla
//       prairie dog
//       mouse
//       rat
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "capybara,squirrel,chipmunk,porcupine,gopher," + _
                            "beaver,groundhog,hamster,guinea pig,gerbil," + _
                            "chinchilla,prairie dog,mouse,rat"
      Dim pattern As String = "\G(\w+\s?\w*),?"
      Dim match As Match = Regex.Match(input, pattern)
      Do While match.Success
         Console.WriteLine(match.Groups(1).Value)
         match = match.NextMatch()
      Loop
   End Sub
End Module
' The example displays the following output:
'       capybara
'       squirrel
'       chipmunk
'       porcupine
'       gopher
'       beaver
'       groundhog
'       hamster
'       guinea pig
'       gerbil
'       chinchilla
'       prairie dog
'       mouse
'       rat
```

The regular expression `\G(\w+\s?\w*),?` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\G`	Begin where the last match ended.
`\w+`	Match one or more word characters.
`\s?`	Match zero or one space.
`\w*`	Match zero or more word characters.
`(\w+\s?\w*)`	Match one or more word characters followed by zero or one space, followed by zero or more word characters. This is the first capturing group.
`,?`	Match zero or one occurrence of a literal comma character.

Back to top

Word Boundary: \b

The `\b` anchor specifies that the match must occur on a boundary between a word character (the `\w` language element) and a non-word character (the `\W` language element). Word characters consist of alphanumeric characters and underscores; a non-word character is any character that is not alphanumeric or an underscore. (For more information, see Character Classes.) The match may also occur on a word boundary at the beginning or end of the string.

The `\b` anchor is frequently used to ensure that a subexpression matches an entire word instead of just the beginning or end of a word. The regular expression `\bare\w*\b` in the following example illustrates this usage. It matches any word that begins with the substring "are". The output from the example also illustrates that `\b` matches both the beginning and the end of the input string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "area bare arena mare";
      string pattern = @"\bare\w*\b";
      Console.WriteLine("Words that begin with 'are':");
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("'{0}' found at position {1}",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       Words that begin with 'are':
//       'area' found at position 0
//       'arena' found at position 10
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "area bare arena mare"
      Dim pattern As String = "\bare\w*\b"
      Console.WriteLine("Words that begin with 'are':")
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("'{0}' found at position {1}", _
                           match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       Words that begin with 'are':
'       'area' found at position 0
'       'arena' found at position 10
```

The regular expression pattern is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`are`	Match the substring "are".
`\w*`	Match zero or more word characters.
`\b`	End the match at a word boundary.

Back to top

Non-Word Boundary: \B

The `\B` anchor specifies that the match must not occur on a word boundary. It is the opposite of the `\b` anchor.

The following example uses the `\B` anchor to locate occurrences of the substring "qu" in a word. The regular expression pattern `\Bqu\w+` matches a substring that begins with a "qu" that does not start a word and that continues to the end of the word.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "equity queen equip acquaint quiet";
      string pattern = @"\Bqu\w+";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("'{0}' found at position {1}",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       'quity' found at position 1
//       'quip' found at position 14
//       'quaint' found at position 21
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "equity queen equip acquaint quiet"
      Dim pattern As String = "\Bqu\w+"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("'{0}' found at position {1}", _
                           match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       'quity' found at position 1
'       'quip' found at position 14
'       'quaint' found at position 21
```

The regular expression pattern is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\B`	Do not begin the match at a word boundary.
`qu`	Match the substring "qu".
`\w+`	Match one or more word characters.

See also

- Regular Expression Language - Quick Reference
- Regular Expression Options

Grouping Constructs in Regular Expressions

9/6/2018 • 37 minutes to read • Edit Online

Grouping constructs delineate the subexpressions of a regular expression and capture the substrings of an input string. You can use grouping constructs to do the following:

- Match a subexpression that is repeated in the input string.
- Apply a quantifier to a subexpression that has multiple regular expression language elements. For more information about quantifiers, see Quantifiers.
- Include a subexpression in the string that is returned by the Regex.Replace and Match.Result methods.
- Retrieve individual subexpressions from the Match.Groups property and process them separately from the matched text as a whole.

The following table lists the grouping constructs supported by the .NET regular expression engine and indicates whether they are capturing or non-capturing.

GROUPING CONSTRUCT	CAPTURING OR NONCAPTURING
Matched subexpressions	Capturing
Named matched subexpressions	Capturing
Balancing group definitions	Capturing
Noncapturing groups	Noncapturing
Group options	Noncapturing
Zero-width positive lookahead assertions	Noncapturing
Zero-width negative lookahead assertions	Noncapturing
Zero-width positive lookbehind assertions	Noncapturing
Zero-width negative lookbehind assertions	Noncapturing
Nonbacktracking subexpressions	Noncapturing

For information on groups and the regular expression object model, see Grouping constructs and regular expression objects.

Matched Subexpressions

The following grouping construct captures a matched subexpression:

`(` *subexpression* `)`

where *subexpression* is any valid regular expression pattern. Captures that use parentheses are numbered automatically from left to right based on the order of the opening parentheses in the regular expression, starting from one. The capture that is numbered zero is the text matched by the entire regular expression pattern.

> **NOTE**
>
> By default, the `(` *subexpression* `)` language element captures the matched subexpression. But if the RegexOptions parameter of a regular expression pattern matching method includes the RegexOptions.ExplicitCapture flag, or if the `n` option is applied to this subexpression (see Group options later in this topic), the matched subexpression is not captured.

You can access captured groups in four ways:

- By using the backreference construct within the regular expression. The matched subexpression is referenced in the same regular expression by using the syntax `\` *number*, where *number* is the ordinal number of the captured subexpression.

- By using the named backreference construct within the regular expression. The matched subexpression is referenced in the same regular expression by using the syntax `\k<` *name* `>`, where *name* is the name of a capturing group, or `\k<` *number* `>`, where *number* is the ordinal number of a capturing group. A capturing group has a default name that is identical to its ordinal number. For more information, see Named matched subexpressions later in this topic.

- By using the `$` *number* replacement sequence in a Regex.Replace or Match.Result method call, where *number* is the ordinal number of the captured subexpression.

- Programmatically, by using the GroupCollection object returned by the Match.Groups property. The member at position zero in the collection represents the entire regular expression match. Each subsequent member represents a matched subexpression. For more information, see the Grouping Constructs and Regular Expression Objects section.

The following example illustrates a regular expression that identifies duplicated words in text. The regular expression pattern's two capturing groups represent the two instances of the duplicated word. The second instance is captured to report its starting position in the input string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\w+)\s(\1)";
      string input = "He said that that was the the correct answer.";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine("Duplicate '{0}' found at positions {1} and {2}.",
                           match.Groups[1].Value, match.Groups[1].Index, match.Groups[2].Index);
   }
}
// The example displays the following output:
//       Duplicate 'that' found at positions 8 and 13.
//       Duplicate 'the' found at positions 22 and 26.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(\w+)\s(\1)\W"
      Dim input As String = "He said that that was the the correct answer."
      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
         Console.WriteLine("Duplicate '{0}' found at positions {1} and {2}.", _
                           match.Groups(1).Value, match.Groups(1).Index, match.Groups(2).Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       Duplicate 'that' found at positions 8 and 13.
'       Duplicate 'the' found at positions 22 and 26.
```

The regular expression pattern is the following:

```
(\w+)\s(\1)\W
```

The following table shows how the regular expression pattern is interpreted.

PATTERN	DESCRIPTION
`(\w+)`	Match one or more word characters. This is the first capturing group.
`\s`	Match a white-space character.
`(\1)`	Match the string in the first captured group. This is the second capturing group. The example assigns it to a captured group so that the starting position of the duplicate word can be retrieved from the `Match.Index` property.
`\W`	Match a non-word character, including white space and punctuation. This prevents the regular expression pattern from matching a word that starts with the word from the first captured group.

Named Matched Subexpressions

The following grouping construct captures a matched subexpression and lets you access it by name or by number:

```
(?<name>subexpression)
```

or:

```
(?'name'subexpression)
```

where *name* is a valid group name, and *subexpression* is any valid regular expression pattern. *name* must not contain any punctuation characters and cannot begin with a number.

> **NOTE**
>
> If the RegexOptions parameter of a regular expression pattern matching method includes the RegexOptions.ExplicitCapture flag, or if the `n` option is applied to this subexpression (see Group options later in this topic), the only way to capture a subexpression is to explicitly name capturing groups.

You can access named captured groups in the following ways:

- By using the named backreference construct within the regular expression. The matched subexpression is referenced in the same regular expression by using the syntax `\k<`*name*`>`, where *name* is the name of the captured subexpression.
- By using the backreference construct within the regular expression. The matched subexpression is referenced in the same regular expression by using the syntax `\`*number*, where *number* is the ordinal number of the captured subexpression. Named matched subexpressions are numbered consecutively from left to right after matched subexpressions.
- By using the `${`*name*`}` replacement sequence in a Regex.Replace or Match.Result method call, where *name* is the name of the captured subexpression.
- By using the `$`*number* replacement sequence in a Regex.Replace or Match.Result method call, where *number* is the ordinal number of the captured subexpression.
- Programmatically, by using the GroupCollection object returned by the Match.Groups property. The member at position zero in the collection represents the entire regular expression match. Each subsequent member represents a matched subexpression. Named captured groups are stored in the collection after numbered captured groups.
- Programmatically, by providing the subexpression name to the GroupCollection object's indexer (in C#) or to its Item[String] property (in Visual Basic).

A simple regular expression pattern illustrates how numbered (unnamed) and named groups can be referenced either programmatically or by using regular expression language syntax. The regular expression `((?<One>abc)\d+)?(?<Two>xyz)(.*)` produces the following capturing groups by number and by name. The first capturing group (number 0) always refers to the entire pattern.

NUMBER	NAME	PATTERN
0	0 (default name)	`((?<One>abc)\d+)?(?<Two>xyz)(.*)`
1	1 (default name)	`((?<One>abc)\d+)`
2	2 (default name)	`(.*)`
3	One	`(?<One>abc)`
4	Two	`(?<Two>xyz)`

The following example illustrates a regular expression that identifies duplicated words and the word that immediately follows each duplicated word. The regular expression pattern defines two named subexpressions: `duplicateWord`, which represents the duplicated word; and `nextWord`, which represents the word that follows the duplicated word.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?<duplicateWord>\w+)\s\k<duplicateWord>\W(?<nextWord>\w+)";
      string input = "He said that that was the the correct answer.";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine("A duplicate '{0}' at position {1} is followed by '{2}'.",
                           match.Groups["duplicateWord"].Value, match.Groups["duplicateWord"].Index,
                           match.Groups["nextWord"].Value);

   }
}
// The example displays the following output:
//       A duplicate 'that' at position 8 is followed by 'was'.
//       A duplicate 'the' at position 22 is followed by 'correct'.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(?<duplicateWord>\w+)\s\k<duplicateWord>\W(?<nextWord>\w+)"
      Dim input As String = "He said that that was the the correct answer."
      Console.WriteLine(Regex.Matches(input, pattern, RegexOptions.IgnoreCase).Count)
      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
         Console.WriteLine("A duplicate '{0}' at position {1} is followed by '{2}'.", _
                           match.Groups("duplicateWord").Value, match.Groups("duplicateWord").Index, _
                           match.Groups("nextWord").Value)
      Next
   End Sub
End Module
' The example displays the following output:
'    A duplicate 'that' at position 8 is followed by 'was'.
'    A duplicate 'the' at position 22 is followed by 'correct'.
```

The regular expression pattern is as follows:

```
(?<duplicateWord>\w+)\s\k<duplicateWord>\W(?<nextWord>\w+)
```

The following table shows how the regular expression is interpreted.

PATTERN	DESCRIPTION
`(?<duplicateWord>\w+)`	Match one or more word characters. Name this capturing group `duplicateWord`.
`\s`	Match a white-space character.
`\k<duplicateWord>`	Match the string from the captured group that is named `duplicateWord`.
`\W`	Match a non-word character, including white space and punctuation. This prevents the regular expression pattern from matching a word that starts with the word from the first captured group.

PATTERN	DESCRIPTION
`(?<nextWord>\w+)`	Match one or more word characters. Name this capturing group `nextWord`.

Note that a group name can be repeated in a regular expression. For example, it is possible for more than one group to be named `digit`, as the following example illustrates. In the case of duplicate names, the value of the Group object is determined by the last successful capture in the input string. In addition, the CaptureCollection is populated with information about each capture just as it would be if the group name was not duplicated.

In the following example, the regular expression `\D+(?<digit>\d+)\D+(?<digit>\d+)?` includes two occurrences of a group named `digit`. The first `digit` named group captures one or more digit characters. The second `digit` named group captures either zero or one occurrence of one or more digit characters. As the output from the example shows, if the second capturing group successfully matches text, the value of that text defines the value of the Group object. If the second capturing group cannot does not match the input string, the value of the last successful match defines the value of the Group object.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      String pattern = @"\D+(?<digit>\d+)\D+(?<digit>\d+)?";
      String[] inputs = { "abc123def456", "abc123def" };
      foreach (var input in inputs) {
         Match m = Regex.Match(input, pattern);
         if (m.Success) {
            Console.WriteLine("Match: {0}", m.Value);
            for (int grpCtr = 1; grpCtr < m.Groups.Count; grpCtr++) {
               Group grp = m.Groups[grpCtr];
               Console.WriteLine("Group {0}: {1}", grpCtr, grp.Value);
               for (int capCtr = 0; capCtr < grp.Captures.Count; capCtr++)
                  Console.WriteLine("   Capture {0}: {1}", capCtr,
                                    grp.Captures[capCtr].Value);
            }
         }
         else {
            Console.WriteLine("The match failed.");
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Match: abc123def456
//       Group 1: 456
//          Capture 0: 123
//          Capture 1: 456
//
//       Match: abc123def
//       Group 1: 123
//          Capture 0: 123
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\D+(?<digit>\d+)\D+(?<digit>\d+)?"
      Dim inputs() As String = { "abc123def456", "abc123def" }
      For Each input As String In inputs
         Dim m As Match = Regex.Match(input, pattern)
         If m.Success Then
            Console.WriteLine("Match: {0}", m.Value)
            For grpCtr As Integer = 1 to m.Groups.Count - 1
               Dim grp As Group = m.Groups(grpCtr)
               Console.WriteLine("Group {0}: {1}", grpCtr, grp.Value)
               For capCtr As Integer = 0 To grp.Captures.Count - 1
                  Console.WriteLine("   Capture {0}: {1}", capCtr,
                                    grp.Captures(capCtr).Value)
               Next
            Next
         Else
            Console.WriteLine("The match failed.")
         End If
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays the following output:
'       Match: abc123def456
'       Group 1: 456
'          Capture 0: 123
'          Capture 1: 456
'
'       Match: abc123def
'       Group 1: 123
'          Capture 0: 123
```

The following table shows how the regular expression is interpreted.

PATTERN	DESCRIPTION
`\D+`	Match one or more non-decimal digit characters.
`(?<digit>\d+)`	Match one or more decimal digit characters. Assign the match to the `digit` named group.
`\D+`	Match one or more non-decimal digit characters.
`(?<digit>\d+)?`	Match zero or one occurrence of one or more decimal digit characters. Assign the match to the `digit` named group.

Balancing Group Definitions

A balancing group definition deletes the definition of a previously defined group and stores, in the current group, the interval between the previously defined group and the current group. This grouping construct has the following format:

```
(?<name1-name2>subexpression)
```

or:

```
(?'name1-name2' subexpression)
```

where *name1* is the current group (optional), *name2* is a previously defined group, and *subexpression* is any valid regular expression pattern. The balancing group definition deletes the definition of *name2* and stores the interval between *name2* and *name1* in *name1*. If no *name2* group is defined, the match backtracks. Because deleting the last definition of *name2* reveals the previous definition of *name2*, this construct lets you use the stack of captures for group *name2* as a counter for keeping track of nested constructs such as parentheses or opening and closing brackets.

The balancing group definition uses *name2* as a stack. The beginning character of each nested construct is placed in the group and in its Group.Captures collection. When the closing character is matched, its corresponding opening character is removed from the group, and the Captures collection is decreased by one. After the opening and closing characters of all nested constructs have been matched, *name1* is empty.

> **NOTE**
>
> After you modify the regular expression in the following example to use the appropriate opening and closing character of a nested construct, you can use it to handle most nested constructs, such as mathematical expressions or lines of program code that include multiple nested method calls.

The following example uses a balancing group definition to match left and right angle brackets (<>) in an input string. The example defines two named groups, `Open` and `Close`, that are used like a stack to track matching pairs of angle brackets. Each captured left angle bracket is pushed into the capture collection of the `Open` group, and each captured right angle bracket is pushed into the capture collection of the `Close` group. The balancing group definition ensures that there is a matching right angle bracket for each left angle bracket. If there is not, the final subpattern, `(?(Open)(?!))`, is evaluated only if the `Open` group is not empty (and, therefore, if all nested constructs have not been closed). If the final subpattern is evaluated, the match fails, because the `(?!)` subpattern is a zero-width negative lookahead assertion that always fails.

```
using System;
using System.Text.RegularExpressions;

class Example
{
   public static void Main()
   {
      string pattern = "^[^<>]*" +
                       "(" +
                       "((?'Open'<)[^<>]*)+" +
                       "((?'Close-Open'>)[^<>]*)+" +
                       ")*" +
                       "(?(Open)(?!))$";
      string input = "<abc><mno<xyz>>";

      Match m = Regex.Match(input, pattern);
      if (m.Success == true)
      {
         Console.WriteLine("Input: \"{0}\" \nMatch: \"{1}\"", input, m);
         int grpCtr = 0;
         foreach (Group grp in m.Groups)
         {
            Console.WriteLine("   Group {0}: {1}", grpCtr, grp.Value);
            grpCtr++;
            int capCtr = 0;
            foreach (Capture cap in grp.Captures)
            {
                Console.WriteLine("      Capture {0}: {1}", capCtr, cap.Value);
                capCtr++;
            }
          }
      }
      else
      {
         Console.WriteLine("Match failed.");
      }
    }
}
// The example displays the following output:
//    Input: "<abc><mno<xyz>>"
//    Match: "<abc><mno<xyz>>"
//       Group 0: <abc><mno<xyz>>
//          Capture 0: <abc><mno<xyz>>
//       Group 1: <mno<xyz>>
//          Capture 0: <abc>
//          Capture 1: <mno<xyz>>
//       Group 2: <xyz
//          Capture 0: <abc
//          Capture 1: <mno
//          Capture 2: <xyz
//       Group 3: >
//          Capture 0: >
//          Capture 1: >
//          Capture 2: >
//       Group 4:
//       Group 5: mno<xyz>
//          Capture 0: abc
//          Capture 1: xyz
//          Capture 2: mno<xyz>
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
        Dim pattern As String = "^[^<>]*" & _
                                "(" + "((?'Open'<)[^<>]*)+" & _
                                "((?'Close-Open'>)[^<>]*)+" + ")*" & _
                                "(?(Open)(?!))$"
        Dim input As String = "<abc><mno<xyz>>"
        Dim rgx AS New Regex(pattern)'
        Dim m As Match = Regex.Match(input, pattern)
        If m.Success Then
            Console.WriteLine("Input: ""{0}"" " & vbCrLf & "Match: ""{1}""", _
                               input, m)
            Dim grpCtr As Integer = 0
            For Each grp As Group In m.Groups
               Console.WriteLine("   Group {0}: {1}", grpCtr, grp.Value)
               grpCtr += 1
               Dim capCtr As Integer = 0
               For Each cap As Capture In grp.Captures
                  Console.WriteLine("      Capture {0}: {1}", capCtr, cap.Value)
                  capCtr += 1
               Next
            Next
        Else
            Console.WriteLine("Match failed.")
        End If
    End Sub
End Module
' The example displays the following output:
'       Input: "<abc><mno<xyz>>"
'       Match: "<abc><mno<xyz>>"
'          Group 0: <abc><mno<xyz>>
'             Capture 0: <abc><mno<xyz>>
'          Group 1: <mno<xyz>>
'             Capture 0: <abc>
'             Capture 1: <mno<xyz>>
'          Group 2: <xyz
'             Capture 0: <abc
'             Capture 1: <mno
'             Capture 2: <xyz
'          Group 3: >
'             Capture 0: >
'             Capture 1: >
'             Capture 2: >
'          Group 4:
'          Group 5: mno<xyz>
'             Capture 0: abc
'             Capture 1: xyz
'             Capture 2: mno<xyz>
```

The regular expression pattern is:

```
^[^<>]*(((?'Open'<)[^<>]*)+((?'Close-Open'>)[^<>]*)+)*(?(Open)(?!))$
```

The regular expression is interpreted as follows:

PATTERN	DESCRIPTION
`^`	Begin at the start of the string.

PATTERN	DESCRIPTION
`[^<>]*`	Match zero or more characters that are not left or right angle brackets.
`(?'Open'<)`	Match a left angle bracket and assign it to a group named `Open`.
`[^<>]*`	Match zero or more characters that are not left or right angle brackets.
`((?'Open'<)[^<>]*)+`	Match one or more occurrences of a left angle bracket followed by zero or more characters that are not left or right angle brackets. This is the second capturing group.
`(?'Close-Open'>)`	Match a right angle bracket, assign the substring between the `Open` group and the current group to the `Close` group, and delete the definition of the `Open` group.
`[^<>]*`	Match zero or more occurrences of any character that is neither a left nor a right angle bracket.
`((?'Close-Open'>)[^<>]*)+`	Match one or more occurrences of a right angle bracket, followed by zero or more occurrences of any character that is neither a left nor a right angle bracket. When matching the right angle bracket, assign the substring between the `Open` group and the current group to the `Close` group, and delete the definition of the `Open` group. This is the third capturing group.
`(((?'Open'<)[^<>]*)+((?'Close-Open'>)[^<>]*)+)*`	Match zero or more occurrences of the following pattern: one or more occurrences of a left angle bracket, followed by zero or more non-angle bracket characters, followed by one or more occurrences of a right angle bracket, followed by zero or more occurrences of non-angle brackets. When matching the right angle bracket, delete the definition of the `Open` group, and assign the substring between the `Open` group and the current group to the `Close` group. This is the first capturing group.
`(?(Open)(?!))`	If the `Open` group exists, abandon the match if an empty string can be matched, but do not advance the position of the regular expression engine in the string. This is a zero-width negative lookahead assertion. Because an empty string is always implicitly present in an input string, this match always fails. Failure of this match indicates that the angle brackets are not balanced.
`$`	Match the end of the input string.

The final subexpression, `(?(Open)(?!))`, indicates whether the nesting constructs in the input string are properly balanced (for example, whether each left angle bracket is matched by a right angle bracket). It uses conditional matching based on a valid captured group; for more information, see Alternation Constructs. If the `Open` group is defined, the regular expression engine attempts to match the subexpression `(?!)` in the input string. The `Open` group should be defined only if nesting constructs are unbalanced. Therefore, the pattern to be matched in the input string should be one that always causes the match to fail. In this case, `(?!)` is a zero-width negative lookahead assertion that always fails, because an empty string is always implicitly present at the next position in the input string.

In the example, the regular expression engine evaluates the input string "<abc><mno<xyz>>" as shown in the following table.

STEP	PATTERN	RESULT
1	`^`	Starts the match at the beginning of the input string
2	`[^<>]*`	Looks for non-angle bracket characters before the left angle bracket;finds no matches.
3	`(((?'Open'<)`	Matches the left angle bracket in "<abc>" and assigns it to the `Open` group.
4	`[^<>]*`	Matches "abc".
5	`)+`	"<abc" is the value of the second captured group. The next character in the input string is not a left angle bracket, so the regular expression engine does not loop back to the `(?'Open'<)[^<>]*)` subpattern.
6	`((?'Close-Open'>)`	Matches the right angle bracket in "<abc>", assigns "abc", which is the substring between the `Open` group and the right angle bracket, to the `Close` group, and deletes the current value ("<") of the `Open` group, leaving it empty.
7	`[^<>]*`	Looks for non-angle bracket characters after the right angle bracket; finds no matches.
8	`)+`	The value of the third captured group is ">". The next character in the input string is not a right angle bracket, so the regular expression engine does not loop back to the `((?'Close-Open'>)[^<>]*)` subpattern.
9	`)*`	The value of the first captured group is "<abc>". The next character in the input string is a left angle bracket, so the regular expression engine loops back to the `(((?'Open'<)` subpattern.

STEP	PATTERN	RESULT
10	`(((?'Open'<)`	Matches the left angle bracket in "<mno>" and assigns it to the `Open` group. Its Group.Captures collection now has a single value, "<".
11	`[^<>]*`	Matches "mno".
12	`)+`	"<mno" is the value of the second captured group. The next character in the input string is an left angle bracket, so the regular expression engine loops back to the `(?'Open'<)[^<>]*)` subpattern.
13	`(((?'Open'<)`	Matches the left angle bracket in "<xyz>" and assigns it to the `Open` group. The Group.Captures collection of the `Open` group now includes two captures: the left angle bracket from "<mno>", and the left angle bracket from "<xyz>".
14	`[^<>]*`	Matches "xyz".
15	`)+`	"<xyz" is the value of the second captured group. The next character in the input string is not a left angle bracket, so the regular expression engine does not loop back to the `(?'Open'<)[^<>]*)` subpattern.
16	`((?'Close-Open'>)`	Matches the right angle bracket in "<xyz>". "xyz", assigns the substring between the `Open` group and the right angle bracket to the `Close` group, and deletes the current value of the `Open` group. The value of the previous capture (the left angle bracket in "<mno>") becomes the current value of the `Open` group. The Captures collection of the `Open` group now includes a single capture, the left angle bracket from "<xyz>".
17	`[^<>]*`	Looks for non-angle bracket characters; finds no matches.

STEP	PATTERN	RESULT
18	`)+`	The value of the third captured group is ">". The next character in the input string is a right angle bracket, so the regular expression engine loops back to the `((?'Close-Open'>)[^<>]*)` subpattern.
19	`((?'Close-Open'>)`	Matches the final right angle bracket in "xyz>>", assigns "mno<xyz>" (the substring between the `Open` group and the right angle bracket) to the `Close` group, and deletes the current value of the `Open` group. The `Open` group is now empty.
20	`[^<>]*`	Looks for non-angle bracket characters; finds no matches.
21	`)+`	The value of the third captured group is ">". The next character in the input string is not a right angle bracket, so the regular expression engine does not loop back to the `((?'Close-Open'>)[^<>]*)` subpattern.
22	`)*`	The value of the first captured group is "<mno<xyz>>". The next character in the input string is not a left angle bracket, so the regular expression engine does not loop back to the `(((?'Open'<)` subpattern.
23	`(?(Open)(?!))`	The `Open` group is not defined, so no match is attempted.
24	`$`	Matches the end of the input string.

Noncapturing Groups

The following grouping construct does not capture the substring that is matched by a subexpression:

```
(?:subexpression)
```

where *subexpression* is any valid regular expression pattern. The noncapturing group construct is typically used when a quantifier is applied to a group, but the substrings captured by the group are of no interest.

> **NOTE**
>
> If a regular expression includes nested grouping constructs, an outer noncapturing group construct does not apply to the inner nested group constructs.

The following example illustrates a regular expression that includes noncapturing groups. Note that the output does not include any captured groups.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?:\b(?:\w+)\W*)+\.";
      string input = "This is a short sentence.";
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match: {0}", match.Value);
      for (int ctr = 1; ctr < match.Groups.Count; ctr++)
         Console.WriteLine("   Group {0}: {1}", ctr, match.Groups[ctr].Value);
   }
}
// The example displays the following output:
//       Match: This is a short sentence.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(?:\b(?:\w+)\W*)+\."
      Dim input As String = "This is a short sentence."
      Dim match As Match = Regex.Match(input, pattern)
      Console.WriteLine("Match: {0}", match.Value)
      For ctr As Integer = 1 To match.Groups.Count - 1
         Console.WriteLine("   Group {0}: {1}", ctr, match.Groups(ctr).Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       Match: This is a short sentence.
```

The regular expression `(?:\b(?:\w+)\W*)+\.` matches a sentence that is terminated by a period. Because the regular expression focuses on sentences and not on individual words, grouping constructs are used exclusively as quantifiers. The regular expression pattern is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`(?:\w+)`	Match one or more word characters. Do not assign the matched text to a captured group.
`\W*`	Match zero or more non-word characters.

PATTERN	DESCRIPTION
`(?:\b(?:\w+)\W*)+`	Match the pattern of one or more word characters starting at a word boundary, followed by zero or more non-word characters, one or more times. Do not assign the matched text to a captured group.
`\.`	Match a period.

Group Options

The following grouping construct applies or disables the specified options within a subexpression:

`(?imnsx-imnsx:` *subexpression* `)`

where *subexpression* is any valid regular expression pattern. For example, `(?i-s:)` turns on case insensitivity and disables single-line mode. For more information about the inline options you can specify, see Regular Expression Options.

> **NOTE**
> You can specify options that apply to an entire regular expression rather than a subexpression by using a System.Text.RegularExpressions.Regex class constructor or a static method. You can also specify inline options that apply after a specific point in a regular expression by using the `(?imnsx-imnsx)` language construct.

The group options construct is not a capturing group. That is, although any portion of a string that is captured by *subexpression* is included in the match, it is not included in a captured group nor used to populate the GroupCollection object.

For example, the regular expression `\b(?ix: d \w+)\s` in the following example uses inline options in a grouping construct to enable case-insensitive matching and ignore pattern white space in identifying all words that begin with the letter "d". The regular expression is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`(?ix: d \w+)`	Using case-insensitive matching and ignoring white space in this pattern, match a "d" followed by one or more word characters.
`\s`	Match a white-space character.

```
string pattern = @"\b(?ix: d \w+)\s";
string input = "Dogs are decidedly good pets.";

foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}// found at index {1}.", match.Value, match.Index);
// The example displays the following output:
//    'Dogs // found at index 0.
//    'decidedly // found at index 9.
```

```
Dim pattern As String = "\b(?ix: d \w+)\s"
Dim input As String = "Dogs are decidedly good pets."

For Each match As Match In Regex.Matches(input, pattern)
   Console.WriteLine("'{0}' found at index {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'    'Dogs ' found at index 0.
'    'decidedly ' found at index 9.
```

Zero-Width Positive Lookahead Assertions

The following grouping construct defines a zero-width positive lookahead assertion:

`(?=` *subexpression* `)`

where *subexpression* is any regular expression pattern. For a match to be successful, the input string must match the regular expression pattern in *subexpression*, although the matched substring is not included in the match result. A zero-width positive lookahead assertion does not backtrack.

Typically, a zero-width positive lookahead assertion is found at the end of a regular expression pattern. It defines a substring that must be found at the end of a string for a match to occur but that should not be included in the match. It is also useful for preventing excessive backtracking. You can use a zero-width positive lookahead assertion to ensure that a particular captured group begins with text that matches a subset of the pattern defined for that captured group. For example, if a capturing group matches consecutive word characters, you can use a zero-width positive lookahead assertion to require that the first character be an alphabetical uppercase character.

The following example uses a zero-width positive lookahead assertion to match the word that precedes the verb "is" in the input string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\w+(?=\sis\b)";
      string[] inputs = { "The dog is a Malamute.",
                          "The island has beautiful birds.",
                          "The pitch missed home plate.",
                          "Sunday is a weekend day." };

      foreach (string input in inputs)
      {
         Match match = Regex.Match(input, pattern);
         if (match.Success)
            Console.WriteLine("'{0}' precedes 'is'.", match.Value);
         else
            Console.WriteLine("'{0}' does not match the pattern.", input);
      }
   }
}
// The example displays the following output:
//    'dog' precedes 'is'.
//    'The island has beautiful birds.' does not match the pattern.
//    'The pitch missed home plate.' does not match the pattern.
//    'Sunday' precedes 'is'.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b\w+(?=\sis\b)"
      Dim inputs() As String = { "The dog is a Malamute.", _
                                 "The island has beautiful birds.", _
                                 "The pitch missed home plate.", _
                                 "Sunday is a weekend day." }

      For Each input As String In inputs
         Dim match As Match = Regex.Match(input, pattern)
         If match.Success Then
            Console.WriteLine("'{0}' precedes 'is'.", match.Value)
         Else
            Console.WriteLine("'{0}' does not match the pattern.", input)
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'       'dog' precedes 'is'.
'       'The island has beautiful birds.' does not match the pattern.
'       'The pitch missed home plate.' does not match the pattern.
'       'Sunday' precedes 'is'.
```

The regular expression `\b\w+(?=\sis\b)` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`\w+`	Match one or more word characters.
`(?=\sis\b)`	Determine whether the word characters are followed by a white-space character and the string "is", which ends on a word boundary. If so, the match is successful.

Zero-Width Negative Lookahead Assertions

The following grouping construct defines a zero-width negative lookahead assertion:

`(?!` *subexpression* `)`

where *subexpression* is any regular expression pattern. For the match to be successful, the input string must not match the regular expression pattern in *subexpression*, although the matched string is not included in the match result.

A zero-width negative lookahead assertion is typically used either at the beginning or at the end of a regular expression. At the beginning of a regular expression, it can define a specific pattern that should not be matched when the beginning of the regular expression defines a similar but more general pattern to be matched. In this case, it is often used to limit backtracking. At the end of a regular expression, it can define a subexpression that cannot occur at the end of a match.

The following example defines a regular expression that uses a zero-width lookahead assertion at the beginning of the regular expression to match words that do not begin with "un".

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?!un)\w+\b";
      string input = "unite one unethical ethics use untie ultimate";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       one
//       ethics
//       use
//       ultimate
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b(?!un)\w+\b"
      Dim input As String = "unite one unethical ethics use untie ultimate"
      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
         Console.WriteLine(match.Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       one
'       ethics
'       use
'       ultimate
```

The regular expression `\b(?!un)\w+\b` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`(?!un)`	Determine whether the next two characters are "un". If they are not, a match is possible.
`\w+`	Match one or more word characters.
`\b`	End the match at a word boundary.

The following example defines a regular expression that uses a zero-width lookahead assertion at the end of the regular expression to match words that do not end with a punctuation character.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\w+\b(?!\p{P})";
      string input = "Disconnected, disjointed thoughts in a sentence fragment.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       disjointed
//       thoughts
//       in
//       a
//       sentence
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b\w+\b(?!\p{P})"
      Dim input As String = "Disconnected, disjointed thoughts in a sentence fragment."
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       disjointed
'       thoughts
'       in
'       a
'       sentence
```

The regular expression `\b\w+\b(?!\p{P})` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`\w+`	Match one or more word characters.
`\b`	End the match at a word boundary.
`\p{P})`	If the next character is not a punctuation symbol (such as a period or a comma), the match succeeds.

Zero-Width Positive Lookbehind Assertions

The following grouping construct defines a zero-width positive lookbehind assertion:

`(?<=` *subexpression* `)`

where *subexpression* is any regular expression pattern. For a match to be successful, *subexpression* must occur at the input string to the left of the current position, although `subexpression` is not included in the match result. A zero-width positive lookbehind assertion does not backtrack.

Zero-width positive lookbehind assertions are typically used at the beginning of regular expressions. The pattern that they define is a precondition for a match, although it is not a part of the match result.

For example, the following example matches the last two digits of the year for the twenty first century (that is, it requires that the digits "20" precede the matched string).

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "2010 1999 1861 2140 2009";
      string pattern = @"(?<=\b20)\d{2}\b";

      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       10
//       09
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "2010 1999 1861 2140 2009"
      Dim pattern As String = "(?<=\b20)\d{2}\b"

      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       10
'       09
```

The regular expression pattern `(?<=\b20)\d{2}\b` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\d{2}`	Match two decimal digits.
`(?<=\b20)`	Continue the match if the two decimal digits are preceded by the decimal digits "20" on a word boundary.
`\b`	End the match at a word boundary.

Zero-width positive lookbehind assertions are also used to limit backtracking when the last character or characters in a captured group must be a subset of the characters that match that group's regular expression pattern. For example, if a group captures all consecutive word characters, you can use a zero-width positive lookbehind assertion to require that the last character be alphabetical.

Zero-Width Negative Lookbehind Assertions

The following grouping construct defines a zero-width negative lookbehind assertion:

`(?<!` *subexpression* `)`

where *subexpression* is any regular expression pattern. For a match to be successful, *subexpression* must not occur at the input string to the left of the current position. However, any substring that does not match `subexpression` is not included in the match result.

Zero-width negative lookbehind assertions are typically used at the beginning of regular expressions. The pattern that they define precludes a match in the string that follows. They are also used to limit backtracking when the last character or characters in a captured group must not be one or more of the characters that match that group's regular expression pattern. For example, if a group captures all consecutive word characters, you can use a zero-width positive lookbehind assertion to require that the last character not be an underscore (_).

The following example matches the date for any day of the week that is not a weekend (that is, that is neither Saturday nor Sunday).

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] dates = { "Monday February 1, 2010",
                         "Wednesday February 3, 2010",
                         "Saturday February 6, 2010",
                         "Sunday February 7, 2010",
                         "Monday, February 8, 2010" };
      string pattern = @"(?<!(Saturday|Sunday) )\b\w+ \d{1,2}, \d{4}\b";

      foreach (string dateValue in dates)
      {
         Match match = Regex.Match(dateValue, pattern);
         if (match.Success)
            Console.WriteLine(match.Value);
      }
   }
}
// The example displays the following output:
//       February 1, 2010
//       February 3, 2010
//       February 8, 2010
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim dates() As String = { "Monday February 1, 2010", _
                                "Wednesday February 3, 2010", _
                                "Saturday February 6, 2010", _
                                "Sunday February 7, 2010", _
                                "Monday, February 8, 2010" }
      Dim pattern As String = "(?<!(Saturday|Sunday) )\b\w+ \d{1,2}, \d{4}\b"

      For Each dateValue As String In dates
         Dim match As Match = Regex.Match(dateValue, pattern)
         If match.Success Then
            Console.WriteLine(match.Value)
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'       February 1, 2010
'       February 3, 2010
'       February 8, 2010
```

The regular expression pattern `(?<!(Saturday|Sunday) )\b\w+ \d{1,2}, \d{4}\b` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`\w+`	Match one or more word characters followed by a white-space character.
`\d{1,2},`	Match either one or two decimal digits followed by a white-space character and a comma.
`\d{4}\b`	Match four decimal digits, and end the match at a word boundary.
`(?<!(Saturday|Sunday) )`	If the match is preceded by something other than the strings "Saturday" or "Sunday" followed by a space, the match is successful.

Nonbacktracking Subexpressions

The following grouping construct represents a nonbacktracking subexpression (also known as a "greedy" subexpression):

`(?>` *subexpression* `)`

where *subexpression* is any regular expression pattern.

Ordinarily, if a regular expression includes an optional or alternative matching pattern and a match does not succeed, the regular expression engine can branch in multiple directions to match an input string with a pattern. If a match is not found when it takes the first branch, the regular expression engine can back up or backtrack to the point where it took the first match and attempt the match using the second branch. This process can continue until all branches have been tried.

The `(?>` *subexpression* `)` language construct disables backtracking. The regular expression engine will match as many characters in the input string as it can. When no further match is possible, it will not backtrack to attempt alternate pattern matches. (That is, the subexpression matches only strings that would be matched by the subexpression alone; it does not attempt to match a string based on the subexpression and any subexpressions that follow it.)

This option is recommended if you know that backtracking will not succeed. Preventing the regular expression engine from performing unnecessary searching improves performance.

The following example illustrates how a nonbacktracking subexpression modifies the results of a pattern match. The backtracking regular expression successfully matches a series of repeated characters followed by one more occurrence of the same character on a word boundary, but the nonbacktracking regular expression does not.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "cccd.", "aaad", "aaaa" };
      string back = @"(\w)\1+.\b";
      string noback = @"(?>(\w)\1+).\b";

      foreach (string input in inputs)
      {
         Match match1 = Regex.Match(input, back);
         Match match2 = Regex.Match(input, noback);
         Console.WriteLine("{0}: ", input);

         Console.Write("   Backtracking : ");
         if (match1.Success)
            Console.WriteLine(match1.Value);
         else
            Console.WriteLine("No match");

         Console.Write("   Nonbacktracking: ");
         if (match2.Success)
            Console.WriteLine(match2.Value);
         else
            Console.WriteLine("No match");
      }
   }
}
// The example displays the following output:
//    cccd.:
//       Backtracking : cccd
//       Nonbacktracking: cccd
//    aaad:
//       Backtracking : aaad
//       Nonbacktracking: aaad
//    aaaa:
//       Backtracking : aaaa
//       Nonbacktracking: No match
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim inputs() As String = { "cccd.", "aaad", "aaaa" }
      Dim back As String = "(\w)\1+.\b"
      Dim noback As String = "(?>(\w)\1+).\b"

      For Each input As String In inputs
         Dim match1 As Match = Regex.Match(input, back)
         Dim match2 As Match = Regex.Match(input, noback)
         Console.WriteLine("{0}: ", input)

         Console.Write("   Backtracking : ")
         If match1.Success Then
            Console.WriteLine(match1.Value)
         Else
            Console.WriteLine("No match")
         End If

         Console.Write("   Nonbacktracking: ")
         If match2.Success Then
            Console.WriteLine(match2.Value)
         Else
            Console.WriteLine("No match")
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'    cccd.:
'       Backtracking : cccd
'       Nonbacktracking: cccd
'    aaad:
'       Backtracking : aaad
'       Nonbacktracking: aaad
'    aaaa:
'       Backtracking : aaaa
'       Nonbacktracking: No match
```

The nonbacktracking regular expression `(?>(\w)\1+).\b` is defined as shown in the following table.

PATTERN	DESCRIPTION
`(\w)`	Match a single word character and assign it to the first capturing group.
`\1+`	Match the value of the first captured substring one or more times.
`.`	Match any character.
`\b`	End the match on a word boundary.
`(?>(\w)\1+)`	Match one or more occurrences of a duplicated word character, but do not backtrack to match the last character on a word boundary.

Grouping Constructs and Regular Expression Objects

Substrings that are matched by a regular expression capturing group are represented by

System.Text.RegularExpressions.Group objects, which can be retrieved from the System.Text.RegularExpressions.GroupCollection object that is returned by the Match.Groups property. The GroupCollection object is populated as follows:

- The first Group object in the collection (the object at index zero) represents the entire match.
- The next set of Group objects represent unnamed (numbered) capturing groups. They appear in the order in which they are defined in the regular expression, from left to right. The index values of these groups range from 1 to the number of unnamed capturing groups in the collection. (The index of a particular group is equivalent to its numbered backreference. For more information about backreferences, see Backreference Constructs.)
- The final set of Group objects represent named capturing groups. They appear in the order in which they are defined in the regular expression, from left to right. The index value of the first named capturing group is one greater than the index of the last unnamed capturing group. If there are no unnamed capturing groups in the regular expression, the index value of the first named capturing group is one.

If you apply a quantifier to a capturing group, the corresponding Group object's Capture.Value, Capture.Index, and Capture.Length properties reflect the last substring that is captured by a capturing group. You can retrieve a complete set of substrings that are captured by groups that have quantifiers from the CaptureCollection object that is returned by the Group.Captures property.

The following example clarifies the relationship between the Group and Capture objects.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\b(\w+)\W+)+";
      string input = "This is a short sentence.";
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match: '{0}'", match.Value);
      for (int ctr = 1; ctr < match.Groups.Count; ctr++)
      {
         Console.WriteLine("   Group {0}: '{1}'", ctr, match.Groups[ctr].Value);
         int capCtr = 0;
         foreach (Capture capture in match.Groups[ctr].Captures)
         {
            Console.WriteLine("      Capture {0}: '{1}'", capCtr, capture.Value);
            capCtr++;
         }
      }
   }
}
// The example displays the following output:
//       Match: 'This is a short sentence.'
//          Group 1: 'sentence.'
//             Capture 0: 'This '
//             Capture 1: 'is '
//             Capture 2: 'a '
//             Capture 3: 'short '
//             Capture 4: 'sentence.'
//          Group 2: 'sentence'
//             Capture 0: 'This'
//             Capture 1: 'is'
//             Capture 2: 'a'
//             Capture 3: 'short'
//             Capture 4: 'sentence'
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(\b(\w+)\W+)+"
      Dim input As String = "This is a short sentence."
      Dim match As Match = Regex.Match(input, pattern)
      Console.WriteLine("Match: '{0}'", match.Value)
      For ctr As Integer = 1 To match.Groups.Count - 1
         Console.WriteLine("   Group {0}: '{1}'", ctr, match.Groups(ctr).Value)
         Dim capCtr As Integer = 0
         For Each capture As Capture In match.Groups(ctr).Captures
            Console.WriteLine("      Capture {0}: '{1}'", capCtr, capture.Value)
            capCtr += 1
         Next
      Next
   End Sub
End Module
' The example displays the following output:
'       Match: 'This is a short sentence.'
'          Group 1: 'sentence.'
'             Capture 0: 'This '
'             Capture 1: 'is '
'             Capture 2: 'a '
'             Capture 3: 'short '
'             Capture 4: 'sentence.'
'          Group 2: 'sentence'
'             Capture 0: 'This'
'             Capture 1: 'is'
'             Capture 2: 'a'
'             Capture 3: 'short'
'             Capture 4: 'sentence'
```

The regular expression pattern `\b(\w+)\W+)+` extracts individual words from a string. It is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`(\w+)`	Match one or more word characters. Together, these characters form a word. This is the second capturing group.
`\W+`	Match one or more non-word characters.
`(\w+)\W+)+`	Match the pattern of one or more word characters followed by one or more non-word characters one or more times. This is the first capturing group.

The first capturing group matches each word of the sentence. The second capturing group matches each word along with the punctuation and white space that follow the word. The Group object whose index is 2 provides information about the text matched by the second capturing group. The complete set of words captured by the capturing group are available from the CaptureCollection object returned by the Group.Captures property.

See also

- Regular Expression Language - Quick Reference
- Backtracking

Quantifiers in Regular Expressions

9/6/2018 • 23 minutes to read • Edit Online

Quantifiers specify how many instances of a character, group, or character class must be present in the input for a match to be found. The following table lists the quantifiers supported by .NET.

GREEDY QUANTIFIER	LAZY QUANTIFIER	DESCRIPTION
`*`	`*?`	Match zero or more times.
`+`	`+?`	Match one or more times.
`?`	`??`	Match zero or one time.
`{` *n* `}`	`{` *n* `}?`	Match exactly *n* times.
`{` *n* `,}`	`{` *n* `,}?`	Match at least *n* times.
`{` *n* `,` *m* `}`	`{` *n* `,` *m* `}?`	Match from *n* to *m* times.

The quantities `n` and `m` are integer constants. Ordinarily, quantifiers are greedy; they cause the regular expression engine to match as many occurrences of particular patterns as possible. Appending the `?` character to a quantifier makes it lazy; it causes the regular expression engine to match as few occurrences as possible. For a complete description of the difference between greedy and lazy quantifiers, see the section Greedy and Lazy Quantifiers later in this topic.

IMPORTANT

Nesting quantifiers (for example, as the regular expression pattern `(a*)*` does) can increase the number of comparisons that the regular expression engine must perform, as an exponential function of the number of characters in the input string. For more information about this behavior and its workarounds, see Backtracking.

Regular Expression Quantifiers

The following sections list the quantifiers supported by .NET regular expressions.

NOTE

If the *, +, ?, {, and } characters are encountered in a regular expression pattern, the regular expression engine interprets them as quantifiers or part of quantifier constructs unless they are included in a character class. To interpret these as literal characters outside a character class, you must escape them by preceding them with a backslash. For example, the string `\*` in a regular expression pattern is interpreted as a literal asterisk ("*") character.

Match Zero or More Times: *

The `*` quantifier matches the preceding element zero or more times. It is equivalent to the `{0,}` quantifier. `*` is a greedy quantifier whose lazy equivalent is `*?`.

The following example illustrates this regular expression. Of the nine digits in the input string, five match the pattern and four (`95` , `929` , `9129` , and `9919`) do not.

```
string pattern = @"\b91*9*\b";
string input = "99 95 919 929 9119 9219 999 9919 91119";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

// The example displays the following output:
//       '99' found at position 0.
//       '919' found at position 6.
//       '9119' found at position 14.
//       '999' found at position 24.
//       '91119' found at position 33.
```

```
Dim pattern As String = "\b91*9*\b"
Dim input As String = "99 95 919 929 9119 9219 999 9919 91119"
For Each match As Match In Regex.Matches(input, pattern)
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       '99' found at position 0.
'       '919' found at position 6.
'       '9119' found at position 14.
'       '999' found at position 24.
'       '91119' found at position 33.
```

The regular expression pattern is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`91*`	Match a "9" followed by zero or more "1" characters.
`9*`	Match zero or more "9" characters.
`\b`	End at a word boundary.

Match One or More Times: +

The `+` quantifier matches the preceding element one or more times. It is equivalent to `{1,}`. `+` is a greedy quantifier whose lazy equivalent is `+?`.

For example, the regular expression `\ban+\w*?\b` tries to match entire words that begin with the letter `a` followed by one or more instances of the letter `n`. The following example illustrates this regular expression. The regular expression matches the words `an`, `annual`, `announcement`, and `antique`, and correctly fails to match `autumn` and `all`.

```
string pattern = @"\ban+\w*?\b";

string input = "Autumn is a great time for an annual announcement to all antique collectors.";
foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

// The example displays the following output:
//       'an' found at position 27.
//       'annual' found at position 30.
//       'announcement' found at position 37.
//       'antique' found at position 57.
```

```
Dim pattern As String = "\ban+\w*?\b"

Dim input As String = "Autumn is a great time for an annual announcement to all antique collectors."
For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'an' found at position 27.
'       'annual' found at position 30.
'       'announcement' found at position 37.
'       'antique' found at position 57.
```

The regular expression pattern is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`an+`	Match an "a" followed by one or more "n" characters.
`\w*?`	Match a word character zero or more times, but as few times as possible.
`\b`	End at a word boundary.

Match Zero or One Time: ?

The `?` quantifier matches the preceding element zero or one time. It is equivalent to `{0,1}`. `?` is a greedy quantifier whose lazy equivalent is `??`.

For example, the regular expression `\ban?\b` tries to match entire words that begin with the letter `a` followed by zero or one instances of the letter `n`. In other words, it tries to match the words `a` and `an`. The following example illustrates this regular expression.

```
string pattern = @"\ban?\b";
string input = "An amiable animal with a large snount and an animated nose.";
foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

// The example displays the following output:
//        'An' found at position 0.
//        'a' found at position 23.
//        'an' found at position 42.
```

```
Dim pattern As String = "\ban?\b"
Dim input As String = "An amiable animal with a large snount and an animated nose."
For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'An' found at position 0.
'       'a' found at position 23.
'       'an' found at position 42.
```

The regular expression pattern is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`an?`	Match an "a" followed by zero or one "n" character.
`\b`	End at a word boundary.

Match Exactly n Times: {n}

The `{` *n* `}` quantifier matches the preceding element exactly *n* times, where *n* is any integer. `{` *n* `}` is a greedy quantifier whose lazy equivalent is `{` *n* `}?`.

For example, the regular expression `\b\d+\,\d{3}\b` tries to match a word boundary followed by one or more decimal digits followed by three decimal digits followed by a word boundary. The following example illustrates this regular expression.

```
string pattern = @"\b\d+\,\d{3}\b";
string input = "Sales totaled 103,524 million in January, " +
                      "106,971 million in February, but only " +
                      "943 million in March.";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        '103,524' found at position 14.
//        '106,971' found at position 45.
```

```
Dim pattern As String = "\b\d+\,\d{3}\b"
Dim input As String = "Sales totaled 103,524 million in January, " + _
                      "106,971 million in February, but only " + _
                      "943 million in March."
For Each match As Match In Regex.Matches(input, pattern)
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       '103,524' found at position 14.
'       '106,971' found at position 45.
```

The regular expression pattern is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.

PATTERN	DESCRIPTION
`\d+`	Match one or more decimal digits.
`\,`	Match a comma character.
`\d{3}`	Match three decimal digits.
`\b`	End at a word boundary.

Match at Least n Times: {n,}

The `{` *n* `,}` quantifier matches the preceding element at least *n* times, where *n* is any integer. `{` *n* `,}` is a greedy quantifier whose lazy equivalent is `{` *n* `,}?`.

For example, the regular expression `\b\d{2,}\b\D+` tries to match a word boundary followed by at least two digits followed by a word boundary and a non-digit character. The following example illustrates this regular expression. The regular expression fails to match the phrase `"7 days"` because it contains just one decimal digit, but it successfully matches the phrases `"10 weeks and 300 years"`.

```
string pattern = @"\b\d{2,}\b\D+";
string input = "7 days, 10 weeks, 300 years";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        '10 weeks, ' found at position 8.
//        '300 years' found at position 18.
```

```
 Dim pattern As String = "\b\d{2,}\b\D+"
 Dim input As String = "7 days, 10 weeks, 300 years"
For Each match As Match In Regex.Matches(input, pattern)
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       '10 weeks, ' found at position 8.
'       '300 years' found at position 18.
```

The regular expression pattern is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`\d{2,}`	Match at least two decimal digits.
`\b`	Match a word boundary.
`\D+`	Match at least one non-decimal digit.

Match Between n and m Times: {n,m}

The `{` *n* `,` *m* `}` quantifier matches the preceding element at least *n* times, but no more than *m* times, where *n* and *m* are integers. `{` *n* `,` *m* `}` is a greedy quantifier whose lazy equivalent is `{` *n* `,` *m* `}?`.

In the following example, the regular expression `(00\s){2,4}` tries to match between two and four occurrences of two zero digits followed by a space. Note that the final portion of the input string includes this pattern five times rather than the maximum of four. However, only the initial portion of this substring (up to the space and the fifth pair of zeros) matches the regular expression pattern.

```
string pattern = @"(00\s){2,4}";
string input = "0x00 FF 00 00 18 17 FF 00 00 00 21 00 00 00 00 00";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        '00 00 ' found at position 8.
//        '00 00 00 ' found at position 23.
//        '00 00 00 00 ' found at position 35.
```

```
Dim pattern As String = "(00\s){2,4}"
Dim input As String = "0x00 FF 00 00 18 17 FF 00 00 00 21 00 00 00 00 00"
For Each match As Match In Regex.Matches(input, pattern)
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       '00 00 ' found at position 8.
'       '00 00 00 ' found at position 23.
'       '00 00 00 00 ' found at position 35.
```

Match Zero or More Times (Lazy Match): *?

The `*?` quantifier matches the preceding element zero or more times, but as few times as possible. It is the lazy counterpart of the greedy quantifier `*`.

In the following example, the regular expression `\b\w*?oo\w*?\b` matches all words that contain the string `oo`.

```
 string pattern = @"\b\w*?oo\w*?\b";
 string input = "woof root root rob oof woo woe";
 foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

 //  The example displays the following output:
//        'woof' found at position 0.
//        'root' found at position 5.
//        'root' found at position 10.
//        'oof' found at position 19.
//        'woo' found at position 23.
```

```
 Dim pattern As String = "\b\w*?oo\w*?\b"
 Dim input As String = "woof root root rob oof woo woe"
 For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
 Next
 ' The example displays the following output:
'       'woof' found at position 0.
'       'root' found at position 5.
'       'root' found at position 10.
'       'oof' found at position 19.
'       'woo' found at position 23.
```

The regular expression pattern is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`\w*?`	Match zero or more word characters, but as few characters as possible.
`oo`	Match the string "oo".
`\w*?`	Match zero or more word characters, but as few characters as possible.
`\b`	End on a word boundary.

Match One or More Times (Lazy Match): +?

The `+?` quantifier matches the preceding element one or more times, but as few times as possible. It is the lazy counterpart of the greedy quantifier `+`.

For example, the regular expression `\b\w+?\b` matches one or more characters separated by word boundaries. The following example illustrates this regular expression.

```
string pattern = @"\b\w+?\b";
string input = "Aa Bb Cc Dd Ee Ff";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        'Aa' found at position 0.
//        'Bb' found at position 3.
//        'Cc' found at position 6.
//        'Dd' found at position 9.
//        'Ee' found at position 12.
//        'Ff' found at position 15.
```

```
 Dim pattern As String = "\b\w+?\b"
 Dim input As String = "Aa Bb Cc Dd Ee Ff"
For Each match As Match In Regex.Matches(input, pattern)
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'Aa' found at position 0.
'       'Bb' found at position 3.
'       'Cc' found at position 6.
'       'Dd' found at position 9.
'       'Ee' found at position 12.
'       'Ff' found at position 15.
```

Match Zero or One Time (Lazy Match): ??

The `??` quantifier matches the preceding element zero or one time, but as few times as possible. It is the lazy counterpart of the greedy quantifier `?`.

For example, the regular expression `^\s*(System.)??Console.Write(Line)??\(??` attempts to match the strings "Console.Write" or "Console.WriteLine". The string can also include "System." before "Console", and it can be followed by an opening parenthesis. The string must be at the beginning of a line, although it can be preceded by white space. The following example illustrates this regular expression.

```
string pattern = @"^\s*(System.)??Console.Write(Line)??\(??";
string input = "System.Console.WriteLine(\"Hello!\")\n" +
                      "Console.Write(\"Hello!\")\n" +
                      "Console.WriteLine(\"Hello!\")\n" +
                      "Console.ReadLine()\n" +
                      "   Console.WriteLine";
foreach (Match match in Regex.Matches(input, pattern,
                                      RegexOptions.IgnorePatternWhitespace |
                                      RegexOptions.IgnoreCase |
                                      RegexOptions.Multiline))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        'System.Console.Write' found at position 0.
//        'Console.Write' found at position 36.
//        'Console.Write' found at position 61.
//        '   Console.Write' found at position 110.
```

```
Dim pattern As String = "^\s*(System.)??Console.Write(Line)??\(??"
Dim input As String = "System.Console.WriteLine(""Hello!"")" + vbCrLf + _
                      "Console.Write(""Hello!"")" + vbCrLf + _
                      "Console.WriteLine(""Hello!"")" + vbCrLf + _
                      "Console.ReadLine()" + vbCrLf + _
                      "   Console.WriteLine"
For Each match As Match In Regex.Matches(input, pattern, _
                                         RegexOptions.IgnorePatternWhitespace Or RegexOptions.IgnoreCase Or
RegexOptions.MultiLine)
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'System.Console.Write' found at position 0.
'       'Console.Write' found at position 36.
'       'Console.Write' found at position 61.
'       '   Console.Write' found at position 110.
```

The regular expression pattern is defined as shown in the following table.

PATTERN	DESCRIPTION
`^`	Match the start of the input stream.
`\s*`	Match zero or more white-space characters.
`(System.)??`	Match zero or one occurrence of the string "System.".
`Console.Write`	Match the string "Console.Write".
`(Line)??`	Match zero or one occurrence of the string "Line".
`\(??`	Match zero or one occurrence of the opening parenthesis.

Match Exactly n Times (Lazy Match): {n}?

The `{` *n* `}?` quantifier matches the preceding element exactly `n` times, where *n* is any integer. It is the lazy counterpart of the greedy quantifier `{` *n* `}+`.

In the following example, the regular expression `\b(\w{3,}?\.){2}?\w{3,}?\b` is used to identify a Web site address. Note that it matches "www.microsoft.com" and "msdn.microsoft.com", but does not match "mywebsite" or "mycompany.com".

```
string pattern = @"\b(\w{3,}?\.){2}?\w{3,}?\b";
string input = "www.microsoft.com msdn.microsoft.com mywebsite mycompany.com";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        'www.microsoft.com' found at position 0.
//        'msdn.microsoft.com' found at position 18.
```

```
 Dim pattern As String = "\b(\w{3,}?\.){2}?\w{3,}?\b"
 Dim input As String = "www.microsoft.com msdn.microsoft.com mywebsite mycompany.com"
For Each match As Match In Regex.Matches(input, pattern)
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'www.microsoft.com' found at position 0.
'       'msdn.microsoft.com' found at position 18.
```

The regular expression pattern is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`(\w{3,}?\.)`	Match at least 3 word characters, but as few characters as possible, followed by a dot or period character. This is the first capturing group.
`(\w{3,}?\.){2}?`	Match the pattern in the first group two times, but as few times as possible.
`\b`	End the match on a word boundary.

Match at Least n Times (Lazy Match): {n,}?

The `{` *n* `,}?` quantifier matches the preceding element at least `n` times, where *n* is any integer, but as few times as possible. It is the lazy counterpart of the greedy quantifier `{` *n* `,}`.

See the example for the `{` *n* `}?` quantifier in the previous section for an illustration. The regular expression in that example uses the `{` *n* `,}` quantifier to match a string that has at least three characters followed by a period.

Match Between n and m Times (Lazy Match): {n,m}?

The `{` *n* `,` *m* `}?` quantifier matches the preceding element between `n` and `m` times, where *n* and *m* are integers, but as few times as possible. It is the lazy counterpart of the greedy quantifier `{` *n* `,` *m* `}`.

In the following example, the regular expression `\b[A-Z](\w*\s+){1,10}?[.!?]` matches sentences that contain between one and ten words. It matches all the sentences in the input string except for one sentence that contains 18 words.

```
string pattern = @"\b[A-Z](\w*?\s*?){1,10}[.!?]";
string input = "Hi. I am writing a short note. Its purpose is " +
                      "to test a regular expression that attempts to find " +
                      "sentences with ten or fewer words. Most sentences " +
                      "in this note are short.";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        'Hi.' found at position 0.
//        'I am writing a short note.' found at position 4.
//        'Most sentences in this note are short.' found at position 132.
```

```
Dim pattern As String = "\b[A-Z](\w*\s?){1,10}?[.!?]"
Dim input As String = "Hi. I am writing a short note. Its purpose is " + _
                      "to test a regular expression that attempts to find " + _
                      "sentences with ten or fewer words. Most sentences " + _
                      "in this note are short."
For Each match As Match In Regex.Matches(input, pattern)
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'Hi.' found at position 0.
'       'I am writing a short note.' found at position 4.
'       'Most sentences in this note are short.' found at position 132.
```

The regular expression pattern is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`[A-Z]`	Match an uppercase character from A to Z.
`(\w*\s+)`	Match zero or more word characters, followed by one or more white-space characters. This is the first capture group.
`{1,10}?`	Match the previous pattern between 1 and 10 times, but as few times as possible.
`[.!?]`	Match any one of the punctuation characters ".", "!", or "?".

Greedy and Lazy Quantifiers

A number of the quantifiers have two versions:

- A greedy version.

  A greedy quantifier tries to match an element as many times as possible.

- A non-greedy (or lazy) version.

  A non-greedy quantifier tries to match an element as few times as possible. You can turn a greedy quantifier into a lazy quantifier by simply adding a `?`.

Consider a simple regular expression that is intended to extract the last four digits from a string of numbers such as a credit card number. The version of the regular expression that uses the `*` greedy quantifier is `\b.*([0-9]{4})\b`. However, if a string contains two numbers, this regular expression matches the last four digits of the second number only, as the following example shows.

```
string greedyPattern = @"\b.*([0-9]{4})\b";
string input1 = "1112223333 3992991999";
foreach (Match match in Regex.Matches(input1, greedyPattern))
   Console.WriteLine("Account ending in ******{0}.", match.Groups[1].Value);

// The example displays the following output:
//       Account ending in ******1999.
```

```
Dim greedyPattern As String = "\b.*([0-9]{4})\b"
Dim input1 As String = "1112223333 3992991999"
For Each match As Match In Regex.Matches(input1, greedypattern)
   Console.WriteLine("Account ending in ******{0}.", match.Groups(1).Value)
Next
' The example displays the following output:
'       Account ending in ******1999.
```

The regular expression fails to match the first number because the `*` quantifier tries to match the previous element as many times as possible in the entire string, and so it finds its match at the end of the string.

This is not the desired behavior. Instead, you can use the `*?` lazy quantifier to extract digits from both numbers, as the following example shows.

```
string lazyPattern = @"\b.*?([0-9]{4})\b";
string input2 = "1112223333 3992991999";
foreach (Match match in Regex.Matches(input2, lazyPattern))
   Console.WriteLine("Account ending in ******{0}.", match.Groups[1].Value);

// The example displays the following output:
//       Account ending in ******3333.
//       Account ending in ******1999.
```

```
Dim lazyPattern As String = "\b.*?([0-9]{4})\b"
Dim input2 As String = "1112223333 3992991999"
For Each match As Match In Regex.Matches(input2, lazypattern)
   Console.WriteLine("Account ending in ******{0}.", match.Groups(1).Value)
Next
' The example displays the following output:
'       Account ending in ******3333.
'       Account ending in ******1999.
```

In most cases, regular expressions with greedy and lazy quantifiers return the same matches. They most commonly return different results when they are used with the wildcard (`.`) metacharacter, which matches any character.

Quantifiers and Empty Matches

The quantifiers `*`, `+`, and `{` *n* `,` *m* `}` and their lazy counterparts never repeat after an empty match when the minimum number of captures has been found. This rule prevents quantifiers from entering infinite loops on empty subexpression matches when the maximum number of possible group captures is infinite or near infinite.

For example, the following code shows the result of a call to the Regex.Match method with the regular expression pattern `(a?)*`, which matches zero or one "a" character zero or more times. Note that the single capturing group captures each "a" as well as String.Empty, but that there is no second empty match, because the first empty match causes the quantifier to stop repeating.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "(a?)*";
      string input = "aaabbb";
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match: '{0}' at index {1}",
                        match.Value, match.Index);
      if (match.Groups.Count > 1) {
         GroupCollection groups = match.Groups;
         for (int grpCtr = 1; grpCtr <= groups.Count - 1; grpCtr++) {
            Console.WriteLine("   Group {0}: '{1}' at index {2}",
                              grpCtr,
                              groups[grpCtr].Value,
                              groups[grpCtr].Index);
            int captureCtr = 0;
            foreach (Capture capture in groups[grpCtr].Captures) {
               captureCtr++;
               Console.WriteLine("      Capture {0}: '{1}' at index {2}",
                                 captureCtr, capture.Value, capture.Index);
            }
         }
      }
   }
}
// The example displays the following output:
//       Match: 'aaa' at index 0
//          Group 1: '' at index 3
//             Capture 1: 'a' at index 0
//             Capture 2: 'a' at index 1
//             Capture 3: 'a' at index 2
//             Capture 4: '' at index 3
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(a?)*"
      Dim input As String = "aaabbb"
      Dim match As Match = Regex.Match(input, pattern)
      Console.WriteLine("Match: '{0}' at index {1}",
                        match.Value, match.Index)
      If match.Groups.Count > 1 Then
         Dim groups As GroupCollection = match.Groups
         For grpCtr As Integer = 1 To groups.Count - 1
            Console.WriteLine("   Group {0}: '{1}' at index {2}",
                              grpCtr,
                              groups(grpCtr).Value,
                              groups(grpCtr).Index)
            Dim captureCtr As Integer = 0
            For Each capture As Capture In groups(grpCtr).Captures
               captureCtr += 1
               Console.WriteLine("      Capture {0}: '{1}' at index {2}",
                                 captureCtr, capture.Value, capture.Index)
            Next
         Next
      End If
   End Sub
End Module
' The example displays the following output:
'       Match: 'aaa' at index 0
'          Group 1: '' at index 3
'             Capture 1: 'a' at index 0
'             Capture 2: 'a' at index 1
'             Capture 3: 'a' at index 2
'             Capture 4: '' at index 3
```

To see the practical difference between a capturing group that defines a minimum and a maximum number of captures and one that defines a fixed number of captures, consider the regular expression patterns `(a\1|(?(1)\1)){0,2}` and `(a\1|(?(1)\1)){2}`. Both regular expressions consist of a single capturing group, which is defined as shown in the following table.

PATTERN	DESCRIPTION
`(a\1`	Either match "a" along with the value of the first captured group ...
`|(?(1)`	... or test whether the first captured group has been defined. (Note that the `(?(1)` construct does not define a capturing group.)
`\1))`	If the first captured group exists, match its value. If the group does not exist, the group will match String.Empty.

The first regular expression tries to match this pattern between zero and two times; the second, exactly two times. Because the first pattern reaches its minimum number of captures with its first capture of String.Empty, it never repeats to try to match `a\1`; the `{0,2}` quantifier allows only empty matches in the last iteration. In contrast, the second regular expression does match "a" because it evaluates `a\1` a second time; the minimum number of iterations, 2, forces the engine to repeat after an empty match.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern, input;

      pattern = @"(a\1|(?(1)\1)){0,2}";
      input = "aaabbb";

      Console.WriteLine("Regex pattern: {0}", pattern);
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match: '{0}' at position {1}.",
                        match.Value, match.Index);
      if (match.Groups.Count > 1) {
         for (int groupCtr = 1; groupCtr <= match.Groups.Count - 1; groupCtr++)
         {
            Group group = match.Groups[groupCtr];
            Console.WriteLine("   Group: {0}: '{1}' at position {2}.",
                              groupCtr, group.Value, group.Index);
            int captureCtr = 0;
            foreach (Capture capture in group.Captures) {
               captureCtr++;
               Console.WriteLine("      Capture: {0}: '{1}' at position {2}.",
                                 captureCtr, capture.Value, capture.Index);
            }
         }
      }
      Console.WriteLine();

      pattern = @"(a\1|(?(1)\1)){2}";
      Console.WriteLine("Regex pattern: {0}", pattern);
      match = Regex.Match(input, pattern);
         Console.WriteLine("Matched '{0}' at position {1}.",
                           match.Value, match.Index);
      if (match.Groups.Count > 1) {
         for (int groupCtr = 1; groupCtr <= match.Groups.Count - 1; groupCtr++)
         {
            Group group = match.Groups[groupCtr];
            Console.WriteLine("   Group: {0}: '{1}' at position {2}.",
                              groupCtr, group.Value, group.Index);
            int captureCtr = 0;
            foreach (Capture capture in group.Captures) {
               captureCtr++;
               Console.WriteLine("      Capture: {0}: '{1}' at position {2}.",
                                 captureCtr, capture.Value, capture.Index);
            }
         }
      }
   }
}
// The example displays the following output:
//       Regex pattern: (a\1|(?(1)\1)){0,2}
//       Match: '' at position 0.
//          Group: 1: '' at position 0.
//             Capture: 1: '' at position 0.
//
//       Regex pattern: (a\1|(?(1)\1)){2}
//       Matched 'a' at position 0.
//          Group: 1: 'a' at position 0.
//             Capture: 1: '' at position 0.
//             Capture: 2: 'a' at position 0.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern, input As String

      pattern = "(a\1|(?(1)\1)){0,2}"
      input = "aaabbb"

      Console.WriteLine("Regex pattern: {0}", pattern)
      Dim match As Match = Regex.Match(input, pattern)
      Console.WriteLine("Match: '{0}' at position {1}.",
                        match.Value, match.Index)
      If match.Groups.Count > 1 Then
         For groupCtr As Integer = 1 To match.Groups.Count - 1
            Dim group As Group = match.Groups(groupCtr)
            Console.WriteLine("   Group: {0}: '{1}' at position {2}.",
                              groupCtr, group.Value, group.Index)
            Dim captureCtr As Integer = 0
            For Each capture As Capture In group.Captures
               captureCtr += 1
               Console.WriteLine("      Capture: {0}: '{1}' at position {2}.",
                                 captureCtr, capture.Value, capture.Index)
            Next
         Next
      End If
      Console.WriteLine()

      pattern = "(a\1|(?(1)\1)){2}"
      Console.WriteLine("Regex pattern: {0}", pattern)
      match = Regex.Match(input, pattern)
         Console.WriteLine("Matched '{0}' at position {1}.",
                           match.Value, match.Index)
      If match.Groups.Count > 1 Then
         For groupCtr As Integer = 1 To match.Groups.Count - 1
            Dim group As Group = match.Groups(groupCtr)
            Console.WriteLine("   Group: {0}: '{1}' at position {2}.",
                              groupCtr, group.Value, group.Index)
            Dim captureCtr As Integer = 0
            For Each capture As Capture In group.Captures
               captureCtr += 1
               Console.WriteLine("      Capture: {0}: '{1}' at position {2}.",
                                 captureCtr, capture.Value, capture.Index)
            Next
         Next
      End If
   End Sub
End Module
' The example displays the following output:
'       Regex pattern: (a\1|(?(1)\1)){0,2}
'       Match: '' at position 0.
'          Group: 1: '' at position 0.
'             Capture: 1: '' at position 0.
'
'       Regex pattern: (a\1|(?(1)\1)){2}
'       Matched 'a' at position 0.
'          Group: 1: 'a' at position 0.
'             Capture: 1: '' at position 0.
'             Capture: 2: 'a' at position 0.
```

See also

- Regular Expression Language - Quick Reference
- Backtracking

Backreference Constructs in Regular Expressions

9/6/2018 • 10 minutes to read • Edit Online

Backreferences provide a convenient way to identify a repeated character or substring within a string. For example, if the input string contains multiple occurrences of an arbitrary substring, you can match the first occurrence with a capturing group, and then use a backreference to match subsequent occurrences of the substring.

> **NOTE**
>
> A separate syntax is used to refer to named and numbered capturing groups in replacement strings. For more information, see Substitutions.

.NET defines separate language elements to refer to numbered and named capturing groups. For more information about capturing groups, see Grouping Constructs.

Numbered Backreferences

A numbered backreference uses the following syntax:

`\` *number*

where *number* is the ordinal position of the capturing group in the regular expression. For example, `\4` matches the contents of the fourth capturing group. If *number* is not defined in the regular expression pattern, a parsing error occurs, and the regular expression engine throws an ArgumentException. For example, the regular expression `\b(\w+)\s\1` is valid, because `(\w+)` is the first and only capturing group in the expression. On the other hand, `\b(\w+)\s\2` is invalid and throws an argument exception, because there is no capturing group numbered `\2`. In addition, if *number* identifies a capturing group in a particular ordinal position, but that capturing group has been assigned a numeric name different than its ordinal position, the regular expression parser also throws an ArgumentException.

Note the ambiguity between octal escape codes (such as `\16`) and `\` *number* backreferences that use the same notation. This ambiguity is resolved as follows:

- The expressions `\1` through `\9` are always interpreted as backreferences, and not as octal codes.
- If the first digit of a multidigit expression is 8 or 9 (such as `\80` or `\91`), the expression as interpreted as a literal.
- Expressions from `\10` and greater are considered backreferences if there is a backreference corresponding to that number; otherwise, they are interpreted as octal codes.
- If a regular expression contains a backreference to an undefined group number, a parsing error occurs, and the regular expression engine throws an ArgumentException.

If the ambiguity is a problem, you can use the `\k<` *name* `>` notation, which is unambiguous and cannot be confused with octal character codes. Similarly, hexadecimal codes such as `\xdd` are unambiguous and cannot be confused with backreferences.

The following example finds doubled word characters in a string. It defines a regular expression, `(\w)\1`, which consists of the following elements.

ELEMENT	DESCRIPTION
`(\w)`	Match a word character and assign it to the first capturing group.
`\1`	Match the next character that is the same as the value of the first capturing group.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\w)\1";
      string input = "trellis llama webbing dresser swagger";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Found '{0}' at position {1}.",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       Found 'll' at position 3.
//       Found 'll' at position 8.
//       Found 'bb' at position 16.
//       Found 'ss' at position 25.
//       Found 'gg' at position 33.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(\w)\1"
      Dim input As String = "trellis llama webbing dresser swagger"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("Found '{0}' at position {1}.", _
                           match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       Found 'll' at position 3.
'       Found 'll' at position 8.
'       Found 'bb' at position 16.
'       Found 'ss' at position 25.
'       Found 'gg' at position 33.
```

Named Backreferences

A named backreference is defined by using the following syntax:

`\k<` *name* `>`

or:

`\k'` *name* `'`

where *name* is the name of a capturing group defined in the regular expression pattern. If *name* is not defined in the regular expression pattern, a parsing error occurs, and the regular expression engine throws an

ArgumentException.

The following example finds doubled word characters in a string. It defines a regular expression, `(?<char>\w)\k<char>`, which consists of the following elements.

ELEMENT	DESCRIPTION
`(?<char>\w)`	Match a word character and assign it to a capturing group named `char`.
`\k<char>`	Match the next character that is the same as the value of the `char` capturing group.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?<char>\w)\k<char>";
      string input = "trellis llama webbing dresser swagger";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Found '{0}' at position {1}.",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       Found 'll' at position 3.
//       Found 'll' at position 8.
//       Found 'bb' at position 16.
//       Found 'ss' at position 25.
//       Found 'gg' at position 33.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(?<char>\w)\k<char>"
      Dim input As String = "trellis llama webbing dresser swagger"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("Found '{0}' at position {1}.", _
                           match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       Found 'll' at position 3.
'       Found 'll' at position 8.
'       Found 'bb' at position 16.
'       Found 'ss' at position 25.
'       Found 'gg' at position 33.
```

Named numeric backreferences

In a named backreference with `\k`, *name* can also be the string representation of a number. For example, the following example uses the regular expression `(?<2>\w)\k<2>` to find doubled word characters in a string. In this case, the example defines a capturing group that is explicitly named "2", and the backreference is correspondingly named "2".

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?<2>\w)\k<2>";
      string input = "trellis llama webbing dresser swagger";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Found '{0}' at position {1}.",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       Found 'll' at position 3.
//       Found 'll' at position 8.
//       Found 'bb' at position 16.
//       Found 'ss' at position 25.
//       Found 'gg' at position 33.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(?<2>\w)\k<2>"
      Dim input As String = "trellis llama webbing dresser swagger"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("Found '{0}' at position {1}.", _
                           match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       Found 'll' at position 3.
'       Found 'll' at position 8.
'       Found 'bb' at position 16.
'       Found 'ss' at position 25.
'       Found 'gg' at position 33.
```

If *name* is the string representation of a number, and no capturing group has that name, `\k<` *name* `>` is the same as the backreference `\` *number*, where *number* is the ordinal position of the capture. In the following example, there is a single capturing group named `char`. The backreference construct refers to it as `\k<1>`. As the output from the example shows, the call to the Regex.IsMatch succeeds because `char` is the first capturing group.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      Console.WriteLine(Regex.IsMatch("aa", @"(?<char>\w)\k<1>"));
      // Displays "True".
   }
}
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Console.WriteLine(Regex.IsMatch("aa", "(?<char>\w)\k<1>"))
      ' Displays "True".
   End Sub
End Module
```

However, if *name* is the string representation of a number and the capturing group in that position has been explicitly assigned a numeric name, the regular expression parser cannot identify the capturing group by its ordinal position. Instead, it throws an ArgumentException.The only capturing group in the following example is named "2". Because the `\k` construct is used to define a backreference named "1", the regular expression parser is unable to identify the first capturing group and throws an exception.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      Console.WriteLine(Regex.IsMatch("aa", @"(?<2>\w)\k<1>"));
      // Throws an ArgumentException.
   }
}
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Console.WriteLine(Regex.IsMatch("aa", "(?<2>\w)\k<1>"))
      ' Throws an ArgumentException.
   End Sub
End Module
```

What Backreferences Match

A backreference refers to the most recent definition of a group (the definition most immediately to the left, when matching left to right). When a group makes multiple captures, a backreference refers to the most recent capture.

The following example includes a regular expression pattern, `(?<1>a)(?<1>\1b)*`, which redefines the \1 named group. The following table describes each pattern in the regular expression.

PATTERN	DESCRIPTION
`(?<1>a)`	Match the character "a" and assign the result to the capturing group named `1`.
`(?<1>\1b)*`	Match 0 or 1 occurrence of the group named `1` along with a "b", and assign the result to the capturing group named `1`.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?<1>a)(?<1>\1b)*";
      string input = "aababb";
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine("Match: " + match.Value);
         foreach (Group group in match.Groups)
            Console.WriteLine("   Group: " + group.Value);
      }
   }
}
// The example displays the following output:
//          Group: aababb
//          Group: abb
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(?<1>a)(?<1>\1b)*"
      Dim input As String = "aababb"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("Match: " + match.Value)
         For Each group As Group In match.Groups
            Console.WriteLIne("   Group: " + group.Value)
         Next
      Next
   End Sub
End Module
' The example display the following output:
'          Group: aababb
'          Group: abb
```

In comparing the regular expression with the input string ("aababb"), the regular expression engine performs the following operations:

1. It starts at the beginning of the string, and successfully matches "a" with the expression `(?<1>a)`. The value of the `1` group is now "a".
2. It advances to the second character, and successfully matches the string "ab" with the expression `\1b`, or "ab". It then assigns the result, "ab" to `\1`.
3. It advances to the fourth character. The expression `(?<1>\1b)` is to be matched zero or more times, so it successfully matches the string "abb" with the expression `\1b`. It assigns the result, "abb", back to `\1`.

In this example, `*` is a looping quantifier -- it is evaluated repeatedly until the regular expression engine cannot match the pattern it defines. Looping quantifiers do not clear group definitions.

If a group has not captured any substrings, a backreference to that group is undefined and never matches. This is illustrated by the regular expression pattern `\b(\p{Lu}{2})(\d{2})?(\p{Lu}{2})\b`, which is defined as follows:

PATTERN	DESCRIPTION
`\b`	Begin the match on a word boundary.

PATTERN	DESCRIPTION
`(\p{Lu}{2})`	Match two uppercase letters. This is the first capturing group.
`(\d{2})?`	Match zero or one occurrence of two decimal digits. This is the second capturing group.
`(\p{Lu}{2})`	Match two uppercase letters. This is the third capturing group.
`\b`	End the match on a word boundary.

An input string can match this regular expression even if the two decimal digits that are defined by the second capturing group are not present. The following example shows that even though the match is successful, an empty capturing group is found between two successful capturing groups.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\p{Lu}{2})(\d{2})?(\p{Lu}{2})\b";
      string[] inputs = { "AA22ZZ", "AABB" };
      foreach (string input in inputs)
      {
         Match match = Regex.Match(input, pattern);
         if (match.Success)
         {
            Console.WriteLine("Match in {0}: {1}", input, match.Value);
            if (match.Groups.Count > 1)
            {
               for (int ctr = 1; ctr <= match.Groups.Count - 1; ctr++)
               {
                  if (match.Groups[ctr].Success)
                     Console.WriteLine("Group {0}: {1}",
                                       ctr, match.Groups[ctr].Value);
                  else
                     Console.WriteLine("Group {0}: <no match>", ctr);
               }
            }
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Match in AA22ZZ: AA22ZZ
//       Group 1: AA
//       Group 2: 22
//       Group 3: ZZ
//
//       Match in AABB: AABB
//       Group 1: AA
//       Group 2: <no match>
//       Group 3: BB
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b(\p{Lu}{2})(\d{2})?(\p{Lu}{2})\b"
      Dim inputs() As String = { "AA22ZZ", "AABB" }
      For Each input As String In inputs
         Dim match As Match = Regex.Match(input, pattern)
         If match.Success Then
            Console.WriteLine("Match in {0}: {1}", input, match.Value)
            If match.Groups.Count > 1 Then
               For ctr As Integer = 1 To match.Groups.Count - 1
                  If match.Groups(ctr).Success Then
                     Console.WriteLine("Group {0}: {1}", _
                                       ctr, match.Groups(ctr).Value)
                  Else
                     Console.WriteLine("Group {0}: <no match>", ctr)
                  End If
               Next
            End If
         End If
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays the following output:
'       Match in AA22ZZ: AA22ZZ
'       Group 1: AA
'       Group 2: 22
'       Group 3: ZZ
'
'       Match in AABB: AABB
'       Group 1: AA
'       Group 2: <no match>
'       Group 3: BB
```

See also

- Regular Expression Language - Quick Reference

Alternation Constructs in Regular Expressions

9/6/2018 • 8 minutes to read • Edit Online

Alternation constructs modify a regular expression to enable either/or or conditional matching. .NET supports three alternation constructs:

- Pattern matching with |
- Conditional matching with (?(expression)yes|no)
- Conditional matching based on a valid captured group

Pattern Matching with |

You can use the vertical bar (`|`) character to match any one of a series of patterns, where the `|` character separates each pattern.

Like the positive character class, the `|` character can be used to match any one of a number of single characters. The following example uses both a positive character class and either/or pattern matching with the `|` character to locate occurrences of the words "gray" or "grey" in a string. In this case, the `|` character produces a regular expression that is more verbose.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      // Regular expression using character class.
      string pattern1 = @"\bgr[ae]y\b";
      // Regular expression using either/or.
      string pattern2 = @"\bgr(a|e)y\b";

      string input = "The gray wolf blended in among the grey rocks.";
      foreach (Match match in Regex.Matches(input, pattern1))
         Console.WriteLine("'{0}' found at position {1}",
                           match.Value, match.Index);
      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, pattern2))
         Console.WriteLine("'{0}' found at position {1}",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       'gray' found at position 4
//       'grey' found at position 35
//
//       'gray' found at position 4
//       'grey' found at position 35
```

```vb
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      ' Regular expression using character class.
      Dim pattern1 As String = "\bgr[ae]y\b"
      ' Regular expression using either/or.
      Dim pattern2 As String = "\bgr(a|e)y\b"

      Dim input As String = "The gray wolf blended in among the grey rocks."
      For Each match As Match In Regex.Matches(input, pattern1)
         Console.WriteLine("'{0}' found at position {1}", _
                           match.Value, match.Index)
      Next
      Console.WriteLine()
      For Each match As Match In Regex.Matches(input, pattern2)
         Console.WriteLine("'{0}' found at position {1}", _
                           match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       'gray' found at position 4
'       'grey' found at position 35
'
'       'gray' found at position 4
'       'grey' found at position 35
```

The regular expression that uses the `|` character, `\bgr(a|e)y\b`, is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`gr`	Match the characters "gr".
`(a|e)`	Match either an "a" or an "e".
`y\b`	Match a "y" on a word boundary.

The `|` character can also be used to perform an either/or match with multiple characters or subexpressions, which can include any combination of character literals and regular expression language elements. (The character class does not provide this functionality.) The following example uses the `|` character to extract either a U.S. Social Security Number (SSN), which is a 9-digit number with the format *ddd-dd-dddd*, or a U.S. Employer Identification Number (EIN), which is a 9-digit number with the format *dd-ddddddd*.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b";
      string input = "01-9999999 020-333333 777-88-9999";
      Console.WriteLine("Matches for {0}:", pattern);
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index);
   }
}
// The example displays the following output:
//       Matches for \b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b"
      Dim input As String = "01-9999999 020-333333 777-88-9999"
      Console.WriteLine("Matches for {0}:", pattern)
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       Matches for \b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b:
'          01-9999999 at position 0
'          777-88-9999 at position 22
```

The regular expression `\b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})`	Match either of the following: two decimal digits followed by a hyphen followed by seven decimal digits; or three decimal digits, a hyphen, two decimal digits, another hyphen, and four decimal digits.
`\d`	End the match at a word boundary.

Back to top

Conditional Matching with an Expression

This language element attempts to match one of two patterns depending on whether it can match an initial pattern. Its syntax is:

`(?(` *expression* `)` *yes* `|` *no* `)`

where *expression* is the initial pattern to match, *yes* is the pattern to match if *expression* is matched, and *no* is the optional pattern to match if *expression* is not matched. The regular expression engine treats *expression* as a zero-width assertion; that is, the regular expression engine does not advance in the input stream after it evaluates *expression*. Therefore, this construct is equivalent to the following:

`(?(?=` *expression* `)` *yes* `|` *no* `)`

where `(?=` *expression* `)` is a zero-width assertion construct. (For more information, see Grouping Constructs.) Because the regular expression engine interprets *expression* as an anchor (a zero-width assertion), *expression* must either be a zero-width assertion (for more information, see Anchors) or a subexpression that is also contained in *yes*. Otherwise, the *yes* pattern cannot be matched.

> **NOTE**
>
> If *expression*is a named or numbered capturing group, the alternation construct is interpreted as a capture test; for more information, see the next section, Conditional Matching Based on a Valid Capture Group. In other words, the regular expression engine does not attempt to match the captured substring, but instead tests for the presence or absence of the group.

The following example is a variation of the example that appears in the Either/Or Pattern Matching with | section. It uses conditional matching to determine whether the first three characters after a word boundary are two digits followed by a hyphen. If they are, it attempts to match a U.S. Employer Identification Number (EIN). If not, it attempts to match a U.S. Social Security Number (SSN).

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b";
      string input = "01-9999999 020-333333 777-88-9999";
      Console.WriteLine("Matches for {0}:", pattern);
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index);
   }
}
// The example displays the following output:
//       Matches for \b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b(?(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b"
      Dim input As String = "01-9999999 020-333333 777-88-9999"
      Console.WriteLine("Matches for {0}:", pattern)
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       Matches for \b(?(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b:
'          01-9999999 at position 0
'          777-88-9999 at position 22
```

The regular expression pattern `\b(?(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`(?(\d{2}-)`	Determine whether the next three characters consist of two digits followed by a hyphen.
`\d{2}-\d{7}`	If the previous pattern matches, match two digits followed by a hyphen followed by seven digits.
`\d{3}-\d{2}-\d{4}`	If the previous pattern does not match, match three decimal digits, a hyphen, two decimal digits, another hyphen, and four decimal digits.
`\b`	Match a word boundary.

Back to top

Conditional Matching Based on a Valid Captured Group

This language element attempts to match one of two patterns depending on whether it has matched a specified capturing group. Its syntax is:

`(?(` *name* `)` *yes* `|` *no* `)`

or

`(?(` *number* `)` *yes* `|` *no* `)`

where *name* is the name and *number* is the number of a capturing group, *yes* is the expression to match if *name* or *number* has a match, and *no* is the optional expression to match if it does not.

If *name* does not correspond to the name of a capturing group that is used in the regular expression pattern, the alternation construct is interpreted as an expression test, as explained in the previous section. Typically, this means that *expression* evaluates to `false`. If *number* does not correspond to a numbered capturing group that is used in the regular expression pattern, the regular expression engine throws an ArgumentException.

The following example is a variation of the example that appears in the Either/Or Pattern Matching with | section. It uses a capturing group named `n2` that consists of two digits followed by a hyphen. The alternation construct tests whether this capturing group has been matched in the input string. If it has, the alternation construct attempts to match the last seven digits of a nine-digit U.S. Employer Identification Number (EIN). If it has not, it attempts to match a nine-digit U.S. Social Security Number (SSN).

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?<n2>\d{2}-)?(?(n2)\d{7}|\d{3}-\d{2}-\d{4})\b";
      string input = "01-9999999 020-333333 777-88-9999";
      Console.WriteLine("Matches for {0}:", pattern);
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index);
   }
}
// The example displays the following output:
//       Matches for \b(?<n2>\d{2}-)?(?(n2)\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
        Dim pattern As String = "\b(?<n2>\d{2}-)?(?(n2)\d{7}|\d{3}-\d{2}-\d{4})\b"
        Dim input As String = "01-9999999 020-333333 777-88-9999"
      Console.WriteLine("Matches for {0}:", pattern)
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index)
      Next
   End Sub
End Module
```

The regular expression pattern `\b(?<n2>\d{2}-)?(?(n2)\d{7}|\d{3}-\d{2}-\d{4})\b` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`(?<n2>\d{2}-)?`	Match zero or one occurrence of two digits followed by a hyphen. Name this capturing group `n2`.
`(?(n2)`	Test whether `n2` was matched in the input string.
`)\d{7}`	If `n2` was matched, match seven decimal digits.
`|\d{3}-\d{2}-\d{4}`	If `n2` was not matched, match three decimal digits, a hyphen, two decimal digits, another hyphen, and four decimal digits.
`\b`	Match a word boundary.

A variation of this example that uses a numbered group instead of a named group is shown in the following example. Its regular expression pattern is `\b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b`.

```csharp
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b";
      string input = "01-9999999 020-333333 777-88-9999";
      Console.WriteLine("Matches for {0}:", pattern);
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index);
   }
}
// The example display the following output:
//       Matches for \b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

```vb
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
        Dim pattern As String = "\b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b"
        Dim input As String = "01-9999999 020-333333 777-88-9999"
      Console.WriteLine("Matches for {0}:", pattern)
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       Matches for \b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b:
'          01-9999999 at position 0
'          777-88-9999 at position 22
```

See also

- Regular Expression Language - Quick Reference

Substitutions in Regular Expressions

9/6/2018 • 16 minutes to read • Edit Online

Substitutions are language elements that are recognized only within replacement patterns. They use a regular expression pattern to define all or part of the text that is to replace matched text in the input string. The replacement pattern can consist of one or more substitutions along with literal characters. Replacement patterns are provided to overloads of the Regex.Replace method that have a `replacement` parameter and to the Match.Result method. The methods replace the matched pattern with the pattern that is defined by the `replacement` parameter.

The .NET Framework defines the substitution elements listed in the following table.

SUBSTITUTION	DESCRIPTION
`$` *number*	Includes the last substring matched by the capturing group that is identified by *number*, where *number* is a decimal value, in the replacement string. For more information, see Substituting a Numbered Group.
`${` *name* `}`	Includes the last substring matched by the named group that is designated by `(?<` *name* `>)` in the replacement string. For more information, see Substituting a Named Group.
`$$`	Includes a single "$" literal in the replacement string. For more information, see Substituting a "$" Symbol.
`$&`	Includes a copy of the entire match in the replacement string. For more information, see Substituting the Entire Match.
``$` ``	Includes all the text of the input string before the match in the replacement string. For more information, see Substituting the Text before the Match.
`$'`	Includes all the text of the input string after the match in the replacement string. For more information, see Substituting the Text after the Match.
`$+`	Includes the last group captured in the replacement string. For more information, see Substituting the Last Captured Group.
`$_`	Includes the entire input string in the replacement string. For more information, see Substituting the Entire Input String.

Substitution Elements and Replacement Patterns

Substitutions are the only special constructs recognized in a replacement pattern. None of the other regular expression language elements, including character escapes and the period (`.`), which matches any character, are supported. Similarly, substitution language elements are recognized only in replacement patterns and are never valid in regular expression patterns.

The only character that can appear either in a regular expression pattern or in a substitution is the `$` character, although it has a different meaning in each context. In a regular expression pattern, `$` is an anchor that matches the end of the string. In a replacement pattern, `$` indicates the beginning of a substitution.

NOTE

For functionality similar to a replacement pattern within a regular expression, use a backreference. For more information about backreferences, see Backreference Constructs.

Substituting a Numbered Group

The `$` *number* language element includes the last substring matched by the *number* capturing group in the replacement string, where *number* is the index of the capturing group. For example, the replacement pattern `$1` indicates that the matched substring is to be replaced by the first captured group. For more information about numbered capturing groups, see Grouping Constructs.

All digits that follow `$` are interpreted as belonging to the *number* group. If this is not your intent, you can substitute a named group instead. For example, you can use the replacement string `${1}1` instead of `$11` to define the replacement string as the value of the first captured group along with the number "1". For more information, see Substituting a Named Group.

Capturing groups that are not explicitly assigned names using the `(?<` *name* `>)` syntax are numbered from left to right starting at one. Named groups are also numbered from left to right, starting at one greater than the index of the last unnamed group. For example, in the regular expression `(\w)(?<digit>\d)`, the index of the `digit` named group is 2.

If *number* does not specify a valid capturing group defined in the regular expression pattern, `$` *number* is interpreted as a literal character sequence that is used to replace each match.

The following example uses the `$` *number* substitution to strip the currency symbol from a decimal value. It removes currency symbols found at the beginning or end of a monetary value, and recognizes the two most common decimal separators ("." and ",").

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\p{Sc}*(\s?\d+[.,]?\d*)\p{Sc}*";
      string replacement = "$1";
      string input = "$16.32 12.19 £16.29 €18.29  €18,29";
      string result = Regex.Replace(input, pattern, replacement);
      Console.WriteLine(result);
   }
}
// The example displays the following output:
//       16.32 12.19 16.29 18.29  18,29
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\p{Sc}*(\s?\d+[.,]?\d*)\p{Sc}*"
      Dim replacement As String = "$1"
      Dim input As String = "$16.32 12.19 £16.29 €18.29  €18,29"
      Dim result As String = Regex.Replace(input, pattern, replacement)
      Console.WriteLine(result)
   End Sub
End Module
' The example displays the following output:
'       16.32 12.19 16.29 18.29  18,29
```

The regular expression pattern `\p{Sc}*(\s?\d+[.,]?\d*)\p{Sc}*` is defined as shown in the following table.

PATTERN	DESCRIPTION
`\p{Sc}*`	Match zero or more currency symbol characters.
`\s?`	Match zero or one white-space characters.
`\d+`	Match one or more decimal digits.
`[.,]?`	Match zero or one period or comma.
`\d*`	Match zero or more decimal digits.
`(\s?\d+[.,]?\d*)`	Match a white space followed by one or more decimal digits, followed by zero or one period or comma, followed by zero or more decimal digits. This is the first capturing group. Because the replacement pattern is `$1`, the call to the Regex.Replace method replaces the entire matched substring with this captured group.

Back to top

Substituting a Named Group

The `${` *name* `}` language element substitutes the last substring matched by the *name* capturing group, where *name* is the name of a capturing group defined by the `(?<` *name* `>)` language element. For more information about named capturing groups, see Grouping Constructs.

If *name* doesn't specify a valid named capturing group defined in the regular expression pattern but consists of digits, `${` *name* `}` is interpreted as a numbered group.

If *name* specifies neither a valid named capturing group nor a valid numbered capturing group defined in the regular expression pattern, `${` *name* `}` is interpreted as a literal character sequence that is used to replace each match.

The following example uses the `${` *name* `}` substitution to strip the currency symbol from a decimal value. It removes currency symbols found at the beginning or end of a monetary value, and recognizes the two most common decimal separators ("." and ",").

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\p{Sc}*(?<amount>\s?\d+[.,]?\d*)\p{Sc}*";
      string replacement = "${amount}";
      string input = "$16.32 12.19 £16.29 €18.29  €18,29";
      string result = Regex.Replace(input, pattern, replacement);
      Console.WriteLine(result);
   }
}
// The example displays the following output:
//       16.32 12.19 16.29 18.29  18,29
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\p{Sc}*(?<amount>\s?\d+[.,]?\d*)\p{Sc}*"
      Dim replacement As String = "${amount}"
      Dim input As String = "$16.32 12.19 £16.29 €18.29  €18,29"
      Dim result As String = Regex.Replace(input, pattern, replacement)
      Console.WriteLine(result)
   End Sub
End Module
' The example displays the following output:
'       16.32 12.19 16.29 18.29  18,29
```

The regular expression pattern `\p{Sc}*(?<amount>\s?\d[.,]?\d*)\p{Sc}*` is defined as shown in the following table.

PATTERN	DESCRIPTION
`\p{Sc}*`	Match zero or more currency symbol characters.
`\s?`	Match zero or one white-space characters.
`\d+`	Match one or more decimal digits.
`[.,]?`	Match zero or one period or comma.
`\d*`	Match zero or more decimal digits.
`(?<amount>\s?\d[.,]?\d*)`	Match a white space, followed by one or more decimal digits, followed by zero or one period or comma, followed by zero or more decimal digits. This is the capturing group named `amount`. Because the replacement pattern is `${amount}`, the call to the Regex.Replace method replaces the entire matched substring with this captured group.

Back to top

Substituting a "$" Character

The `$$` substitution inserts a literal "$" character in the replaced string.

The following example uses the NumberFormatInfo object to determine the current culture's currency symbol and its placement in a currency string. It then builds both a regular expression pattern and a replacement pattern dynamically. If the example is run on a computer whose current culture is en-US, it generates the regular expression pattern `\b(\d+)(\.(\d+))?` and the replacement pattern `$$ $1$2`. The replacement pattern replaces the matched text with a currency symbol and a space followed by the first and second captured groups.

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      // Define array of decimal values.
      string[] values= { "16.35", "19.72", "1234", "0.99"};
      // Determine whether currency precedes (True) or follows (False) number.
      bool precedes = NumberFormatInfo.CurrentInfo.CurrencyPositivePattern % 2 == 0;
      // Get decimal separator.
      string cSeparator = NumberFormatInfo.CurrentInfo.CurrencyDecimalSeparator;
      // Get currency symbol.
      string symbol = NumberFormatInfo.CurrentInfo.CurrencySymbol;
      // If symbol is a "$", add an extra "$".
      if (symbol == "$") symbol = "$$";

      // Define regular expression pattern and replacement string.
      string pattern = @"\b(\d+)(" + cSeparator + @"(\d+))?";
      string replacement = "$1$2";
      replacement = precedes ? symbol + " " + replacement : replacement + " " + symbol;
      foreach (string value in values)
         Console.WriteLine("{0} --> {1}", value, Regex.Replace(value, pattern, replacement));
   }
}
// The example displays the following output:
//       16.35 --> $ 16.35
//       19.72 --> $ 19.72
//       1234 --> $ 1234
//       0.99 --> $ 0.99
```

```
Imports System.Globalization
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      ' Define array of decimal values.
      Dim values() As String = { "16.35", "19.72", "1234", "0.99"}
      ' Determine whether currency precedes (True) or follows (False) number.
      Dim precedes As Boolean = (NumberFormatInfo.CurrentInfo.CurrencyPositivePattern Mod 2 = 0)
      ' Get decimal separator.
      Dim cSeparator As String = NumberFormatInfo.CurrentInfo.CurrencyDecimalSeparator
      ' Get currency symbol.
      Dim symbol As String = NumberFormatInfo.CurrentInfo.CurrencySymbol
      ' If symbol is a "$", add an extra "$".
      If symbol = "$" Then symbol = "$$"

      ' Define regular expression pattern and replacement string.
      Dim pattern As String = "\b(\d+)(" + cSeparator + "(\d+))?"
      Dim replacement As String = "$1$2"
      replacement = If(precedes, symbol + " " + replacement, replacement + " " + symbol)
      For Each value In values
         Console.WriteLine("{0} --> {1}", value, Regex.Replace(value, pattern, replacement))
      Next
   End Sub
End Module
' The example displays the following output:
'       16.35 --> $ 16.35
'       19.72 --> $ 19.72
'       1234 --> $ 1234
'       0.99 --> $ 0.99
```

The regular expression pattern `\b(\d+)(\.(\d+))?` is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Start the match at the beginning of a word boundary.
`(\d+)`	Match one or more decimal digits. This is the first capturing group.
`\.`	Match a period (the decimal separator).
`(\d+)`	Match one or more decimal digits. This is the third capturing group.
`(\.(\d+))?`	Match zero or one occurrence of a period followed by one or more decimal digits. This is the second capturing group.

Substituting the Entire Match

The `$&` substitution includes the entire match in the replacement string. Often, it is used to add a substring to the beginning or end of the matched string. For example, the `($&)` replacement pattern adds parentheses to the beginning and end of each match. If there is no match, the `$&` substitution has no effect.

The following example uses the `$&` substitution to add quotation marks at the beginning and end of book titles stored in a string array.

```csharp
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"^(\w+\s?)+$";
      string[] titles = { "A Tale of Two Cities",
                          "The Hound of the Baskervilles",
                          "The Protestant Ethic and the Spirit of Capitalism",
                          "The Origin of Species" };
      string replacement = "\"$&\"";
      foreach (string title in titles)
         Console.WriteLine(Regex.Replace(title, pattern, replacement));
   }
}
// The example displays the following output:
//       "A Tale of Two Cities"
//       "The Hound of the Baskervilles"
//       "The Protestant Ethic and the Spirit of Capitalism"
//       "The Origin of Species"
```

```vb
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "^(\w+\s?)+$"
      Dim titles() As String = { "A Tale of Two Cities", _
                                 "The Hound of the Baskervilles", _
                                 "The Protestant Ethic and the Spirit of Capitalism", _
                                 "The Origin of Species" }
      Dim replacement As String = """$&"""
      For Each title As String In titles
         Console.WriteLine(Regex.Replace(title, pattern, replacement))
      Next
   End Sub
End Module
' The example displays the following output:
'       "A Tale of Two Cities"
'       "The Hound of the Baskervilles"
'       "The Protestant Ethic and the Spirit of Capitalism"
'       "The Origin of Species"
```

The regular expression pattern `^(\w+\s?)+$` is defined as shown in the following table.

PATTERN	DESCRIPTION
`^`	Start the match at the beginning of the input string.
`(\w+\s?)+`	Match the pattern of one or more word characters followed by zero or one white-space characters one or more times.
`$`	Match the end of the input string.

The `"$&"` replacement pattern adds a literal quotation mark to the beginning and end of each match.

Back to top

Substituting the Text Before the Match

The `$`` substitution replaces the matched string with the entire input string before the match. That is, it duplicates the input string up to the match while removing the matched text. Any text that follows the matched text is unchanged in the result string. If there are multiple matches in an input string, the replacement text is derived from the original input string, rather than from the string in which text has been replaced by earlier matches. (The example provides an illustration.) If there is no match, the `$`` substitution has no effect.

The following example uses the regular expression pattern `\d+` to match a sequence of one or more decimal digits in the input string. The replacement string `$`` replaces these digits with the text that precedes the match.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "aa1bb2cc3dd4ee5";
      string pattern = @"\d+";
      string substitution = "$`";
      Console.WriteLine("Matches:");
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index);

      Console.WriteLine("Input string:  {0}", input);
      Console.WriteLine("Output string: " +
                        Regex.Replace(input, pattern, substitution));
   }
}
// The example displays the following output:
//    Matches:
//       1 at position 2
//       2 at position 5
//       3 at position 8
//       4 at position 11
//       5 at position 14
//    Input string:  aa1bb2cc3dd4ee5
//    Output string: aaaabbaa1bbccaa1bb2ccddaa1bb2cc3ddeeaa1bb2cc3dd4ee
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "aa1bb2cc3dd4ee5"
      Dim pattern As String = "\d+"
      Dim substitution As String = "$`"
      Console.WriteLine("Matches:")
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index)
      Next
      Console.WriteLine("Input string:  {0}", input)
      Console.WriteLine("Output string: " + _
                        Regex.Replace(input, pattern, substitution))
   End Sub
End Module
' The example displays the following output:
'    Matches:
'       1 at position 2
'       2 at position 5
'       3 at position 8
'       4 at position 11
'       5 at position 14
'    Input string:  aa1bb2cc3dd4ee5
'    Output string: aaaabbaa1bbccaa1bb2ccddaa1bb2cc3ddeeaa1bb2cc3dd4ee
```

In this example, the input string `"aa1bb2cc3dd4ee5"` contains five matches. The following table illustrates how the `` $` `` substitution causes the regular expression engine to replace each match in the input string. Inserted text is shown in bold in the results column.

MATCH	POSITION	STRING BEFORE MATCH	RESULT STRING
1	2	aa	aa**aa**bb2cc3dd4ee5
2	5	aa1bb	aaaabb**aa1bb**cc3dd4ee5
3	8	aa1bb2cc	aaaabbaa1bbcc**aa1bb2cc**dd4ee5
4	11	aa1bb2cc3dd	aaaabbaa1bbccaa1bb2ccdd**aa1bb2cc3dd**ee5
5	14	aa1bb2cc3dd4ee	aaaabbaa1bbccaa1bb2ccddaa1bb2cc3ddee**aa1bb2cc3dd4ee**

Back to top

Substituting the Text After the Match

The `$'` substitution replaces the matched string with the entire input string after the match. That is, it duplicates the input string after the match while removing the matched text. Any text that precedes the matched text is unchanged in the result string. If there is no match, the `$'` substitution has no effect.

The following example uses the regular expression pattern `\d+` to match a sequence of one or more decimal digits in the input string. The replacement string `$'` replaces these digits with the text that follows the match.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "aa1bb2cc3dd4ee5";
      string pattern = @"\d+";
      string substitution = "$'";
      Console.WriteLine("Matches:");
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index);
      Console.WriteLine("Input string:  {0}", input);
      Console.WriteLine("Output string: " +
                        Regex.Replace(input, pattern, substitution));
   }
}
// The example displays the following output:
//    Matches:
//       1 at position 2
//       2 at position 5
//       3 at position 8
//       4 at position 11
//       5 at position 14
//    Input string:  aa1bb2cc3dd4ee5
//    Output string: aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5ddee5ee
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "aa1bb2cc3dd4ee5"
      Dim pattern As String = "\d+"
      Dim substitution As String = "$'"
      Console.WriteLine("Matches:")
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index)
      Next
      Console.WriteLine("Input string:  {0}", input)
      Console.WriteLine("Output string: " + _
                        Regex.Replace(input, pattern, substitution))
   End Sub
End Module
' The example displays the following output:
'    Matches:
'       1 at position 2
'       2 at position 5
'       3 at position 8
'       4 at position 11
'       5 at position 14
'    Input string:  aa1bb2cc3dd4ee5
'    Output string: aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5ddee5ee
```

In this example, the input string `"aa1bb2cc3dd4ee5"` contains five matches. The following table illustrates how the `$'` substitution causes the regular expression engine to replace each match in the input string. Inserted text is shown in bold in the results column.

MATCH	POSITION	STRING AFTER MATCH	RESULT STRING
1	2	bb2cc3dd4ee5	aa**bb2cc3dd4ee5**bb2cc3dd4ee5
2	5	cc3dd4ee5	aabb2cc3dd4ee5bb**cc3dd4ee5**cc3dd4ee5
3	8	dd4ee5	aabb2cc3dd4ee5bbcc3dd4ee5cc**dd4ee5**dd4ee5
4	11	ee5	aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5dd**ee5**ee5
5	14	String.Empty	aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5ddee5ee

Back to top

Substituting the Last Captured Group

The `$+` substitution replaces the matched string with the last captured group. If there are no captured groups or if the value of the last captured group is String.Empty, the `$+` substitution has no effect.

The following example identifies duplicate words in a string and uses the `$+` substitution to replace them with a single occurrence of the word. The RegexOptions.IgnoreCase option is used to ensure that words that differ in case but that are otherwise identical are considered duplicates.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+)\s\1\b";
      string substitution = "$+";
      string input = "The the dog jumped over the fence fence.";
      Console.WriteLine(Regex.Replace(input, pattern, substitution,
                        RegexOptions.IgnoreCase));
   }
}
// The example displays the following output:
//      The dog jumped over the fence.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b(\w+)\s\1\b"
      Dim substitution As String = "$+"
      Dim input As String = "The the dog jumped over the fence fence."
      Console.WriteLine(Regex.Replace(input, pattern, substitution, _
                                      RegexOptions.IgnoreCase))
   End Sub
End Module
' The example displays the following output:
'      The dog jumped over the fence.
```

The regular expression pattern `\b(\w+)\s\1\b` is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`(\w+)`	Match one or more word characters. This is the first capturing group.
`\s`	Match a white-space character.
`\1`	Match the first captured group.
`\b`	End the match at a word boundary.

Back to top

Substituting the Entire Input String

The `$_` substitution replaces the matched string with the entire input string. That is, it removes the matched text and replaces it with the entire string, including the matched text.

The following example matches one or more decimal digits in the input string. It uses the `$_` substitution to replace them with the entire input string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "ABC123DEF456";
      string pattern = @"\d+";
      string substitution = "$_";
      Console.WriteLine("Original string:          {0}", input);
      Console.WriteLine("String with substitution: {0}",
                        Regex.Replace(input, pattern, substitution));
   }
}
// The example displays the following output:
//       Original string:          ABC123DEF456
//       String with substitution: ABCABC123DEF456DEFABC123DEF456
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "ABC123DEF456"
      Dim pattern As String = "\d+"
      Dim substitution As String = "$_"
      Console.WriteLine("Original string:          {0}", input)
      Console.WriteLine("String with substitution: {0}", _
                        Regex.Replace(input, pattern, substitution))
   End Sub
End Module
' The example displays the following output:
'       Original string:          ABC123DEF456
'       String with substitution: ABCABC123DEF456DEFABC123DEF456
```

In this example, the input string `"ABC123DEF456"` contains two matches. The following table illustrates how the `$_` substitution causes the regular expression engine to replace each match in the input string. Inserted text is shown in bold in the results column.

MATCH	POSITION	MATCH	RESULT STRING
1	3	123	ABC**ABC123DEF456**DEF456
2	5	456	ABCABC123DEF456DEF**ABC123DEF456**

See also

- Regular Expression Language - Quick Reference

Regular Expression Options

9/6/2018 • 46 minutes to read • Edit Online

By default, the comparison of an input string with any literal characters in a regular expression pattern is case sensitive, white space in a regular expression pattern is interpreted as literal white-space characters, and capturing groups in a regular expression are named implicitly as well as explicitly. You can modify these and several other aspects of default regular expression behavior by specifying regular expression options. These options, which are listed in the following table, can be included inline as part of the regular expression pattern, or they can be supplied to a System.Text.RegularExpressions.Regex class constructor or static pattern matching method as a System.Text.RegularExpressions.RegexOptions enumeration value.

REGEXOPTIONS MEMBER	INLINE CHARACTER	EFFECT
None	Not available	Use default behavior. For more information, see Default Options.
IgnoreCase	`i`	Use case-insensitive matching. For more information, see Case-Insensitive Matching.
Multiline	`m`	Use multiline mode, where `^` and `$` match the beginning and end of each line (instead of the beginning and end of the input string). For more information, see Multiline Mode.
Singleline	`s`	Use single-line mode, where the period (.) matches every character (instead of every character except `\n`). For more information, see Singleline Mode.
ExplicitCapture	`n`	Do not capture unnamed groups. The only valid captures are explicitly named or numbered groups of the form `(?<` *name* `>` *subexpression* `)`. For more information, see Explicit Captures Only.
Compiled	Not available	Compile the regular expression to an assembly. For more information, see Compiled Regular Expressions.
IgnorePatternWhitespace	`x`	Exclude unescaped white space from the pattern, and enable comments after a number sign (`#`). For more information, see Ignore White Space.
RightToLeft	Not available	Change the search direction. Search moves from right to left instead of from left to right. For more information, see Right-to-Left Mode.

REGEXOPTIONS MEMBER	INLINE CHARACTER	EFFECT
ECMAScript	Not available	Enable ECMAScript-compliant behavior for the expression. For more information, see ECMAScript Matching Behavior.
CultureInvariant	Not available	Ignore cultural differences in language. For more information, see Comparison Using the Invariant Culture.

Specifying the Options

You can specify options for regular expressions in one of three ways:

- In the `options` parameter of a System.Text.RegularExpressions.Regex class constructor or static (`Shared` in Visual Basic) pattern-matching method, such as Regex.Regex(String, RegexOptions) or Regex.Match(String, String, RegexOptions). The `options` parameter is a bitwise OR combination of System.Text.RegularExpressions.RegexOptions enumerated values.

  When options are supplied to a Regex instance by using the `options` parameter of a class constructor, the options are are assigned to the System.Text.RegularExpressions.RegexOptions property. However, the System.Text.RegularExpressions.RegexOptions property does not reflect inline options in the regular expression pattern itself.

  The following example provides an illustration. It uses the `options` parameter of the Regex.Match(String, String, RegexOptions) method to enable case-insensitive matching and to ignore pattern white space when identifying words that begin with the letter "d".

  ```
  string pattern = @"d \w+ \s";
  string input = "Dogs are decidedly good pets.";
  RegexOptions options = RegexOptions.IgnoreCase | RegexOptions.IgnorePatternWhitespace;

  foreach (Match match in Regex.Matches(input, pattern, options))
     Console.WriteLine("'{0}// found at index {1}.", match.Value, match.Index);
  // The example displays the following output:
  //    'Dogs // found at index 0.
  //    'decidedly // found at index 9.
  ```

  ```
  Dim pattern As String = "d \w+ \s"
  Dim input As String = "Dogs are decidedly good pets."
  Dim options As RegexOptions = RegexOptions.IgnoreCase Or RegexOptions.IgnorePatternWhitespace

  For Each match As Match In Regex.Matches(input, pattern, options)
     Console.WriteLine("'{0}' found at index {1}.", match.Value, match.Index)
  Next
  ' The example displays the following output:
  '    'Dogs ' found at index 0.
  '    'decidedly ' found at index 9.
  ```

- By applying inline options in a regular expression pattern with the syntax `(?imnsx-imnsx)`. The option applies to the pattern from the point that the option is defined to either the end of the pattern or to the point at which the option is undefined by another inline option. Note that the System.Text.RegularExpressions.RegexOptions property of a Regex instance does not reflect these inline options. For more information, see the Miscellaneous Constructs topic.

The following example provides an illustration. It uses inline options to enable case-insensitive matching and to ignore pattern white space when identifying words that begin with the letter "d".

```
string pattern = @"(?ix) d \w+ \s";
string input = "Dogs are decidedly good pets.";

foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}// found at index {1}.", match.Value, match.Index);
// The example displays the following output:
//    'Dogs // found at index 0.
//    'decidedly // found at index 9.
```

```
Dim pattern As String = "\b(?ix) d \w+ \s"
Dim input As String = "Dogs are decidedly good pets."

For Each match As Match In Regex.Matches(input, pattern)
   Console.WriteLine("'{0}' found at index {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'    'Dogs ' found at index 0.
'    'decidedly ' found at index 9.
```

- By applying inline options in a particular grouping construct in a regular expression pattern with the syntax `(?imnsx-imnsx:` *subexpression* `)`. No sign before a set of options turns the set on; a minus sign before a set of options turns the set off. (`?` is a fixed part of the language construct's syntax that is required whether options are enabled or disabled.) The option applies only to that group. For more information, see Grouping Constructs.

  The following example provides an illustration. It uses inline options in a grouping construct to enable case-insensitive matching and to ignore pattern white space when identifying words that begin with the letter "d".

```
string pattern = @"\b(?ix: d \w+)\s";
string input = "Dogs are decidedly good pets.";

foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}// found at index {1}.", match.Value, match.Index);
// The example displays the following output:
//    'Dogs // found at index 0.
//    'decidedly // found at index 9.
```

```
Dim pattern As String = "\b(?ix: d \w+)\s"
Dim input As String = "Dogs are decidedly good pets."

For Each match As Match In Regex.Matches(input, pattern)
   Console.WriteLine("'{0}' found at index {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'    'Dogs ' found at index 0.
'    'decidedly ' found at index 9.
```

If options are specified inline, a minus sign (`-`) before an option or set of options turns off those options. For example, the inline construct `(?ix-ms)` turns on the RegexOptions.IgnoreCase and RegexOptions.IgnorePatternWhitespace options and turns off the RegexOptions.Multiline and RegexOptions.Singleline options. All regular expression options are turned off by default.

> **NOTE**
>
> If the regular expression options specified in the `options` parameter of a constructor or method call conflict with the options specified inline in a regular expression pattern, the inline options are used.

The following five regular expression options can be set both with the options parameter and inline:

- RegexOptions.IgnoreCase
- RegexOptions.Multiline
- RegexOptions.Singleline
- RegexOptions.ExplicitCapture
- RegexOptions.IgnorePatternWhitespace

The following five regular expression options can be set using the `options` parameter but cannot be set inline:

- RegexOptions.None
- RegexOptions.Compiled
- RegexOptions.RightToLeft
- RegexOptions.CultureInvariant
- RegexOptions.ECMAScript

Determining the Options

You can determine which options were provided to a Regex object when it was instantiated by retrieving the value of the read-only Regex.Options property. This property is particularly useful for determining the options that are defined for a compiled regular expression created by the Regex.CompileToAssembly method.

To test for the presence of any option except RegexOptions.None, perform an AND operation with the value of the Regex.Options property and the RegexOptions value in which you are interested. Then test whether the result equals that RegexOptions value. The following example tests whether the RegexOptions.IgnoreCase option has been set.

```
if ((rgx.Options & RegexOptions.IgnoreCase) == RegexOptions.IgnoreCase)
   Console.WriteLine("Case-insensitive pattern comparison.");
else
   Console.WriteLine("Case-sensitive pattern comparison.");
```

```
If (rgx.Options And RegexOptions.IgnoreCase) = RegexOptions.IgnoreCase Then
   Console.WriteLine("Case-insensitive pattern comparison.")
Else
   Console.WriteLine("Case-sensitive pattern comparison.")
End If
```

To test for RegexOptions.None, determine whether the value of the Regex.Options property is equal to RegexOptions.None, as the following example illustrates.

```
if (rgx.Options == RegexOptions.None)
   Console.WriteLine("No options have been set.");
```

```
If rgx.Options = RegexOptions.None Then
   Console.WriteLine("No options have been set.")
End If
```

The following sections list the options supported by regular expression in .NET.

Default Options

The RegexOptions.None option indicates that no options have been specified, and the regular expression engine uses its default behavior. This includes the following:

- The pattern is interpreted as a canonical rather than an ECMAScript regular expression.
- The regular expression pattern is matched in the input string from left to right.
- Comparisons are case-sensitive.
- The `^` and `$` language elements match the beginning and end of the input string.
- The `.` language element matches every character except `\n`.
- Any white space in a regular expression pattern is interpreted as a literal space character.
- The conventions of the current culture are used when comparing the pattern to the input string.
- Capturing groups in the regular expression pattern are implicit as well as explicit.

> **NOTE**
>
> The RegexOptions.None option has no inline equivalent. When regular expression options are applied inline, the default behavior is restored on an option-by-option basis, by turning a particular option off. For example, `(?i)` turns on case-insensitive comparison, and `(?-i)` restores the default case-sensitive comparison.

Because the RegexOptions.None option represents the default behavior of the regular expression engine, it is rarely explicitly specified in a method call. A constructor or static pattern-matching method without an `options` parameter is called instead.

Back to Top

Case-Insensitive Matching

The IgnoreCase option, or the `i` inline option, provides case-insensitive matching. By default, the casing conventions of the current culture are used.

The following example defines a regular expression pattern, `\bthe\w*\b`, that matches all words starting with "the". Because the first call to the Match method uses the default case-sensitive comparison, the output indicates that the string "The" that begins the sentence is not matched. It is matched when the Match method is called with options set to IgnoreCase.

```csharp
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\bthe\w*\b";
      string input = "The man then told them about that event.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index);

      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, pattern,
                                            RegexOptions.IgnoreCase))
         Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index);
   }
}
// The example displays the following output:
//       Found then at index 8.
//       Found them at index 18.
//
//       Found The at index 0.
//       Found then at index 8.
//       Found them at index 18.
```

```vb
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\bthe\w*\b"
      Dim input As String = "The man then told them about that event."
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index)
      Next
      Console.WriteLine()
      For Each match As Match In Regex.Matches(input, pattern, _
                                               RegexOptions.IgnoreCase)
         Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       Found then at index 8.
'       Found them at index 18.
'
'       Found The at index 0.
'       Found then at index 8.
'       Found them at index 18.
```

The following example modifies the regular expression pattern from the previous example to use inline options instead of the `options` parameter to provide case-insensitive comparison. The first pattern defines the case-insensitive option in a grouping construct that applies only to the letter "t" in the string "the". Because the option construct occurs at the beginning of the pattern, the second pattern applies the case-insensitive option to the entire regular expression.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?i:t)he\w*\b";
      string input = "The man then told them about that event.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index);

      Console.WriteLine();
      pattern = @"(?i)\bthe\w*\b";
      foreach (Match match in Regex.Matches(input, pattern,
                                            RegexOptions.IgnoreCase))
         Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index);
   }
}
// The example displays the following output:
//       Found The at index 0.
//       Found then at index 8.
//       Found them at index 18.
//
//       Found The at index 0.
//       Found then at index 8.
//       Found them at index 18.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b(?i:t)he\w*\b"
      Dim input As String = "The man then told them about that event."
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index)
      Next
      Console.WriteLine()
      pattern = "(?i)\bthe\w*\b"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       Found The at index 0.
'       Found then at index 8.
'       Found them at index 18.
'
'       Found The at index 0.
'       Found then at index 8.
'       Found them at index 18.
```

Back to Top

Multiline Mode

The RegexOptions.Multiline option, or the `m` inline option, enables the regular expression engine to handle an input string that consists of multiple lines. It changes the interpretation of the `^` and `$` language elements so that they match the beginning and end of a line, instead of the beginning and end of the input string.

By default, `$` matches only the end of the input string. If you specify the RegexOptions.Multiline option, it matches either the newline character (`\n`) or the end of the input string. It does not, however, match the carriage return/line feed character combination. To successfully match them, use the subexpression `\r?$` instead of just `$` .

The following example extracts bowlers' names and scores and adds them to a SortedList<TKey,TValue> collection that sorts them in descending order. The Matches method is called twice. In the first method call, the regular expression is `^(\w+)\s(\d+)$` and no options are set. As the output shows, because the regular expression engine cannot match the input pattern along with the beginning and end of the input string, no matches are found. In the second method call, the regular expression is changed to `^(\w+)\s(\d+)\r?$` and the options are set to RegexOptions.Multiline. As the output shows, the names and scores are successfully matched, and the scores are displayed in descending order.

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      SortedList<int, string> scores = new SortedList<int, string>(new DescendingComparer<int>());

      string input = "Joe 164\n" +
                     "Sam 208\n" +
                     "Allison 211\n" +
                     "Gwen 171\n";
      string pattern = @"^(\w+)\s(\d+)$";
      bool matched = false;

      Console.WriteLine("Without Multiline option:");
      foreach (Match match in Regex.Matches(input, pattern))
      {
         scores.Add(Int32.Parse(match.Groups[2].Value), (string) match.Groups[1].Value);
         matched = true;
      }
      if (! matched)
         Console.WriteLine("   No matches.");
      Console.WriteLine();

      // Redefine pattern to handle multiple lines.
      pattern = @"^(\w+)\s(\d+)\r*$";
      Console.WriteLine("With multiline option:");
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Multiline))
         scores.Add(Int32.Parse(match.Groups[2].Value), (string) match.Groups[1].Value);

      // List scores in descending order.
      foreach (KeyValuePair<int, string> score in scores)
         Console.WriteLine("{0}: {1}", score.Value, score.Key);
   }
}

public class DescendingComparer<T> : IComparer<T>
{
   public int Compare(T x, T y)
   {
      return Comparer<T>.Default.Compare(x, y) * -1;
   }
}
// The example displays the following output:
//   Without Multiline option:
//      No matches.
//
//   With multiline option:
//   Allison: 211
//   Sam: 208
//   Gwen: 171
//   Joe: 164
```

```vb
Imports System.Collections.Generic
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim scores As New SortedList(Of Integer, String)(New DescendingComparer(Of Integer)())

      Dim input As String = "Joe 164" + vbCrLf + _
                            "Sam 208" + vbCrLf + _
                            "Allison 211" + vbCrLf + _
                            "Gwen 171" + vbCrLf
      Dim pattern As String = "^(\w+)\s(\d+)$"
      Dim matched As Boolean = False

      Console.WriteLine("Without Multiline option:")
      For Each match As Match In Regex.Matches(input, pattern)
         scores.Add(CInt(match.Groups(2).Value), match.Groups(1).Value)
         matched = True
      Next
      If Not matched Then Console.WriteLine("   No matches.")
      Console.WriteLine()

      ' Redefine pattern to handle multiple lines.
      pattern = "^(\w+)\s(\d+)\r*$"
      Console.WriteLine("With multiline option:")
      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.Multiline)
         scores.Add(CInt(match.Groups(2).Value), match.Groups(1).Value)
      Next
      ' List scores in descending order.
      For Each score As KeyValuePair(Of Integer, String) In scores
         Console.WriteLine("{0}: {1}", score.Value, score.Key)
      Next
   End Sub
End Module

Public Class DescendingComparer(Of T) : Implements IComparer(Of T)
   Public Function Compare(x As T, y As T) As Integer _
          Implements IComparer(Of T).Compare
      Return Comparer(Of T).Default.Compare(x, y) * -1
   End Function
End Class
' The example displays the following output:
'    Without Multiline option:
'       No matches.
'
'    With multiline option:
'    Allison: 211
'    Sam: 208
'    Gwen: 171
'    Joe: 164
```

The regular expression pattern `^(\w+)\s(\d+)\r*$` is defined as shown in the following table.

PATTERN	DESCRIPTION
`^`	Begin at the start of the line.
`(\w+)`	Match one or more word characters. This is the first capturing group.
`\s`	Match a white-space character.

PATTERN	DESCRIPTION
`(\d+)`	Match one or more decimal digits. This is the second capturing group.
`\r?`	Match zero or one carriage return character.
`$`	End at the end of the line.

The following example is equivalent to the previous one, except that it uses the inline option `(?m)` to set the multiline option.

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      SortedList<int, string> scores = new SortedList<int, string>(new DescendingComparer<int>());

      string input = "Joe 164\n" +
                     "Sam 208\n" +
                     "Allison 211\n" +
                     "Gwen 171\n";
      string pattern = @"(?m)^(\w+)\s(\d+)\r*$";

      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Multiline))
         scores.Add(Convert.ToInt32(match.Groups[2].Value), match.Groups[1].Value);

      // List scores in descending order.
      foreach (KeyValuePair<int, string> score in scores)
         Console.WriteLine("{0}: {1}", score.Value, score.Key);
   }
}

public class DescendingComparer<T> : IComparer<T>
{
   public int Compare(T x, T y)
   {
      return Comparer<T>.Default.Compare(x, y) * -1;
   }
}
// The example displays the following output:
//    Allison: 211
//    Sam: 208
//    Gwen: 171
//    Joe: 164
```

```
Imports System.Collections.Generic
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim scores As New SortedList(Of Integer, String)(New DescendingComparer(Of Integer)())

      Dim input As String = "Joe 164" + vbCrLf + _
                            "Sam 208" + vbCrLf + _
                            "Allison 211" + vbCrLf + _
                            "Gwen 171" + vbCrLf
      Dim pattern As String = "(?m)^(\w+)\s(\d+)\r*$"

      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.Multiline)
         scores.Add(CInt(match.Groups(2).Value), match.Groups(1).Value)
      Next
      ' List scores in descending order.
      For Each score As KeyValuePair(Of Integer, String) In scores
         Console.WriteLine("{0}: {1}", score.Value, score.Key)
      Next
   End Sub
End Module

Public Class DescendingComparer(Of T) : Implements IComparer(Of T)
   Public Function Compare(x As T, y As T) As Integer _
          Implements IComparer(Of T).Compare
      Return Comparer(Of T).Default.Compare(x, y) * -1
   End Function
End Class
' The example displays the following output:
'    Allison: 211
'    Sam: 208
'    Gwen: 171
'    Joe: 164
```

Back to Top

Single-line Mode

The RegexOptions.Singleline option, or the `s` inline option, causes the regular expression engine to treat the input string as if it consists of a single line. It does this by changing the behavior of the period (`.`) language element so that it matches every character, instead of matching every character except for the newline character `\n` or \u000A.

The following example illustrates how the behavior of the `.` language element changes when you use the RegexOptions.Singleline option. The regular expression `^.+` starts at the beginning of the string and matches every character. By default, the match ends at the end of the first line; the regular expression pattern matches the carriage return character, `\r` or \u000D, but it does not match `\n`. Because the RegexOptions.Singleline option interprets the entire input string as a single line, it matches every character in the input string, including `\n`.

```csharp
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "^.+";
      string input = "This is one line and" + Environment.NewLine + "this is the second.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(Regex.Escape(match.Value));

      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Singleline))
         Console.WriteLine(Regex.Escape(match.Value));
   }
}
// The example displays the following output:
//       This\ is\ one\ line\ and\r
//
//       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

```vb
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "^.+"
      Dim input As String = "This is one line and" + vbCrLf + "this is the second."
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(Regex.Escape(match.Value))
      Next
      Console.WriteLine()
      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.SingleLine)
         Console.WriteLine(Regex.Escape(match.Value))
      Next
   End Sub
End Module
' The example displays the following output:
'       This\ is\ one\ line\ and\r
'
'       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

The following example is equivalent to the previous one, except that it uses the inline option `(?s)` to enable single-line mode.

```csharp
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "(?s)^.+";
      string input = "This is one line and" + Environment.NewLine + "this is the second.";

      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(Regex.Escape(match.Value));
   }
}
// The example displays the following output:
//       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "(?s)^.+"
      Dim input As String = "This is one line and" + vbCrLf + "this is the second."

      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(Regex.Escape(match.Value))
      Next
   End Sub
End Module
' The example displays the following output:
'       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

Back to Top

Explicit Captures Only

By default, capturing groups are defined by the use of parentheses in the regular expression pattern. Named groups are assigned a name or number by the `(?<` *name* `>` *subexpression* `)` language option, whereas unnamed groups are accessible by index. In the GroupCollection object, unnamed groups precede named groups.

Grouping constructs are often used only to apply quantifiers to multiple language elements, and the captured substrings are of no interest. For example, if the following regular expression:

```
\b\(?((\w+),?\s?)+[\.!?]\)?
```

is intended only to extract sentences that end with a period, exclamation point, or question mark from a document, only the resulting sentence (which is represented by the Match object) is of interest. The individual words in the collection are not.

Capturing groups that are not subsequently used can be expensive, because the regular expression engine must populate both the GroupCollection and CaptureCollection collection objects. As an alternative, you can use either the RegexOptions.ExplicitCapture option or the `n` inline option to specify that the only valid captures are explicitly named or numbered groups that are designated by the `(?<` *name* `>` *subexpression* `)` construct.

The following example displays information about the matches returned by the `\b\(?((\w+),?\s?)+[\.!?]\)?` regular expression pattern when the Match method is called with and without the RegexOptions.ExplicitCapture option. As the output from the first method call shows, the regular expression engine fully populates the GroupCollection and CaptureCollection collection objects with information about captured substrings. Because the second method is called with `options` set to RegexOptions.ExplicitCapture, it does not capture information on groups.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is the first sentence. Is it the beginning " +
                     "of a literary masterpiece? I think not. Instead, " +
                     "it is a nonsensical paragraph.";
      string pattern = @"\b\(?((?>\w+),?\s?)+[\.!?]\)?";
      Console.WriteLine("With implicit captures:");
```

```
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine("The match: {0}", match.Value);
         int groupCtr = 0;
         foreach (Group group in match.Groups)
         {
            Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value);
            groupCtr++;
            int captureCtr = 0;
            foreach (Capture capture in group.Captures)
            {
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value);
               captureCtr++;
            }
         }
      }
      Console.WriteLine();
      Console.WriteLine("With explicit captures only:");
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.ExplicitCapture))
      {
         Console.WriteLine("The match: {0}", match.Value);
         int groupCtr = 0;
         foreach (Group group in match.Groups)
         {
            Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value);
            groupCtr++;
            int captureCtr = 0;
            foreach (Capture capture in group.Captures)
            {
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value);
               captureCtr++;
            }
         }
      }
   }
}
// The example displays the following output:
//    With implicit captures:
//    The match: This is the first sentence.
//       Group 0: This is the first sentence.
//          Capture 0: This is the first sentence.
//       Group 1: sentence
//          Capture 0: This
//          Capture 1: is
//          Capture 2: the
//          Capture 3: first
//          Capture 4: sentence
//       Group 2: sentence
//          Capture 0: This
//          Capture 1: is
//          Capture 2: the
//          Capture 3: first
//          Capture 4: sentence
//    The match: Is it the beginning of a literary masterpiece?
//       Group 0: Is it the beginning of a literary masterpiece?
//          Capture 0: Is it the beginning of a literary masterpiece?
//       Group 1: masterpiece
//          Capture 0: Is
//          Capture 1: it
//          Capture 2: the
//          Capture 3: beginning
//          Capture 4: of
//          Capture 5: a
//          Capture 6: literary
//          Capture 7: masterpiece
//       Group 2: masterpiece
//          Capture 0: Is
//          Capture 1: it
//          Capture 2: the
```

```
//         Capture 3: beginning
//         Capture 4: of
//         Capture 5: a
//         Capture 6: literary
//         Capture 7: masterpiece
//    The match: I think not.
//       Group 0: I think not.
//          Capture 0: I think not.
//       Group 1: not
//          Capture 0: I
//          Capture 1: think
//          Capture 2: not
//       Group 2: not
//          Capture 0: I
//          Capture 1: think
//          Capture 2: not
//    The match: Instead, it is a nonsensical paragraph.
//       Group 0: Instead, it is a nonsensical paragraph.
//          Capture 0: Instead, it is a nonsensical paragraph.
//       Group 1: paragraph
//          Capture 0: Instead,
//          Capture 1: it
//          Capture 2: is
//          Capture 3: a
//          Capture 4: nonsensical
//          Capture 5: paragraph
//       Group 2: paragraph
//          Capture 0: Instead
//          Capture 1: it
//          Capture 2: is
//          Capture 3: a
//          Capture 4: nonsensical
//          Capture 5: paragraph
//
//    With explicit captures only:
//    The match: This is the first sentence.
//       Group 0: This is the first sentence.
//          Capture 0: This is the first sentence.
//    The match: Is it the beginning of a literary masterpiece?
//       Group 0: Is it the beginning of a literary masterpiece?
//          Capture 0: Is it the beginning of a literary masterpiece?
//    The match: I think not.
//       Group 0: I think not.
//          Capture 0: I think not.
//    The match: Instead, it is a nonsensical paragraph.
//       Group 0: Instead, it is a nonsensical paragraph.
//          Capture 0: Instead, it is a nonsensical paragraph.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "This is the first sentence. Is it the beginning " + _
                            "of a literary masterpiece? I think not. Instead, " + _
                            "it is a nonsensical paragraph."
      Dim pattern As String = "\b\(?((?>\w+),?\s?)+[\.!?]\)?"
      Console.WriteLine("With implicit captures:")
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("The match: {0}", match.Value)
         Dim groupCtr As Integer = 0
         For Each group As Group In match.Groups
            Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value)
            groupCtr += 1
            Dim captureCtr As Integer = 0
            For Each capture As Capture In group.Captures
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value)
               captureCtr += 1
```

```
            Next
         Next
      Next
      Console.WriteLine()
      Console.WriteLine("With explicit captures only:")
      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.ExplicitCapture)
         Console.WriteLine("The match: {0}", match.Value)
         Dim groupCtr As Integer = 0
         For Each group As Group In match.Groups
            Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value)
            groupCtr += 1
            Dim captureCtr As Integer = 0
            For Each capture As Capture In group.Captures
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value)
               captureCtr += 1
            Next
         Next
      Next
   End Sub
End Module
' The example displays the following output:
'    With implicit captures:
'    The match: This is the first sentence.
'       Group 0: This is the first sentence.
'          Capture 0: This is the first sentence.
'       Group 1: sentence
'          Capture 0: This
'          Capture 1: is
'          Capture 2: the
'          Capture 3: first
'          Capture 4: sentence
'       Group 2: sentence
'          Capture 0: This
'          Capture 1: is
'          Capture 2: the
'          Capture 3: first
'          Capture 4: sentence
'    The match: Is it the beginning of a literary masterpiece?
'       Group 0: Is it the beginning of a literary masterpiece?
'          Capture 0: Is it the beginning of a literary masterpiece?
'       Group 1: masterpiece
'          Capture 0: Is
'          Capture 1: it
'          Capture 2: the
'          Capture 3: beginning
'          Capture 4: of
'          Capture 5: a
'          Capture 6: literary
'          Capture 7: masterpiece
'       Group 2: masterpiece
'          Capture 0: Is
'          Capture 1: it
'          Capture 2: the
'          Capture 3: beginning
'          Capture 4: of
'          Capture 5: a
'          Capture 6: literary
'          Capture 7: masterpiece
'    The match: I think not.
'       Group 0: I think not.
'          Capture 0: I think not.
'       Group 1: not
'          Capture 0: I
'          Capture 1: think
'          Capture 2: not
'       Group 2: not
'          Capture 0: I
'          Capture 1: think
'          Capture 2: not
```

```
'    The match: Instead, it is a nonsensical paragraph.
'       Group 0: Instead, it is a nonsensical paragraph.
'          Capture 0: Instead, it is a nonsensical paragraph.
'       Group 1: paragraph
'          Capture 0: Instead,
'          Capture 1: it
'          Capture 2: is
'          Capture 3: a
'          Capture 4: nonsensical
'          Capture 5: paragraph
'       Group 2: paragraph
'          Capture 0: Instead
'          Capture 1: it
'          Capture 2: is
'          Capture 3: a
'          Capture 4: nonsensical
'          Capture 5: paragraph
'
'    With explicit captures only:
'    The match: This is the first sentence.
'       Group 0: This is the first sentence.
'          Capture 0: This is the first sentence.
'    The match: Is it the beginning of a literary masterpiece?
'       Group 0: Is it the beginning of a literary masterpiece?
'          Capture 0: Is it the beginning of a literary masterpiece?
'    The match: I think not.
'       Group 0: I think not.
'          Capture 0: I think not.
'    The match: Instead, it is a nonsensical paragraph.
'       Group 0: Instead, it is a nonsensical paragraph.
'          Capture 0: Instead, it is a nonsensical paragraph.
```

The regular expression pattern `\b\(?((?>\w+),?\s?)+[\.!?]\)?` is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin at a word boundary.
`\(?`	Match zero or one occurrences of the opening parenthesis ("(").
`(?>\w+),?`	Match one or more word characters, followed by zero or one commas. Do not backtrack when matching word characters.
`\s?`	Match zero or one white-space characters.
`((\w+),?\s?)+`	Match the combination of one or more word characters, zero or one commas, and zero or one white-space characters one or more times.
`[\.!?]\)?`	Match any of the three punctuation symbols, followed by zero or one closing parentheses (")").

You can also use the `(?n)` inline element to suppress automatic captures. The following example modifies the previous regular expression pattern to use the `(?n)` inline element instead of the RegexOptions.ExplicitCapture option.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is the first sentence. Is it the beginning " +
                     "of a literary masterpiece? I think not. Instead, " +
                     "it is a nonsensical paragraph.";
      string pattern = @"(?n)\b\(?((?>\w+),?\s?)+[\.!?]\)?";

      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine("The match: {0}", match.Value);
         int groupCtr = 0;
         foreach (Group group in match.Groups)
         {
            Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value);
            groupCtr++;
            int captureCtr = 0;
            foreach (Capture capture in group.Captures)
            {
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value);
               captureCtr++;
            }
         }
      }
   }
}
// The example displays the following output:
//       The match: This is the first sentence.
//          Group 0: This is the first sentence.
//             Capture 0: This is the first sentence.
//       The match: Is it the beginning of a literary masterpiece?
//          Group 0: Is it the beginning of a literary masterpiece?
//             Capture 0: Is it the beginning of a literary masterpiece?
//       The match: I think not.
//          Group 0: I think not.
//             Capture 0: I think not.
//       The match: Instead, it is a nonsensical paragraph.
//          Group 0: Instead, it is a nonsensical paragraph.
//             Capture 0: Instead, it is a nonsensical paragraph.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "This is the first sentence. Is it the beginning " + _
                            "of a literary masterpiece? I think not. Instead, " + _
                            "it is a nonsensical paragraph."
      Dim pattern As String = "(?n)\b\(?((?>\w+),?\s?)+[\.!?]\)?"

      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("The match: {0}", match.Value)
         Dim groupCtr As Integer = 0
         For Each group As Group In match.Groups
            Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value)
            groupCtr += 1
            Dim captureCtr As Integer = 0
            For Each capture As Capture In group.Captures
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value)
               captureCtr += 1
            Next
         Next
      Next
   End Sub
End Module
' The example displays the following output:
'       The match: This is the first sentence.
'          Group 0: This is the first sentence.
'             Capture 0: This is the first sentence.
'       The match: Is it the beginning of a literary masterpiece?
'          Group 0: Is it the beginning of a literary masterpiece?
'             Capture 0: Is it the beginning of a literary masterpiece?
'       The match: I think not.
'          Group 0: I think not.
'             Capture 0: I think not.
'       The match: Instead, it is a nonsensical paragraph.
'          Group 0: Instead, it is a nonsensical paragraph.
'             Capture 0: Instead, it is a nonsensical paragraph.
```

Finally, you can use the inline group element `(?n:)` to suppress automatic captures on a group-by-group basis. The following example modifies the previous pattern to suppress unnamed captures in the outer group, `((?>\w+),?\s?)`. Note that this suppresses unnamed captures in the inner group as well.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is the first sentence. Is it the beginning " +
                     "of a literary masterpiece? I think not. Instead, " +
                     "it is a nonsensical paragraph.";
      string pattern = @"\b\(?((?>\w+),?\s?)+[\.!?]\)?";

      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine("The match: {0}", match.Value);
         int groupCtr = 0;
         foreach (Group group in match.Groups)
         {
            Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value);
            groupCtr++;
            int captureCtr = 0;
            foreach (Capture capture in group.Captures)
            {
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value);
               captureCtr++;
            }
         }
      }
   }
}
// The example displays the following output:
//       The match: This is the first sentence.
//          Group 0: This is the first sentence.
//             Capture 0: This is the first sentence.
//       The match: Is it the beginning of a literary masterpiece?
//          Group 0: Is it the beginning of a literary masterpiece?
//             Capture 0: Is it the beginning of a literary masterpiece?
//       The match: I think not.
//          Group 0: I think not.
//             Capture 0: I think not.
//       The match: Instead, it is a nonsensical paragraph.
//          Group 0: Instead, it is a nonsensical paragraph.
//             Capture 0: Instead, it is a nonsensical paragraph.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "This is the first sentence. Is it the beginning " + _
                            "of a literary masterpiece? I think not. Instead, " + _
                            "it is a nonsensical paragraph."
      Dim pattern As String = "\b\(?(?n:(?>\w+),?\s?)+[\.!?]\)?"

      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("The match: {0}", match.Value)
         Dim groupCtr As Integer = 0
         For Each group As Group In match.Groups
            Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value)
            groupCtr += 1
            Dim captureCtr As Integer = 0
            For Each capture As Capture In group.Captures
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value)
               captureCtr += 1
            Next
         Next
      Next
   End Sub
End Module
' The example displays the following output:
'       The match: This is the first sentence.
'          Group 0: This is the first sentence.
'             Capture 0: This is the first sentence.
'       The match: Is it the beginning of a literary masterpiece?
'          Group 0: Is it the beginning of a literary masterpiece?
'             Capture 0: Is it the beginning of a literary masterpiece?
'       The match: I think not.
'          Group 0: I think not.
'             Capture 0: I think not.
'       The match: Instead, it is a nonsensical paragraph.
'          Group 0: Instead, it is a nonsensical paragraph.
'             Capture 0: Instead, it is a nonsensical paragraph.
```

Back to Top

Compiled Regular Expressions

By default, regular expressions in .NET are interpreted. When a Regex object is instantiated or a static Regex method is called, the regular expression pattern is parsed into a set of custom opcodes, and an interpreter uses these opcodes to run the regular expression. This involves a tradeoff: The cost of initializing the regular expression engine is minimized at the expense of run-time performance.

You can use compiled instead of interpreted regular expressions by using the RegexOptions.Compiled option. In this case, when a pattern is passed to the regular expression engine, it is parsed into a set of opcodes and then converted to Microsoft intermediate language (MSIL), which can be passed directly to the common language runtime. Compiled regular expressions maximize run-time performance at the expense of initialization time.

> **NOTE**
>
> A regular expression can be compiled only by supplying the RegexOptions.Compiled value to the `options` parameter of a Regex class constructor or a static pattern-matching method. It is not available as an inline option.

You can use compiled regular expressions in calls to both static and instance regular expressions. In static regular expressions, the RegexOptions.Compiled option is passed to the `options` parameter of the regular expression pattern-matching method. In instance regular expressions, it is passed to the `options` parameter of the Regex class constructor. In both cases, it results in enhanced performance.

However, this improvement in performance occurs only under the following conditions:

- A Regex object that represents a particular regular expression is used in multiple calls to regular expression pattern-matching methods.
- The Regex object is not allowed to go out of scope, so it can be reused.
- A static regular expression is used in multiple calls to regular expression pattern-matching methods. (The performance improvement is possible because regular expressions used in static method calls are cached by the regular expression engine.)

> **NOTE**
>
> The RegexOptions.Compiled option is unrelated to the Regex.CompileToAssembly method, which creates a special-purpose assembly that contains predefined compiled regular expressions.

Back to Top

Ignore White Space

By default, white space in a regular expression pattern is significant; it forces the regular expression engine to match a white-space character in the input string. Because of this, the regular expression "`\b\w+\s`" and "`\b\w+`" are roughly equivalent regular expressions. In addition, when the number sign (#) is encountered in a regular expression pattern, it is interpreted as a literal character to be matched.

The RegexOptions.IgnorePatternWhitespace option, or the `x` inline option, changes this default behavior as follows:

- Unescaped white space in the regular expression pattern is ignored. To be part of a regular expression pattern, white-space characters must be escaped (for example, as `\s` or "`\ `").
- The number sign (#) is interpreted as the beginning of a comment, rather than as a literal character. All text in the regular expression pattern from the # character to the end of the string is interpreted as a comment.

However, in the following cases, white-space characters in a regular expression aren't ignored, even if you use the RegexOptions.IgnorePatternWhitespace option:

- White space within a character class is always interpreted literally. For example, the regular expression pattern `[ .,;:]` matches any single white-space character, period, comma, semicolon, or colon.
- White space isn't allowed within a bracketed quantifier, such as `{` *n* `}`, `{` *n* `,}`, and `{` *n* `,` *m* `}`. For example, the regular expression pattern `\d{1, 3}` fails to match any sequences of digits from one to three digits because it contains a white-space character.
- White space isn't allowed within a character sequence that introduces a language element. For example:
  - The language element `(?:` *subexpression* `)` represents a noncapturing group, and the `(?:` portion of the element can't have embedded spaces. The pattern `(? :` *subexpression* `)` throws an ArgumentException at run time because the regular expression engine can't parse the pattern, and the pattern `( ?:` *subexpression* `)` fails to match *subexpression*.
  - The language element `\p{` *name* `}`, which represents a Unicode category or named block, can't include embedded spaces in the `\p{` portion of the element. If you do include a white space, the

element throws an ArgumentException at run time.

Enabling this option helps simplify regular expressions that are often difficult to parse and to understand. It improves readability, and makes it possible to document a regular expression.

The following example defines the following regular expression pattern:

`\b \(? ( (?>\w+) ,?\s? )+ [\.!?] \)? # Matches an entire sentence.`

This pattern is similar to the pattern defined in the Explicit Captures Only section, except that it uses the RegexOptions.IgnorePatternWhitespace option to ignore pattern white space.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is the first sentence. Is it the beginning " +
                     "of a literary masterpiece? I think not. Instead, " +
                     "it is a nonsensical paragraph.";
      string pattern = @"\b\(?((?>\w+),?\s?)+[\.!?]\)?";

      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnorePatternWhitespace))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       This is the first sentence.
//       Is it the beginning of a literary masterpiece?
//       I think not.
//       Instead, it is a nonsensical paragraph.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "This is the first sentence. Is it the beginning " + _
                            "of a literary masterpiece? I think not. Instead, " + _
                            "it is a nonsensical paragraph."
      Dim pattern As String = "\b \(? ( (?>\w+) ,?\s? )+  [\.!?] \)? # Matches an entire sentence."

      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnorePatternWhitespace)
         Console.WriteLine(match.Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       This is the first sentence.
'       Is it the beginning of a literary masterpiece?
'       I think not.
'       Instead, it is a nonsensical paragraph.
```

The following example uses the inline option `(?x)` to ignore pattern white space.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is the first sentence. Is it the beginning " +
                     "of a literary masterpiece? I think not. Instead, " +
                     "it is a nonsensical paragraph.";
      string pattern = @"(?x)\b \(? ( (?>\w+) ,?\s? )+  [\.!?] \)? # Matches an entire sentence.";

      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       This is the first sentence.
//       Is it the beginning of a literary masterpiece?
//       I think not.
//       Instead, it is a nonsensical paragraph.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "This is the first sentence. Is it the beginning " + _
                            "of a literary masterpiece? I think not. Instead, " + _
                            "it is a nonsensical paragraph."
      Dim pattern As String = "(?x)\b \(? ( (?>\w+) ,?\s? )+  [\.!?] \)? # Matches an entire sentence."

      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       This is the first sentence.
'       Is it the beginning of a literary masterpiece?
'       I think not.
'       Instead, it is a nonsensical paragraph.
```

Back to Top

Right-to-Left Mode

By default, the regular expression engine searches from left to right. You can reverse the search direction by using the RegexOptions.RightToLeft option. The search automatically begins at the last character position of the string. For pattern-matching methods that include a starting position parameter, such as Regex.Match(String, Int32), the starting position is the index of the rightmost character position at which the search is to begin.

> **NOTE**
>
> Right-to-left pattern mode is available only by supplying the RegexOptions.RightToLeft value to the `options` parameter of a Regex class constructor or static pattern-matching method. It is not available as an inline option.

The RegexOptions.RightToLeft option changes the search direction only; it does not interpret the regular expression pattern from right to left. For example, the regular expression `\bb\w+\s` matches words that begin with the letter "b" and are followed by a white-space character. In the following example, the input string consists of three words that include one or more "b" characters. The first word begins with "b", the second ends with "b", and the third includes two "b" characters in the middle of the word. As the output from the example shows, only the first word matches the regular expression pattern.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\bb\w+\s";
      string input = "builder rob rabble";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.RightToLeft))
         Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);
   }
}
// The example displays the following output:
//       'builder ' found at position 0.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\bb\w+\s"
      Dim input As String = "builder rob rabble"
      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.RightToLeft)
         Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       'builder ' found at position 0.
```

Also note that the lookahead assertion (the `(?=` *subexpression* `)` language element) and the lookbehind assertion (the `(?<=` *subexpression* `)` language element) do not change direction. The lookahead assertions look to the right; the lookbehind assertions look to the left. For example, the regular expression `(?<=\d{1,2}\s)\w+,?\s\d{4}` uses the lookbehind assertion to test for a date that precedes a month name. The regular expression then matches the month and the year. For information on lookahead and lookbehind assertsions, see Grouping Constructs.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "1 May 1917", "June 16, 2003" };
      string pattern = @"(?<=\d{1,2}\s)\w+,?\s\d{4}";

      foreach (string input in inputs)
      {
         Match match = Regex.Match(input, pattern, RegexOptions.RightToLeft);
         if (match.Success)
            Console.WriteLine("The date occurs in {0}.", match.Value);
         else
            Console.WriteLine("{0} does not match.", input);
      }
   }
}
// The example displays the following output:
//       The date occurs in May 1917.
//       June 16, 2003 does not match.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim inputs() As String = { "1 May 1917", "June 16, 2003" }
      Dim pattern As String = "(?<=\d{1,2}\s)\w+,?\s\d{4}"

      For Each input As String In inputs
         Dim match As Match = Regex.Match(input, pattern, RegexOptions.RightToLeft)
         If match.Success Then
            Console.WriteLine("The date occurs in {0}.", match.Value)
         Else
            Console.WriteLine("{0} does not match.", input)
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'       The date occurs in May 1917.
'       June 16, 2003 does not match.
```

The regular expression pattern is defined as shown in the following table.

PATTERN	DESCRIPTION
`(?<=\d{1,2}\s)`	The beginning of the match must be preceded by one or two decimal digits followed by a space.
`\w+`	Match one or more word characters.
`,?`	Match zero or one comma characters.
`\s`	Match a white-space character.
`\d{4}`	Match four decimal digits.

Back to Top

ECMAScript Matching Behavior

By default, the regular expression engine uses canonical behavior when matching a regular expression pattern to input text. However, you can instruct the regular expression engine to use ECMAScript matching behavior by specifying the RegexOptions.ECMAScript option.

> **NOTE**
>
> ECMAScript-compliant behavior is available only by supplying the RegexOptions.ECMAScript value to the `options` parameter of a Regex class constructor or static pattern-matching method. It is not available as an inline option.

The RegexOptions.ECMAScript option can be combined only with the RegexOptions.IgnoreCase and RegexOptions.Multiline options. The use of any other option in a regular expression results in an ArgumentOutOfRangeException.

The behavior of ECMAScript and canonical regular expressions differs in three areas: character class syntax, self-referencing capturing groups, and octal versus backreference interpretation.

- Character class syntax. Because canonical regular expressions support Unicode whereas ECMAScript does not, character classes in ECMAScript have a more limited syntax, and some character class language elements have a different meaning. For example, ECMAScript does not support language elements such as the Unicode category or block elements `\p` and `\P`. Similarly, the `\w` element, which matches a word character, is equivalent to the `[a-zA-Z_0-9]` character class when using ECMAScript and `[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]` when using canonical behavior. For more information, see Character Classes.

  The following example illustrates the difference between canonical and ECMAScript pattern matching. It defines a regular expression, `\b(\w+\s*)+`, that matches words followed by white-space characters. The input consists of two strings, one that uses the Latin character set and the other that uses the Cyrillic character set. As the output shows, the call to the Regex.IsMatch(String, String, RegexOptions) method that uses ECMAScript matching fails to match the Cyrillic words, whereas the method call that uses canonical matching does match these words.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] values = { "целый мир", "the whole world" };
      string pattern = @"\b(\w+\s*)+";
      foreach (var value in values)
      {
         Console.Write("Canonical matching: ");
         if (Regex.IsMatch(value, pattern))
            Console.WriteLine("'{0}' matches the pattern.", value);
         else
            Console.WriteLine("{0} does not match the pattern.", value);

         Console.Write("ECMAScript matching: ");
         if (Regex.IsMatch(value, pattern, RegexOptions.ECMAScript))
            Console.WriteLine("'{0}' matches the pattern.", value);
         else
            Console.WriteLine("{0} does not match the pattern.", value);
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Canonical matching: 'целый мир' matches the pattern.
//       ECMAScript matching: целый мир does not match the pattern.
//
//       Canonical matching: 'the whole world' matches the pattern.
//       ECMAScript matching: 'the whole world' matches the pattern.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim values() As String = { "целый мир", "the whole world" }
      Dim pattern As String = "\b(\w+\s*)+"
      For Each value In values
         Console.Write("Canonical matching: ")
         If Regex.IsMatch(value, pattern)
            Console.WriteLine("'{0}' matches the pattern.", value)
         Else
            Console.WriteLine("{0} does not match the pattern.", value)
         End If

         Console.Write("ECMAScript matching: ")
         If Regex.IsMatch(value, pattern, RegexOptions.ECMAScript)
            Console.WriteLine("'{0}' matches the pattern.", value)
         Else
            Console.WriteLine("{0} does not match the pattern.", value)
         End If
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays the following output:
'       Canonical matching: 'целый мир' matches the pattern.
'       ECMAScript matching: целый мир does not match the pattern.
'
'       Canonical matching: 'the whole world' matches the pattern.
'       ECMAScript matching: 'the whole world' matches the pattern.
```

- Self-referencing capturing groups. A regular expression capture class with a backreference to itself must be updated with each capture iteration. As the following example shows, this feature enables the regular expression `((a+)(\1) ?)+` to match the input string " aa aaaa aaaaaa " when using ECMAScript, but not when using canonical matching.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   static string pattern;

   public static void Main()
   {
      string input = "aa aaaa aaaaaa ";
      pattern = @"((a+)(\1) ?)+";

      // Match input using canonical matching.
      AnalyzeMatch(Regex.Match(input, pattern));

      // Match input using ECMAScript.
      AnalyzeMatch(Regex.Match(input, pattern, RegexOptions.ECMAScript));
   }

   private static void AnalyzeMatch(Match m)
   {
      if (m.Success)
      {
         Console.WriteLine("'{0}' matches {1} at position {2}.",
                           pattern, m.Value, m.Index);
         int grpCtr = 0;
         foreach (Group grp in m.Groups)
         {
            Console.WriteLine("   {0}: '{1}'", grpCtr, grp.Value);
            grpCtr++;
            int capCtr = 0;
            foreach (Capture cap in grp.Captures)
            {
               Console.WriteLine("      {0}: '{1}'", capCtr, cap.Value);
               capCtr++;
            }
         }
      }
      else
      {
         Console.WriteLine("No match found.");
      }
      Console.WriteLine();
   }
}
// The example displays the following output:
//    No match found.
//
//    '((a+)(\1) ?)+' matches aa aaaa aaaaaa  at position 0.
//       0: 'aa aaaa aaaaaa '
//          0: 'aa aaaa aaaaaa '
//       1: 'aaaaaa '
//          0: 'aa '
//          1: 'aaaa '
//          2: 'aaaaaa '
//       2: 'aa'
//          0: 'aa'
//          1: 'aa'
//          2: 'aa'
//       3: 'aaaa '
//          0: ''
//          1: 'aa '
//          2: 'aaaa '
```

```
Imports System.Text.RegularExpressions

Module Example
   Dim pattern As String

   Public Sub Main()
      Dim input As String = "aa aaaa aaaaaa "
      pattern = "((a+)(\1) ?)+"

      ' Match input using canonical matching.
      AnalyzeMatch(Regex.Match(input, pattern))

      ' Match input using ECMAScript.
      AnalyzeMatch(Regex.Match(input, pattern, RegexOptions.ECMAScript))
   End Sub

   Private Sub AnalyzeMatch(m As Match)
      If m.Success
         Console.WriteLine("'{0}' matches {1} at position {2}.", _
                           pattern, m.Value, m.Index)
         Dim grpCtr As Integer = 0
         For Each grp As Group In m.Groups
            Console.WriteLine("   {0}: '{1}'", grpCtr, grp.Value)
            grpCtr += 1
            Dim capCtr As Integer = 0
            For Each cap As Capture In grp.Captures
               Console.WriteLine("      {0}: '{1}'", capCtr, cap.Value)
               capCtr += 1
            Next
         Next
      Else
         Console.WriteLine("No match found.")
      End If
      Console.WriteLine()
   End Sub
End Module
' The example displays the following output:
'    No match found.
'
'    '((a+)(\1) ?)+' matches aa aaaa aaaaaa  at position 0.
'       0: 'aa aaaa aaaaaa '
'          0: 'aa aaaa aaaaaa '
'       1: 'aaaaaa '
'          0: 'aa '
'          1: 'aaaa '
'          2: 'aaaaaa '
'       2: 'aa'
'          0: 'aa'
'          1: 'aa'
'          2: 'aa'
'       3: 'aaaa '
'          0: ''
'          1: 'aa '
'          2: 'aaaa '
```

The regular expression is defined as shown in the following table.

PATTERN	DESCRIPTION
(a+)	Match the letter "a" one or more times. This is the second capturing group.
(\1)	Match the substring captured by the first capturing group. This is the third capturing group.

PATTERN	DESCRIPTION
?	Match zero or one space characters.
((a+)(\1) ?)+	Match the pattern of one or more "a" characters followed by a string that matches the first capturing group followed by zero or one space characters one or more times. This is the first capturing group.

- Resolution of ambiguities between octal escapes and backreferences. The following table summarizes the differences in octal versus backreference interpretation by canonical and ECMAScript regular expressions.

REGULAR EXPRESSION	CANONICAL BEHAVIOR	ECMASCRIPT BEHAVIOR
`\0` followed by 0 to 2 octal digits	Interpret as an octal. For example, `\044` is always interpreted as an octal value and means "$".	Same behavior.
`\` followed by a digit from 1 to 9, followed by no additional decimal digits,	Interpret as a backreference. For example, `\9` always means backreference 9, even if a ninth capturing group does not exist. If the capturing group does not exist, the regular expression parser throws an ArgumentException.	If a single decimal digit capturing group exists, backreference to that digit. Otherwise, interpret the value as a literal.
`\` followed by a digit from 1 to 9, followed by additional decimal digits	Interpret the digits as a decimal value. If that capturing group exists, interpret the expression as a backreference. Otherwise, interpret the leading octal digits up to octal 377; that is, consider only the low 8 bits of the value. Interpret the remaining digits as literals. For example, in the expression `\3000`, if capturing group 300 exists, interpret as backreference 300; if capturing group 300 does not exist, interpret as octal 300 followed by 0.	Interpret as a backreference by converting as many digits as possible to a decimal value that can refer to a capture. If no digits can be converted, interpret as an octal by using the leading octal digits up to octal 377; interpret the remaining digits as literals.

Back to Top

Comparison Using the Invariant Culture

By default, when the regular expression engine performs case-insensitive comparisons, it uses the casing conventions of the current culture to determine equivalent uppercase and lowercase characters.

However, this behavior is undesirable for some types of comparisons, particularly when comparing user input to the names of system resources, such as passwords, files, or URLs. The following example illustrates such as scenario. The code is intended to block access to any resource whose URL is prefaced with **FILE://**. The regular expression attempts a case-insensitive match with the string by using the regular expression `$FILE://` . However, when the current system culture is tr-TR (Turkish-Turkey), "I" is not the uppercase equivalent of "i". As a result, the call to the Regex.IsMatch method returns `false`, and access to the file is allowed.

```
CultureInfo defaultCulture = Thread.CurrentThread.CurrentCulture;
Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");

string input = "file://c:/Documents.MyReport.doc";
string pattern = "FILE://";

Console.WriteLine("Culture-sensitive matching ({0} culture)...",
                  Thread.CurrentThread.CurrentCulture.Name);
if (Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase))
   Console.WriteLine("URLs that access files are not allowed.");
else
   Console.WriteLine("Access to {0} is allowed.", input);

Thread.CurrentThread.CurrentCulture = defaultCulture;
// The example displays the following output:
//       Culture-sensitive matching (tr-TR culture)...
//       Access to file://c:/Documents.MyReport.doc is allowed.
```

```
Dim defaultCulture As CultureInfo = Thread.CurrentThread.CurrentCulture
Thread.CurrentThread.CurrentCulture = New CultureInfo("tr-TR")

Dim input As String = "file://c:/Documents.MyReport.doc"
Dim pattern As String = "$FILE://"

Console.WriteLine("Culture-sensitive matching ({0} culture)...", _
                  Thread.CurrentThread.CurrentCulture.Name)
If Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase) Then
   Console.WriteLine("URLs that access files are not allowed.")
Else
   Console.WriteLine("Access to {0} is allowed.", input)
End If

Thread.CurrentThread.CurrentCulture = defaultCulture
' The example displays the following output:
'       Culture-sensitive matching (tr-TR culture)...
'       Access to file://c:/Documents.MyReport.doc is allowed.
```

NOTE

For more information about string comparisons that are case-sensitive and that use the invariant culture, see Best Practices for Using Strings.

Instead of using the case-insensitive comparisons of the current culture, you can specify the RegexOptions.CultureInvariant option to ignore cultural differences in language and to use the conventions of the invariant culture.

NOTE

Comparison using the invariant culture is available only by supplying the RegexOptions.CultureInvariant value to the `options` parameter of a Regex class constructor or static pattern-matching method. It is not available as an inline option.

The following example is identical to the previous example, except that the static Regex.IsMatch(String, String, RegexOptions) method is called with options that include RegexOptions.CultureInvariant. Even when the current culture is set to Turkish (Turkey), the regular expression engine is able to successfully match "FILE" and "file" and block access to the file resource.

```
CultureInfo defaultCulture = Thread.CurrentThread.CurrentCulture;
Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");

string input = "file://c:/Documents.MyReport.doc";
string pattern = "FILE://";

Console.WriteLine("Culture-insensitive matching...");
if (Regex.IsMatch(input, pattern,
                  RegexOptions.IgnoreCase | RegexOptions.CultureInvariant))
   Console.WriteLine("URLs that access files are not allowed.");
else
   Console.WriteLine("Access to {0} is allowed.", input);

Thread.CurrentThread.CurrentCulture = defaultCulture;
// The example displays the following output:
//       Culture-insensitive matching...
//       URLs that access files are not allowed.
```

```
Dim defaultCulture As CultureInfo = Thread.CurrentThread.CurrentCulture
Thread.CurrentThread.CurrentCulture = New CultureInfo("tr-TR")

Dim input As String = "file://c:/Documents.MyReport.doc"
Dim pattern As String = "$FILE://"

Console.WriteLine("Culture-insensitive matching...")
If Regex.IsMatch(input, pattern, _
               RegexOptions.IgnoreCase Or RegexOptions.CultureInvariant) Then
   Console.WriteLine("URLs that access files are not allowed.")
Else
   Console.WriteLine("Access to {0} is allowed.", input)
End If
Thread.CurrentThread.CurrentCulture = defaultCulture
' The example displays the following output:
'        Culture-insensitive matching...
'        URLs that access files are not allowed.
```

See also

- Regular Expression Language - Quick Reference

Miscellaneous Constructs in Regular Expressions

9/6/2018 • 7 minutes to read • Edit Online

Regular expressions in .NET include three miscellaneous language constructs. One lets you enable or disable particular matching options in the middle of a regular expression pattern. The remaining two let you include comments in a regular expression.

Inline Options

You can set or disable specific pattern matching options for part of a regular expression by using the syntax

```
(?imnsx-imnsx)
```

You list the options you want to enable after the question mark, and the options you want to disable after the minus sign. The following table describes each option. For more information about each option, see Regular Expression Options.

OPTION	DESCRIPTION
`i`	Case-insensitive matching.
`m`	Multiline mode.
`n`	Explicit captures only. (Parentheses do not act as capturing groups.)
`s`	Single-line mode.
`x`	Ignore unescaped white space, and allow x-mode comments.

Any change in regular expression options defined by the `(?imnsx-imnsx)` construct remains in effect until the end of the enclosing group.

> **NOTE**
> The `(?imnsx-imnsx:` *subexpression* `)` grouping construct provides identical functionality for a subexpression. For more information, see Grouping Constructs.

The following example uses the `i`, `n`, and `x` options to enable case insensitivity and explicit captures, and to ignore white space in the regular expression pattern in the middle of a regular expression.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern;
      string input = "double dare double Double a Drooling dog The Dreaded Deep";

      pattern = @"\b(D\w+)\s(d\w+)\b";
      // Match pattern using default options.
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine(match.Value);
         if (match.Groups.Count > 1)
            for (int ctr = 1; ctr < match.Groups.Count; ctr++)
               Console.WriteLine("   Group {0}: {1}", ctr, match.Groups[ctr].Value);
      }
      Console.WriteLine();

      // Change regular expression pattern to include options.
      pattern = @"\b(D\w+)(?ixn) \s (d\w+) \b";
      // Match new pattern with options.
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine(match.Value);
         if (match.Groups.Count > 1)
            for (int ctr = 1; ctr < match.Groups.Count; ctr++)
               Console.WriteLine("   Group {0}: '{1}'", ctr, match.Groups[ctr].Value);
      }
   }
}
// The example displays the following output:
//       Drooling dog
//          Group 1: Drooling
//          Group 2: dog
//
//       Drooling dog
//          Group 1: 'Drooling'
//       Dreaded Deep
//          Group 1: 'Dreaded'
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String
      Dim input As String = "double dare double Double a Drooling dog The Dreaded Deep"

      pattern = "\b(D\w+)\s(d\w+)\b"
      ' Match pattern using default options.
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Value)
         If match.Groups.Count > 1 Then
            For ctr As Integer = 1 To match.Groups.Count - 1
               Console.WriteLine("   Group {0}: {1}", ctr, match.Groups(ctr).Value)
            Next
         End If
      Next
      Console.WriteLine()

      ' Change regular expression pattern to include options.
      pattern = "\b(D\w+)(?ixn) \s (d\w+) \b"
      ' Match new pattern with options.
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Value)
         If match.Groups.Count > 1 Then
            For ctr As Integer = 1 To match.Groups.Count - 1
               Console.WriteLine("   Group {0}: '{1}'", ctr, match.Groups(ctr).Value)
            Next
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'       Drooling dog
'          Group 1: Drooling
'          Group 2: dog
'
'       Drooling dog
'          Group 1: 'Drooling'
'       Dreaded Deep
'          Group 1: 'Dreaded'
```

The example defines two regular expressions. The first, `\b(D\w+)\s(d\w+)\b`, matches two consecutive words that begin with an uppercase "D" and a lowercase "d". The second regular expression, `\b(D\w+)(?ixn) \s (d\w+) \b`, uses inline options to modify this pattern, as described in the following table. A comparison of the results confirms the effect of the `(?ixn)` construct.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`(D\w+)`	Match a capital "D" followed by one or more word characters. This is the first capture group.
`(?ixn)`	From this point on, make comparisons case-insensitive, make only explicit captures, and ignore white space in the regular expression pattern.
`\s`	Match a white-space character.

PATTERN	DESCRIPTION
`(d\w+)`	Match an uppercase or lowercase "d" followed by one or more word characters. This group is not captured because the `n` (explicit capture) option was enabled..
`\b`	Match a word boundary.

Inline Comment

The `(?#` *comment* `)` construct lets you include an inline comment in a regular expression. The regular expression engine does not use any part of the comment in pattern matching, although the comment is included in the string that is returned by the Regex.ToString method. The comment ends at the first closing parenthesis.

The following example repeats the first regular expression pattern from the example in the previous section. It adds two inline comments to the regular expression to indicate whether the comparison is case-sensitive. The regular expression pattern,
`\b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive comparison)d\w+)\b`, is defined as follows.

PATTERN	DESCRIPTION
`\b`	Start at a word boundary.
`(?# case-sensitive comparison)`	A comment. It does not affect pattern-matching behavior.
`(D\w+)`	Match a capital "D" followed by one or more word characters. This is the first capturing group.
`\s`	Match a white-space character.
`(?ixn)`	From this point on, make comparisons case-insensitive, make only explicit captures, and ignore white space in the regular expression pattern.
`(?#case-insensitive comparison)`	A comment. It does not affect pattern-matching behavior.
`(d\w+)`	Match an uppercase or lowercase "d" followed by one or more word characters. This is the second capture group.
`\b`	Match a word boundary.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive
comparison)d\w+)\b";
      Regex rgx = new Regex(pattern);
      string input = "double dare double Double a Drooling dog The Dreaded Deep";

      Console.WriteLine("Pattern: " + pattern.ToString());
      // Match pattern using default options.
      foreach (Match match in rgx.Matches(input))
      {
         Console.WriteLine(match.Value);
         if (match.Groups.Count > 1)
         {
            for (int ctr = 1; ctr <match.Groups.Count; ctr++)
               Console.WriteLine("   Group {0}: {1}", ctr, match.Groups[ctr].Value);
         }
      }
   }
}
// The example displays the following output:
//    Pattern: \b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive comp
//    arison)d\w+)\b
//    Drooling dog
//       Group 1: Drooling
//    Dreaded Deep
//       Group 1: Dreaded
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive
comparison)d\w+)\b"
      Dim rgx As New Regex(pattern)
      Dim input As String = "double dare double Double a Drooling dog The Dreaded Deep"

      Console.WriteLine("Pattern: " + pattern.ToString())
      ' Match pattern using default options.
      For Each match As Match In rgx.Matches(input)
         Console.WriteLine(match.Value)
         If match.Groups.Count > 1 Then
            For ctr As Integer = 1 To match.Groups.Count - 1
               Console.WriteLine("   Group {0}: {1}", ctr, match.Groups(ctr).Value)
            Next
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'    Pattern: \b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive comp
'    arison)d\w+)\b
'    Drooling dog
'       Group 1: Drooling
'    Dreaded Deep
'       Group 1: Dreaded
```

End-of-Line Comment

A number sign (`#`)marks an x-mode comment, which starts at the unescaped # character at the end of the regular expression pattern and continues until the end of the line. To use this construct, you must either enable the `x` option (through inline options) or supply the RegexOptions.IgnorePatternWhitespace value to the `option` parameter when instantiating the Regex object or calling a static Regex method.

The following example illustrates the end-of-line comment construct. It determines whether a string is a composite format string that includes at least one format item. The following table describes the constructs in the regular expression pattern:

`\{\d+(,-*\d+)*(\:\w{1,4}?)*\}(?x) # Looks for a composite format item.`

PATTERN	DESCRIPTION
`\{`	Match an opening brace.
`\d+`	Match one or more decimal digits.
`(,-*\d+)*`	Match zero or one occurrence of a comma, followed by an optional minus sign, followed by one or more decimal digits.
`(\:\w{1,4}?)*`	Match zero or one occurrence of a colon, followed by one to four, but as few as possible, white-space characters.
`\}`	Match a closing brace.
`(?x)`	Enable the ignore pattern white-space option so that the end-of-line comment will be recognized.
`# Looks for a composite format item.`	An end-of-line comment.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\{\d+(,-*\d+)*(\:\w{1,4}?)*\}(?x) # Looks for a composite format item.";
      string input = "{0,-3:F}";
      Console.WriteLine("'{0}':", input);
      if (Regex.IsMatch(input, pattern))
         Console.WriteLine("   contains a composite format item.");
      else
         Console.WriteLine("   does not contain a composite format item.");
   }
}
// The example displays the following output:
//       '{0,-3:F}':
//          contains a composite format item.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\{\d+(,-*\d+)*(\:\w{1,4}?)*\}(?x) # Looks for a composite format item."
      Dim input As String = "{0,-3:F}"
      Console.WriteLine("'{0}':", input)
      If Regex.IsMatch(input, pattern) Then
         Console.WriteLine("   contains a composite format item.")
      Else
         Console.WriteLine("   does not contain a composite format item.")
      End If
   End Sub
End Module
' The example displays the following output:
'       '{0,-3:F}':
'          contains a composite format item.
```

Note that, instead of providing the `(?x)` construct in the regular expression, the comment could also have been recognized by calling the Regex.IsMatch(String, String, RegexOptions) method and passing it the RegexOptions.IgnorePatternWhitespace enumeration value.

See also

- Regular Expression Language - Quick Reference

Best Practices for Regular Expressions in .NET

7/13/2018 • 39 minutes to read • Edit Online

The regular expression engine in .NET is a powerful, full-featured tool that processes text based on pattern matches rather than on comparing and matching literal text. In most cases, it performs pattern matching rapidly and efficiently. However, in some cases, the regular expression engine can appear to be very slow. In extreme cases, it can even appear to stop responding as it processes a relatively small input over the course of hours or even days.

This topic outlines some of the best practices that developers can adopt to ensure that their regular expressions achieve optimal performance. It contains the following sections:

- Consider the Input Source
- Handle Object Instantiation Appropriately
- Take Charge of Backtracking
- Use Time-out Values
- Capture Only When Necessary
- Related Topics

Consider the Input Source

In general, regular expressions can accept two types of input: constrained or unconstrained. Constrained input is text that originates from a known or reliable source and follows a predefined format. Unconstrained input is text that originates from an unreliable source, such as a web user, and may not follow a predefined or expected format.

Regular expression patterns are typically written to match valid input. That is, developers examine the text that they want to match and then write a regular expression pattern that matches it. Developers then determine whether this pattern requires correction or further elaboration by testing it with multiple valid input items. When the pattern matches all presumed valid inputs, it is declared to be production-ready and can be included in a released application. This makes a regular expression pattern suitable for matching constrained input. However, it does not make it suitable for matching unconstrained input.

To match unconstrained input, a regular expression must be able to efficiently handle three kinds of text:

- Text that matches the regular expression pattern.
- Text that does not match the regular expression pattern.
- Text that nearly matches the regular expression pattern.

The last text type is especially problematic for a regular expression that has been written to handle constrained input. If that regular expression also relies on extensive backtracking, the regular expression engine can spend an inordinate amount of time (in some cases, many hours or days) processing seemingly innocuous text.

> **WARNING**
> The following example uses a regular expression that is prone to excessive backtracking and that is likely to reject valid email addresses. You should not use it in an email validation routine. If you would like a regular expression that validates email addresses, see How to: Verify that Strings Are in Valid Email Format.

For example, consider a very commonly used but extremely problematic regular expression for validating the alias of an email address. The regular expression `^[0-9A-Z]([-.\w]*[0-9A-Z])*$` is written to process what is considered to be a valid email address, which consists of an alphanumeric character, followed by zero or more characters that can be alphanumeric, periods, or hyphens. The regular expression must end with an alphanumeric character. However, as the following example shows, although this regular expression handles valid input easily, its performance is very inefficient when it is processing nearly valid input.

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      Stopwatch sw;
      string[] addresses = { "AAAAAAAAAAA@contoso.com",
                             "AAAAAAAAAAaaaaaaaaaa!@contoso.com" };
      // The following regular expression should not actually be used to
      // validate an email address.
      string pattern = @"^[0-9A-Z]([-.\w]*[0-9A-Z])*$";
      string input;

      foreach (var address in addresses) {
         string mailBox = address.Substring(0, address.IndexOf("@"));
         int index = 0;
         for (int ctr = mailBox.Length - 1; ctr >= 0; ctr--) {
            index++;

            input = mailBox.Substring(ctr, index);
            sw = Stopwatch.StartNew();
            Match m = Regex.Match(input, pattern, RegexOptions.IgnoreCase);
            sw.Stop();
            if (m.Success)
               Console.WriteLine("{0,2}. Matched '{1,25}' in {2}",
                                 index, m.Value, sw.Elapsed);
            else
               Console.WriteLine("{0,2}. Failed  '{1,25}' in {2}",
                                 index, input, sw.Elapsed);
         }
         Console.WriteLine();
      }
   }
}

// The example displays output similar to the following:
//     1. Matched '                        A' in 00:00:00.0007122
//     2. Matched '                       AA' in 00:00:00.0000282
//     3. Matched '                      AAA' in 00:00:00.0000042
//     4. Matched '                     AAAA' in 00:00:00.0000038
//     5. Matched '                    AAAAA' in 00:00:00.0000042
//     6. Matched '                   AAAAAA' in 00:00:00.0000042
//     7. Matched '                  AAAAAAA' in 00:00:00.0000042
//     8. Matched '                 AAAAAAAA' in 00:00:00.0000087
//     9. Matched '                AAAAAAAAA' in 00:00:00.0000045
//    10. Matched '               AAAAAAAAAA' in 00:00:00.0000045
//    11. Matched '              AAAAAAAAAAA' in 00:00:00.0000045
//
//     1. Failed  '                        !' in 00:00:00.0000447
//     2. Failed  '                       a!' in 00:00:00.0000071
//     3. Failed  '                      aa!' in 00:00:00.0000071
//     4. Failed  '                     aaa!' in 00:00:00.0000061
//     5. Failed  '                    aaaa!' in 00:00:00.0000081
//     6. Failed  '                   aaaaa!' in 00:00:00.0000126
//     7. Failed  '                  aaaaaa!' in 00:00:00.0000359
//     8. Failed  '                 aaaaaaa!' in 00:00:00.0000414
```

```
//     9. Failed  '                 aaaaaaaa!' in 00:00:00.0000758
//    10. Failed  '                aaaaaaaaa!' in 00:00:00.0001462
//    11. Failed  '               aaaaaaaaaa!' in 00:00:00.0002885
//    12. Failed  '              Aaaaaaaaaaa!' in 00:00:00.0005780
//    13. Failed  '             AAaaaaaaaaaa!' in 00:00:00.0011628
//    14. Failed  '            AAAaaaaaaaaaa!' in 00:00:00.0022851
//    15. Failed  '           AAAAaaaaaaaaaa!' in 00:00:00.0045864
//    16. Failed  '          AAAAAaaaaaaaaaa!' in 00:00:00.0093168
//    17. Failed  '         AAAAAAaaaaaaaaaa!' in 00:00:00.0185993
//    18. Failed  '        AAAAAAAaaaaaaaaaa!' in 00:00:00.0366723
//    19. Failed  '       AAAAAAAAaaaaaaaaaa!' in 00:00:00.1370108
//    20. Failed  '      AAAAAAAAAaaaaaaaaaa!' in 00:00:00.1553966
//    21. Failed  '     AAAAAAAAAAaaaaaaaaaa!' in 00:00:00.3223372
```

```
Imports System.Diagnostics
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim sw As Stopwatch
      Dim addresses() As String = { "AAAAAAAAAAA@contoso.com",
                                 "AAAAAAAAAAaaaaaaaaaa!@contoso.com" }
      ' The following regular expression should not actually be used to
      ' validate an email address.
      Dim pattern As String = "^[0-9A-Z]([-.\w]*[0-9A-Z])*$"
      Dim input As String

      For Each address In addresses
         Dim mailBox As String = address.Substring(0, address.IndexOf("@"))
         Dim index As Integer = 0
         For ctr As Integer = mailBox.Length - 1 To 0 Step -1
            index += 1
            input = mailBox.Substring(ctr, index)
            sw = Stopwatch.StartNew()
            Dim m As Match = Regex.Match(input, pattern, RegexOptions.IgnoreCase)
            sw.Stop()
            if m.Success Then
               Console.WriteLine("{0,2}. Matched '{1,25}' in {2}",
                                 index, m.Value, sw.Elapsed)
            Else
               Console.WriteLine("{0,2}. Failed  '{1,25}' in {2}",
                                 index, input, sw.Elapsed)
            End If
         Next
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays output similar to the following:
'     1. Matched '                        A' in 00:00:00.0007122
'     2. Matched '                       AA' in 00:00:00.0000282
'     3. Matched '                      AAA' in 00:00:00.0000042
'     4. Matched '                     AAAA' in 00:00:00.0000038
'     5. Matched '                    AAAAA' in 00:00:00.0000042
'     6. Matched '                   AAAAAA' in 00:00:00.0000042
'     7. Matched '                  AAAAAAA' in 00:00:00.0000042
'     8. Matched '                 AAAAAAAA' in 00:00:00.0000087
'     9. Matched '                AAAAAAAAA' in 00:00:00.0000045
'    10. Matched '               AAAAAAAAAA' in 00:00:00.0000045
'    11. Matched '              AAAAAAAAAAA' in 00:00:00.0000045
'
'     1. Failed  '                        !' in 00:00:00.0000447
'     2. Failed  '                       a!' in 00:00:00.0000071
'     3. Failed  '                      aa!' in 00:00:00.0000071
'     4. Failed  '                     aaa!' in 00:00:00.0000061
'     5. Failed  '                    aaaa!' in 00:00:00.0000081
'     6. Failed  '                   aaaaa!' in 00:00:00.0000126
```

```
'      7. Failed  '                  aaaaaa!' in 00:00:00.0000359
'      8. Failed  '                 aaaaaaa!' in 00:00:00.0000414
'      9. Failed  '                aaaaaaaa!' in 00:00:00.0000758
'     10. Failed  '               aaaaaaaaa!' in 00:00:00.0001462
'     11. Failed  '              aaaaaaaaaa!' in 00:00:00.0002885
'     12. Failed  '             Aaaaaaaaaaa!' in 00:00:00.0005780
'     13. Failed  '            AAaaaaaaaaaa!' in 00:00:00.0011628
'     14. Failed  '           AAAaaaaaaaaaa!' in 00:00:00.0022851
'     15. Failed  '          AAAAaaaaaaaaaa!' in 00:00:00.0045864
'     16. Failed  '         AAAAAaaaaaaaaaa!' in 00:00:00.0093168
'     17. Failed  '        AAAAAAaaaaaaaaaa!' in 00:00:00.0185993
'     18. Failed  '       AAAAAAAaaaaaaaaaa!' in 00:00:00.0366723
'     19. Failed  '      AAAAAAAAaaaaaaaaaa!' in 00:00:00.1370108
'     20. Failed  '     AAAAAAAAAaaaaaaaaaa!' in 00:00:00.1553966
'     21. Failed  '    AAAAAAAAAAaaaaaaaaaa!' in 00:00:00.3223372
```

As the output from the example shows, the regular expression engine processes the valid email alias in about the same time interval regardless of its length. On the other hand, when the nearly valid email address has more than five characters, processing time approximately doubles for each additional character in the string. This means that a nearly valid 28-character string would take over an hour to process, and a nearly valid 33-character string would take nearly a day to process.

Because this regular expression was developed solely by considering the format of input to be matched, it fails to take account of input that does not match the pattern. This, in turn, can allow unconstrained input that nearly matches the regular expression pattern to significantly degrade performance.

To solve this problem, you can do the following:

- When developing a pattern, you should consider how backtracking might affect the performance of the regular expression engine, particularly if your regular expression is designed to process unconstrained input. For more information, see the Take Charge of Backtracking section.

- Thoroughly test your regular expression using invalid and near-valid input as well as valid input. To generate input for a particular regular expression randomly, you can use Rex, which is a regular expression exploration tool from Microsoft Research.

Back to top

Handle Object Instantiation Appropriately

At the heart of .NET's regular expression object model is the System.Text.RegularExpressions.Regex class, which represents the regular expression engine. Often, the single greatest factor that affects regular expression performance is the way in which the Regex engine is used. Defining a regular expression involves tightly coupling the regular expression engine with a regular expression pattern. That coupling process, whether it involves instantiating a Regex object by passing its constructor a regular expression pattern or calling a static method by passing it the regular expression pattern along with the string to be analyzed, is by necessity an expensive one.

> **NOTE**
> For a more detailed discussion of the performance implications of using interpreted and compiled regular expressions, see Optimizing Regular Expression Performance, Part II: Taking Charge of Backtracking in the BCL Team blog.

You can couple the regular expression engine with a particular regular expression pattern and then use the engine to match text in several ways:

- You can call a static pattern-matching method, such as Regex.Match(String, String). This does not require instantiation of a regular expression object.

- You can instantiate a Regex object and call an instance pattern-matching method of an interpreted regular expression. This is the default method for binding the regular expression engine to a regular expression pattern. It results when a Regex object is instantiated without an `options` argument that includes the Compiled flag.
- You can instantiate a Regex object and call an instance pattern-matching method of a compiled regular expression. Regular expression objects represent compiled patterns when a Regex object is instantiated with an `options` argument that includes the Compiled flag.
- You can create a special-purpose Regex object that is tightly coupled with a particular regular expression pattern, compile it, and save it to a standalone assembly. You do this by calling the Regex.CompileToAssembly method.

The particular way in which you call regular expression matching methods can have a significant impact on your application. The following sections discuss when to use static method calls, interpreted regular expressions, and compiled regular expressions to improve your application's performance.

> **IMPORTANT**
> The form of the method call (static, interpreted, compiled) affects performance if the same regular expression is used repeatedly in method calls, or if an application makes extensive use of regular expression objects.

Static Regular Expressions

Static regular expression methods are recommended as an alternative to repeatedly instantiating a regular expression object with the same regular expression. Unlike regular expression patterns used by regular expression objects, either the operation codes or the compiled Microsoft intermediate language (MSIL) from patterns used in instance method calls is cached internally by the regular expression engine.

For example, an event handler frequently calls another method to validate user input. This is reflected in the following code, in which a Button control's Click event is used to call a method named `IsValidCurrency`, which checks whether the user has entered a currency symbol followed by at least one decimal digit.

```
public void OKButton_Click(object sender, EventArgs e)
{
   if (! String.IsNullOrEmpty(sourceCurrency.Text))
      if (RegexLib.IsValidCurrency(sourceCurrency.Text))
         PerformConversion();
      else
         status.Text = "The source currency value is invalid.";
}
```

```
Public Sub OKButton_Click(sender As Object, e As EventArgs) _
           Handles OKButton.Click

   If Not String.IsNullOrEmpty(sourceCurrency.Text) Then
      If RegexLib.IsValidCurrency(sourceCurrency.Text) Then
         PerformConversion()
      Else
         status.Text = "The source currency value is invalid."
      End If
   End If
End Sub
```

A very inefficient implementation of the `IsValidCurrency` method is shown in the following example. Note that each method call reinstantiates a Regex object with the same pattern. This, in turn, means that the regular expression pattern must be recompiled each time the method is called.

```
using System;
using System.Text.RegularExpressions;

public class RegexLib
{
   public static bool IsValidCurrency(string currencyValue)
   {
      string pattern = @"\p{Sc}+\s*\d+";
      Regex currencyRegex = new Regex(pattern);
      return currencyRegex.IsMatch(currencyValue);
   }
}
```

```
Imports System.Text.RegularExpressions

Public Module RegexLib
   Public Function IsValidCurrency(currencyValue As String) As Boolean
      Dim pattern As String = "\p{Sc}+\s*\d+"
      Dim currencyRegex As New Regex(pattern)
      Return currencyRegex.IsMatch(currencyValue)
   End Function
End Module
```

You should replace this inefficient code with a call to the static Regex.IsMatch(String, String) method. This eliminates the need to instantiate a Regex object each time you want to call a pattern-matching method, and enables the regular expression engine to retrieve a compiled version of the regular expression from its cache.

```
using System;
using System.Text.RegularExpressions;

public class RegexLib
{
   public static bool IsValidCurrency(string currencyValue)
   {
      string pattern = @"\p{Sc}+\s*\d+";
      return Regex.IsMatch(currencyValue, pattern);
   }
}
```

```
Imports System.Text.RegularExpressions

Public Module RegexLib
   Public Function IsValidCurrency(currencyValue As String) As Boolean
      Dim pattern As String = "\p{Sc}+\s*\d+"
      Return Regex.IsMatch(currencyValue, pattern)
   End Function
End Module
```

By default, the last 15 most recently used static regular expression patterns are cached. For applications that require a larger number of cached static regular expressions, the size of the cache can be adjusted by setting the Regex.CacheSize property.

The regular expression `\p{Sc}+\s*\d+` that is used in this example verifies that the input string consists of a currency symbol and at least one decimal digit. The pattern is defined as shown in the following table.

PATTERN	DESCRIPTION

PATTERN	DESCRIPTION
`\p{Sc}+`	Match one or more characters in the Unicode Symbol, Currency category.
`\s*`	Match zero or more white-space characters.
`\d+`	Match one or more decimal digits.

Interpreted vs. Compiled Regular Expressions

Regular expression patterns that are not bound to the regular expression engine through the specification of the Compiled option are interpreted. When a regular expression object is instantiated, the regular expression engine converts the regular expression to a set of operation codes. When an instance method is called, the operation codes are converted to MSIL and executed by the JIT compiler. Similarly, when a static regular expression method is called and the regular expression cannot be found in the cache, the regular expression engine converts the regular expression to a set of operation codes and stores them in the cache. It then converts these operation codes to MSIL so that the JIT compiler can execute them. Interpreted regular expressions reduce startup time at the cost of slower execution time. Because of this, they are best used when the regular expression is used in a small number of method calls, or if the exact number of calls to regular expression methods is unknown but is expected to be small. As the number of method calls increases, the performance gain from reduced startup time is outstripped by the slower execution speed.

Regular expression patterns that are bound to the regular expression engine through the specification of the Compiled option are compiled. This means that, when a regular expression object is instantiated, or when a static regular expression method is called and the regular expression cannot be found in the cache, the regular expression engine converts the regular expression to an intermediary set of operation codes, which it then converts to MSIL. When a method is called, the JIT compiler executes the MSIL. In contrast to interpreted regular expressions, compiled regular expressions increase startup time but execute individual pattern-matching methods faster. As a result, the performance benefit that results from compiling the regular expression increases in proportion to the number of regular expression methods called.

To summarize, we recommend that you use interpreted regular expressions when you call regular expression methods with a specific regular expression relatively infrequently. You should use compiled regular expressions when you call regular expression methods with a specific regular expression relatively frequently. The exact threshold at which the slower execution speeds of interpreted regular expressions outweigh gains from their reduced startup time, or the threshold at which the slower startup times of compiled regular expressions outweigh gains from their faster execution speeds, is difficult to determine. It depends on a variety of factors, including the complexity of the regular expression and the specific data that it processes. To determine whether interpreted or compiled regular expressions offer the best performance for your particular application scenario, you can use the Stopwatch class to compare their execution times.

The following example compares the performance of compiled and interpreted regular expressions when reading the first ten sentences and when reading all the sentences in the text of Theodore Dreiser's *The Financier*. As the output from the example shows, when only ten calls are made to regular expression matching methods, an interpreted regular expression offers better performance than a compiled regular expression. However, a compiled regular expression offers better performance when a large number of calls (in this case, over 13,000) are made.

```
using System;
using System.Diagnostics;
using System.IO;
using System.Text.RegularExpressions;

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

```
{
   string pattern = @"\b(\w+((\r?\n)|,?\s))*\w+[.?:;!]";
   Stopwatch sw;
   Match match;
   int ctr;

   StreamReader inFile = new StreamReader(@".\Dreiser_TheFinancier.txt");
   string input = inFile.ReadToEnd();
   inFile.Close();

   // Read first ten sentences with interpreted regex.
   Console.WriteLine("10 Sentences with Interpreted Regex:");
   sw = Stopwatch.StartNew();
   Regex int10 = new Regex(pattern, RegexOptions.Singleline);
   match = int10.Match(input);
   for (ctr = 0; ctr <= 9; ctr++) {
      if (match.Success)
         // Do nothing with the match except get the next match.
         match = match.NextMatch();
      else
         break;
   }
   sw.Stop();
   Console.WriteLine("   {0} matches in {1}", ctr, sw.Elapsed);

   // Read first ten sentences with compiled regex.
   Console.WriteLine("10 Sentences with Compiled Regex:");
   sw = Stopwatch.StartNew();
   Regex comp10 = new Regex(pattern,
                RegexOptions.Singleline | RegexOptions.Compiled);
   match = comp10.Match(input);
   for (ctr = 0; ctr <= 9; ctr++) {
      if (match.Success)
         // Do nothing with the match except get the next match.
         match = match.NextMatch();
      else
         break;
   }
   sw.Stop();
   Console.WriteLine("   {0} matches in {1}", ctr, sw.Elapsed);

   // Read all sentences with interpreted regex.
   Console.WriteLine("All Sentences with Interpreted Regex:");
   sw = Stopwatch.StartNew();
   Regex intAll = new Regex(pattern, RegexOptions.Singleline);
   match = intAll.Match(input);
   int matches = 0;
   while (match.Success) {
      matches++;
      // Do nothing with the match except get the next match.
      match = match.NextMatch();
   }
   sw.Stop();
   Console.WriteLine("   {0:N0} matches in {1}", matches, sw.Elapsed);

   // Read all sentnces with compiled regex.
   Console.WriteLine("All Sentences with Compiled Regex:");
   sw = Stopwatch.StartNew();
   Regex compAll = new Regex(pattern,
                   RegexOptions.Singleline | RegexOptions.Compiled);
   match = compAll.Match(input);
   matches = 0;
   while (match.Success) {
      matches++;
      // Do nothing with the match except get the next match.
      match = match.NextMatch();
   }
   sw.Stop();
   Console.WriteLine("   {0:N0} matches in {1}", matches, sw.Elapsed);
```

```
      Console.WriteLine("   {0:N0} matches in {1}", matches, sw.Elapsed);
   }
}
// The example displays the following output:
//       10 Sentences with Interpreted Regex:
//          10 matches in 00:00:00.0047491
//       10 Sentences with Compiled Regex:
//          10 matches in 00:00:00.0141872
//       All Sentences with Interpreted Regex:
//          13,443 matches in 00:00:01.1929928
//       All Sentences with Compiled Regex:
//          13,443 matches in 00:00:00.7635869
//
//       >compare1
//       10 Sentences with Interpreted Regex:
//          10 matches in 00:00:00.0046914
//       10 Sentences with Compiled Regex:
//          10 matches in 00:00:00.0143727
//       All Sentences with Interpreted Regex:
//          13,443 matches in 00:00:01.1514100
//       All Sentences with Compiled Regex:
//          13,443 matches in 00:00:00.7432921
```

```
Imports System.Diagnostics
Imports System.IO
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b(\w+((\r?\n)|,?\s))*\w+[.?:;!]"
      Dim sw As Stopwatch
      Dim match As Match
      Dim ctr As Integer

      Dim inFile As New StreamReader(".\Dreiser_TheFinancier.txt")
      Dim input As String = inFile.ReadToEnd()
      inFile.Close()

      ' Read first ten sentences with interpreted regex.
      Console.WriteLine("10 Sentences with Interpreted Regex:")
      sw = Stopwatch.StartNew()
      Dim int10 As New Regex(pattern, RegexOptions.SingleLine)
      match = int10.Match(input)
      For ctr = 0 To 9
         If match.Success Then
            ' Do nothing with the match except get the next match.
            match = match.NextMatch()
         Else
            Exit For
         End If
      Next
      sw.Stop()
      Console.WriteLine("   {0} matches in {1}", ctr, sw.Elapsed)

      ' Read first ten sentences with compiled regex.
      Console.WriteLine("10 Sentences with Compiled Regex:")
      sw = Stopwatch.StartNew()
      Dim comp10 As New Regex(pattern,
                   RegexOptions.SingleLine Or RegexOptions.Compiled)
      match = comp10.Match(input)
      For ctr = 0 To 9
         If match.Success Then
            ' Do nothing with the match except get the next match.
            match = match.NextMatch()
         Else
            Exit For
         End If
      Next
```

```
      sw.Stop()
      Console.WriteLine("   {0} matches in {1}", ctr, sw.Elapsed)

      ' Read all sentences with interpreted regex.
      Console.WriteLine("All Sentences with Interpreted Regex:")
      sw = Stopwatch.StartNew()
      Dim intAll As New Regex(pattern, RegexOptions.SingleLine)
      match = intAll.Match(input)
      Dim matches As Integer = 0
      Do While match.Success
         matches += 1
         ' Do nothing with the match except get the next match.
         match = match.NextMatch()
      Loop
      sw.Stop()
      Console.WriteLine("   {0:N0} matches in {1}", matches, sw.Elapsed)

      ' Read all sentnces with compiled regex.
      Console.WriteLine("All Sentences with Compiled Regex:")
      sw = Stopwatch.StartNew()
      Dim compAll As New Regex(pattern,
                      RegexOptions.SingleLine Or RegexOptions.Compiled)
      match = compAll.Match(input)
      matches = 0
      Do While match.Success
         matches += 1
         ' Do nothing with the match except get the next match.
         match = match.NextMatch()
      Loop
      sw.Stop()
      Console.WriteLine("   {0:N0} matches in {1}", matches, sw.Elapsed)
   End Sub
End Module
' The example displays output like the following:
'       10 Sentences with Interpreted Regex:
'          10 matches in 00:00:00.0047491
'       10 Sentences with Compiled Regex:
'          10 matches in 00:00:00.0141872
'       All Sentences with Interpreted Regex:
'          13,443 matches in 00:00:01.1929928
'       All Sentences with Compiled Regex:
'          13,443 matches in 00:00:00.7635869
'
'       >compare1
'       10 Sentences with Interpreted Regex:
'          10 matches in 00:00:00.0046914
'       10 Sentences with Compiled Regex:
'          10 matches in 00:00:00.0143727
'       All Sentences with Interpreted Regex:
'          13,443 matches in 00:00:01.1514100
'       All Sentences with Compiled Regex:
'          13,443 matches in 00:00:00.7432921
```

The regular expression pattern used in the example, `\b(\w+((\r?\n)|,?\s))*\w+[.?:;!]`, is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`\w+`	Match one or more word characters.

PATTERN	DESCRIPTION
`(\r?\n)|,?\s)`	Match either zero or one carriage return followed by a newline character, or zero or one comma followed by a white-space character.
`(\w+((\r?\n)|,?\s))*`	Match zero or more occurrences of one or more word characters that are followed either by zero or one carriage return and a newline character, or by zero or one comma followed by a white-space character.
`\w+`	Match one or more word characters.
`[.?:;!]`	Match a period, question mark, colon, semicolon, or exclamation point.

Regular Expressions: Compiled to an Assembly

.NET also enables you to create an assembly that contains compiled regular expressions. This moves the performance hit of regular expression compilation from run time to design time. However, it also involves some additional work: You must define the regular expressions in advance and compile them to an assembly. The compiler can then reference this assembly when compiling source code that uses the assembly's regular expressions. Each compiled regular expression in the assembly is represented by a class that derives from Regex.

To compile regular expressions to an assembly, you call the Regex.CompileToAssembly(RegexCompilationInfo[], AssemblyName) method and pass it an array of RegexCompilationInfo objects that represent the regular expressions to be compiled, and an AssemblyName object that contains information about the assembly to be created.

We recommend that you compile regular expressions to an assembly in the following situations:

- If you are a component developer who wants to create a library of reusable regular expressions.
- If you expect your regular expression's pattern-matching methods to be called an indeterminate number of times -- anywhere from once or twice to thousands or tens of thousands of times. Unlike compiled or interpreted regular expressions, regular expressions that are compiled to separate assemblies offer performance that is consistent regardless of the number of method calls.

If you are using compiled regular expressions to optimize performance, you should not use reflection to create the assembly, load the regular expression engine, and execute its pattern-matching methods. This requires that you avoid building regular expression patterns dynamically, and that you specify any pattern-matching options (such as case-insensitive pattern matching) at the time the assembly is created. It also requires that you separate the code that creates the assembly from the code that uses the regular expression.

The following example shows how to create an assembly that contains a compiled regular expression. It creates an assembly named `RegexLib.dll` with a single regular expression class, `SentencePattern`, that contains the sentence-matching regular expression pattern used in the Interpreted vs. Compiled Regular Expressions section.

```
using System;
using System.Reflection;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      RegexCompilationInfo SentencePattern =
                           new RegexCompilationInfo(@"\b(\w+((\r?\n)|,?\s))*\w+[.?:;!]",
                                                    RegexOptions.Multiline,
                                                    "SentencePattern",
                                                    "Utilities.RegularExpressions",
                                                    true);
      RegexCompilationInfo[] regexes = { SentencePattern };
      AssemblyName assemName = new AssemblyName("RegexLib, Version=1.0.0.1001, Culture=neutral,
PublicKeyToken=null");
      Regex.CompileToAssembly(regexes, assemName);
   }
}
```

```
Imports System.Reflection
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim SentencePattern As New RegexCompilationInfo("\b(\w+((\r?\n)|,?\s))*\w+[.?:;!]",
                                                      RegexOptions.Multiline,
                                                      "SentencePattern",
                                                      "Utilities.RegularExpressions",
                                                      True)
      Dim regexes() As RegexCompilationInfo = {SentencePattern}
      Dim assemName As New AssemblyName("RegexLib, Version=1.0.0.1001, Culture=neutral, PublicKeyToken=null")
      Regex.CompileToAssembly(regexes, assemName)
   End Sub
End Module
```

When the example is compiled to an executable and run, it creates an assembly named `RegexLib.dll`. The regular expression is represented by a class named `Utilities.RegularExpressions.SentencePattern` that is derived from Regex. The following example then uses the compiled regular expression to extract the sentences from the text of Theodore Dreiser's *The Financier*.

```
using System;
using System.IO;
using System.Text.RegularExpressions;
using Utilities.RegularExpressions;

public class Example
{
   public static void Main()
   {
      SentencePattern pattern = new SentencePattern();
      StreamReader inFile = new StreamReader(@".\Dreiser_TheFinancier.txt");
      string input = inFile.ReadToEnd();
      inFile.Close();

      MatchCollection matches = pattern.Matches(input);
      Console.WriteLine("Found {0:N0} sentences.", matches.Count);
   }
}
// The example displays the following output:
//      Found 13,443 sentences.
```

```
Imports System.IO
Imports System.Text.RegularExpressions
Imports Utilities.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As New SentencePattern()
      Dim inFile As New StreamReader(".\Dreiser_TheFinancier.txt")
      Dim input As String = inFile.ReadToEnd()
      inFile.Close()

      Dim matches As MatchCollection = pattern.Matches(input)
      Console.WriteLine("Found {0:N0} sentences.", matches.Count)
   End Sub
End Module
' The example displays the following output:
'      Found 13,443 sentences.
```

Back to top

Take Charge of Backtracking

Ordinarily, the regular expression engine uses linear progression to move through an input string and compare it to a regular expression pattern. However, when indeterminate quantifiers such as `*`, `+`, and `?` are used in a regular expression pattern, the regular expression engine may give up a portion of successful partial matches and return to a previously saved state in order to search for a successful match for the entire pattern. This process is known as backtracking.

> **NOTE**
>
> For more information on backtracking, see Details of Regular Expression Behavior and Backtracking. For a detailed discussion of backtracking, see Optimizing Regular Expression Performance, Part II: Taking Charge of Backtracking in the BCL Team blog.

Support for backtracking gives regular expressions power and flexibility. It also places the responsibility for controlling the operation of the regular expression engine in the hands of regular expression developers. Because developers are often not aware of this responsibility, their misuse of backtracking or reliance on excessive backtracking often plays the most significant role in degrading regular expression performance. In a worst-case scenario, execution time can double for each additional character in the input string. In fact, by using backtracking excessively, it is easy to create the programmatic equivalent of an endless loop if input nearly matches the regular expression pattern; the regular expression engine may take hours or even days to process a relatively short input string.

Often, applications pay a performance penalty for using backtracking despite the fact that backtracking is not essential for a match. For example, the regular expression `\b\p{Lu}\w*\b` matches all words that begin with an uppercase character, as the following table shows.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`\p{Lu}`	Match an uppercase character.
`\w*`	Match zero or more word characters.
`\b`	End the match at a word boundary.

Because a word boundary is not the same as, or a subset of, a word character, there is no possibility that the regular expression engine will cross a word boundary when matching word characters. This means that for this regular expression, backtracking can never contribute to the overall success of any match -- it can only degrade performance, because the regular expression engine is forced to save its state for each successful preliminary match of a word character.

If you determine that backtracking is not necessary, you can disable it by using the `(?>subexpression)` language element. The following example parses an input string by using two regular expressions. The first, `\b\p{Lu}\w*\b`, relies on backtracking. The second, `\b\p{Lu}(?>\w*)\b`, disables backtracking. As the output from the example shows, they both produce the same result.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This this word Sentence name Capital";
      string pattern = @"\b\p{Lu}\w*\b";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);

      Console.WriteLine();

      pattern = @"\b\p{Lu}(?>\w*)\b";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       This
//       Sentence
//       Capital
//
//       This
//       Sentence
//       Capital
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "This this word Sentence name Capital"
      Dim pattern As String = "\b\p{Lu}\w*\b"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Value)
      Next
      Console.WriteLine()

      pattern = "\b\p{Lu}(?>\w*)\b"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       This
'       Sentence
'       Capital
'
'       This
'       Sentence
'       Capital
```

In many cases, backtracking is essential for matching a regular expression pattern to input text. However, excessive backtracking can severely degrade performance and create the impression that an application has stopped responding. In particular, this happens when quantifiers are nested and the text that matches the outer subexpression is a subset of the text that matches the inner subexpression.

WARNING

In addition to avoiding excessive backtracking, you should use the timeout feature to ensure that excessive backtracking does not severely degrade regular expression performance. For more information, see the Use Time-out Values section.

For example, the regular expression pattern `^[0-9A-Z]([-.\w]*[0-9A-Z])*\$$` is intended to match a part number that consists of at least one alphanumeric character. Any additional characters can consist of an alphanumeric character, a hyphen, an underscore, or a period, though the last character must be alphanumeric. A dollar sign terminates the part number. In some cases, this regular expression pattern can exhibit extremely poor performance because quantifiers are nested, and because the subexpression `[0-9A-Z]` is a subset of the subexpression `[-.\w]*`.

In these cases, you can optimize regular expression performance by removing the nested quantifiers and replacing the outer subexpression with a zero-width lookahead or lookbehind assertion. Lookahead and lookbehind assertions are anchors; they do not move the pointer in the input string, but instead look ahead or behind to check whether a specified condition is met. For example, the part number regular expression can be rewritten as `^[0-9A-Z][-.\w]*(?<=[0-9A-Z])\$$`. This regular expression pattern is defined as shown in the following table.

PATTERN	DESCRIPTION
`^`	Begin the match at the beginning of the input string.
`[0-9A-Z]`	Match an alphanumeric character. The part number must consist of at least this character.
`[-.\w]*`	Match zero or more occurrences of any word character, hyphen, or period.

PATTERN	DESCRIPTION
`\$`	Match a dollar sign.
`(?<=[0-9A-Z])`	Look ahead of the ending dollar sign to ensure that the previous character is alphanumeric.
`$`	End the match at the end of the input string.

The following example illustrates the use of this regular expression to match an array containing possible part numbers.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"^[0-9A-Z][-.\w]*(?<=[0-9A-Z])\$$";
      string[] partNos = { "A1C$", "A4", "A4$", "A1603D$", "A1603D#" };

      foreach (var input in partNos) {
         Match match = Regex.Match(input, pattern);
         if (match.Success)
            Console.WriteLine(match.Value);
         else
            Console.WriteLine("Match not found.");
      }
   }
}
// The example displays the following output:
//       A1C$
//       Match not found.
//       A4$
//       A1603D$
//       Match not found.
```

```vb
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "^[0-9A-Z][-.\w]*(?<=[0-9A-Z])\$$"
      Dim partNos() As String = { "A1C$", "A4", "A4$", "A1603D$",
                                  "A1603D#" }

      For Each input As String In partNos
         Dim match As Match = Regex.Match(input, pattern)
         If match.Success Then
            Console.WriteLine(match.Value)
         Else
            Console.WriteLine("Match not found.")
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'       A1C$
'       Match not found.
'       A4$
'       A1603D$
'       Match not found.
```

The regular expression language in .NET includes the following language elements that you can use to eliminate nested quantifiers. For more information, see Grouping Constructs.

LANGUAGE ELEMENT	DESCRIPTION
`(?=` `subexpression` `)`	Zero-width positive lookahead. Look ahead of the current position to determine whether `subexpression` matches the input string.
`(?!` `subexpression` `)`	Zero-width negative lookahead. Look ahead of the current position to determine whether `subexpression` does not match the input string.
`(?<=` `subexpression` `)`	Zero-width positive lookbehind. Look behind the current position to determine whether `subexpression` matches the input string.
`(?<!` `subexpression` `)`	Zero-width negative lookbehind. Look behind the current position to determine whether `subexpression` does not match the input string.

Back to top

Use Time-out Values

If your regular expressions processes input that nearly matches the regular expression pattern, it can often rely on excessive backtracking, which impacts its performance significantly. In addition to carefully considering your use of backtracking and testing the regular expression against near-matching input, you should always set a time-out value to ensure that the impact of excessive backtracking, if it occurs, is minimized.

The regular expression time-out interval defines the period of time that the regular expression engine will look for a single match before it times out. The default time-out interval is Regex.InfiniteMatchTimeout, which means that the regular expression will not time out. You can override this value and define a time-out interval as follows:

- By providing a time-out value when you instantiate a Regex object by calling the Regex.Regex(String, RegexOptions, TimeSpan) constructor.
- By calling a static pattern matching method, such as Regex.Match(String, String, RegexOptions, TimeSpan) or Regex.Replace(String, String, String, RegexOptions, TimeSpan), that includes a `matchTimeout` parameter.
- For compiled regular expressions that are created by calling the Regex.CompileToAssembly method, by calling the constructor that has a parameter of type TimeSpan.

If you have defined a time-out interval and a match is not found at the end of that interval, the regular expression method throws a RegexMatchTimeoutException exception. In your exception handler, you can choose to retry the match with a longer time-out interval, abandon the match attempt and assume that there is no match, or abandon the match attempt and log the exception information for future analysis.

The following example defines a `GetWordData` method that instantiates a regular expression with a time-out interval of 350 milliseconds to calculate the number of words and average number of characters in a word in a text document. If the matching operation times out, the time-out interval is increased by 350 milliseconds and the Regex object is re-instantiated. If the new time-out interval exceeds 1 second, the method re-throws the exception to the caller.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      RegexUtilities util = new RegexUtilities();
      string title = "Doyle - The Hound of the Baskervilles.txt";
      try {
         var info = util.GetWordData(title);
         Console.WriteLine("Words:               {0:N0}", info.Item1);
         Console.WriteLine("Average Word Length: {0:N2} characters", info.Item2);
      }
      catch (IOException e) {
         Console.WriteLine("IOException reading file '{0}'", title);
         Console.WriteLine(e.Message);
      }
      catch (RegexMatchTimeoutException e) {
         Console.WriteLine("The operation timed out after {0:N0} milliseconds",
                           e.MatchTimeout.TotalMilliseconds);
      }
   }
}

public class RegexUtilities
{
   public Tuple<int, double> GetWordData(string filename)
   {
      const int MAX_TIMEOUT = 1000;   // Maximum timeout interval in milliseconds.
      const int INCREMENT = 350;      // Milliseconds increment of timeout.

      List<string> exclusions = new List<string>( new string[] { "a", "an", "the" });
      int[] wordLengths = new int[29];        // Allocate an array of more than ample size.
      string input = null;
      StreamReader sr = null;
      try {
         sr = new StreamReader(filename);
         input = sr.ReadToEnd();
      }
      catch (FileNotFoundException e) {
         string msg = String.Format("Unable to find the file '{0}'", filename);
```

```
         throw new IOException(msg, e);
      }
      catch (IOException e) {
         throw new IOException(e.Message, e);
      }
      finally {
         if (sr != null) sr.Close();
      }

      int timeoutInterval = INCREMENT;
      bool init = false;
      Regex rgx = null;
      Match m = null;
      int indexPos = 0;
      do {
         try {
            if (! init) {
               rgx = new Regex(@"\b\w+\b", RegexOptions.None,
                               TimeSpan.FromMilliseconds(timeoutInterval));
               m = rgx.Match(input, indexPos);
               init = true;
            }
            else {
               m = m.NextMatch();
            }
            if (m.Success) {
               if ( !exclusions.Contains(m.Value.ToLower()))
                  wordLengths[m.Value.Length]++;

               indexPos += m.Length + 1;
            }
         }
         catch (RegexMatchTimeoutException e) {
            if (e.MatchTimeout.TotalMilliseconds < MAX_TIMEOUT) {
               timeoutInterval += INCREMENT;
               init = false;
            }
            else {
               // Rethrow the exception.
               throw;
            }
         }
      } while (m.Success);

      // If regex completed successfully, calculate number of words and average length.
      int nWords = 0;
      long totalLength = 0;

      for (int ctr = wordLengths.GetLowerBound(0); ctr <= wordLengths.GetUpperBound(0); ctr++) {
         nWords += wordLengths[ctr];
         totalLength += ctr * wordLengths[ctr];
      }
      return new Tuple<int, double>(nWords, totalLength/nWords);
   }
}
```

```
Imports System.Collections.Generic
Imports System.IO
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim util As New RegexUtilities()
      Dim title As String = "Doyle - The Hound of the Baskervilles.txt"
      Try
         Dim info = util.GetWordData(title)
         Console.WriteLine("Words:               {0:N0}", info.Item1)
```

```
         Console.WriteLine("Average Word Length: {0:N2} characters", info.Item2)
      Catch e As IOException
         Console.WriteLine("IOException reading file '{0}'", title)
         Console.WriteLine(e.Message)
      Catch e As RegexMatchTimeoutException
         Console.WriteLine("The operation timed out after {0:N0} milliseconds",
                           e.MatchTimeout.TotalMilliseconds)
      End Try
   End Sub
End Module

Public Class RegexUtilities
   Public Function GetWordData(filename As String) As Tuple(Of Integer, Double)
      Const MAX_TIMEOUT As Integer = 1000  ' Maximum timeout interval in milliseconds.
      Const INCREMENT As Integer = 350     ' Milliseconds increment of timeout.

      Dim exclusions As New List(Of String)({"a", "an", "the" })
      Dim wordLengths(30) As Integer        ' Allocate an array of more than ample size.
      Dim input As String = Nothing
      Dim sr As StreamReader = Nothing
      Try
         sr = New StreamReader(filename)
         input = sr.ReadToEnd()
      Catch e As FileNotFoundException
         Dim msg As String = String.Format("Unable to find the file '{0}'", filename)
         Throw New IOException(msg, e)
      Catch e As IOException
         Throw New IOException(e.Message, e)
      Finally
         If sr IsNot Nothing Then sr.Close()
      End Try

      Dim timeoutInterval As Integer = INCREMENT
      Dim init As Boolean = False
      Dim rgx As Regex = Nothing
      Dim m As Match = Nothing
      Dim indexPos As Integer = 0
      Do
         Try
            If Not init Then
               rgx = New Regex("\b\w+\b", RegexOptions.None,
                               TimeSpan.FromMilliseconds(timeoutInterval))
               m = rgx.Match(input, indexPos)
               init = True
            Else
               m = m.NextMatch()
            End If
            If m.Success Then
               If Not exclusions.Contains(m.Value.ToLower()) Then
                  wordLengths(m.Value.Length) += 1
               End If
               indexPos += m.Length + 1
            End If
         Catch e As RegexMatchTimeoutException
            If e.MatchTimeout.TotalMilliseconds < MAX_TIMEOUT Then
               timeoutInterval += INCREMENT
               init = False
            Else
               ' Rethrow the exception.
               Throw
            End If
         End Try
      Loop While m.Success

      ' If regex completed successfully, calculate number of words and average length.
      Dim nWords As Integer
      Dim totalLength As Long

      For ctr As Integer = wordLengths.GetLowerBound(0) To wordLengths.GetUpperBound(0)
```

```
            nWords += wordLengths(ctr)
            totalLength += ctr * wordLengths(ctr)
        Next
        Return New Tuple(Of Integer, Double)(nWords, totalLength/nWords)
    End Function
End Class
```

Back to top

Capture Only When Necessary

Regular expressions in .NET support a number of grouping constructs, which let you group a regular expression pattern into one or more subexpressions. The most commonly used grouping constructs in .NET regular expression language are `(` *subexpression* `)`, which defines a numbered capturing group, and `(?<` *name* `>` *subexpression* `)`, which defines a named capturing group. Grouping constructs are essential for creating backreferences and for defining a subexpression to which a quantifier is applied.

However, the use of these language elements has a cost. They cause the GroupCollection object returned by the Match.Groups property to be populated with the most recent unnamed or named captures, and if a single grouping construct has captured multiple substrings in the input string, they also populate the CaptureCollection object returned by the Group.Captures property of a particular capturing group with multiple Capture objects.

Often, grouping constructs are used in a regular expression only so that quantifiers can be applied to them, and the groups captured by these subexpressions are not subsequently used. For example, the regular expression `\b(\w+[;,]?\s?)+[.?!]` is designed to capture an entire sentence. The following table describes the language elements in this regular expression pattern and their effect on the Match object's Match.Groups and Group.Captures collections.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`\w+`	Match one or more word characters.
`[;,]?`	Match zero or one comma or semicolon.
`\s?`	Match zero or one white-space character.
`(\w+[;,]?\s?)+`	Match one or more occurrences of one or more word characters followed by an optional comma or semicolon followed by an optional white-space character. This defines the first capturing group, which is necessary so that the combination of multiple word characters (that is, a word) followed by an optional punctuation symbol will be repeated until the regular expression engine reaches the end of a sentence.
`[.?!]`	Match a period, question mark, or exclamation point.

As the following example shows, when a match is found, both the GroupCollection and CaptureCollection objects are populated with captures from the match. In this case, the capturing group `(\w+[;,]?\s?)` exists so that the `+` quantifier can be applied to it, which enables the regular expression pattern to match each word in a sentence. Otherwise, it would match the last word in a sentence.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is one sentence. This is another.";
      string pattern = @"\b(\w+[;,]?\s?)+[.?!]";

      foreach (Match match in Regex.Matches(input, pattern)) {
         Console.WriteLine("Match: '{0}' at index {1}.",
                           match.Value, match.Index);
         int grpCtr = 0;
         foreach (Group grp in match.Groups) {
            Console.WriteLine("   Group {0}: '{1}' at index {2}.",
                              grpCtr, grp.Value, grp.Index);
            int capCtr = 0;
            foreach (Capture cap in grp.Captures) {
               Console.WriteLine("      Capture {0}: '{1}' at {2}.",
                                 capCtr, cap.Value, cap.Index);
               capCtr++;
            }
            grpCtr++;
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Match: 'This is one sentence.' at index 0.
//          Group 0: 'This is one sentence.' at index 0.
//             Capture 0: 'This is one sentence.' at 0.
//          Group 1: 'sentence' at index 12.
//             Capture 0: 'This ' at 0.
//             Capture 1: 'is ' at 5.
//             Capture 2: 'one ' at 8.
//             Capture 3: 'sentence' at 12.
//
//       Match: 'This is another.' at index 22.
//          Group 0: 'This is another.' at index 22.
//             Capture 0: 'This is another.' at 22.
//          Group 1: 'another' at index 30.
//             Capture 0: 'This ' at 22.
//             Capture 1: 'is ' at 27.
//             Capture 2: 'another' at 30.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "This is one sentence. This is another."
      Dim pattern As String = "\b(\w+[;,]?\s?)+[.?!]"

      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("Match: '{0}' at index {1}.",
                           match.Value, match.Index)
         Dim grpCtr As Integer = 0
         For Each grp As Group In match.Groups
            Console.WriteLine("   Group {0}: '{1}' at index {2}.",
                              grpCtr, grp.Value, grp.Index)
            Dim capCtr As Integer = 0
            For Each cap As Capture In grp.Captures
               Console.WriteLine("      Capture {0}: '{1}' at {2}.",
                                 capCtr, cap.Value, cap.Index)
               capCtr += 1
            Next
            grpCtr += 1
         Next
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays the following output:
'       Match: 'This is one sentence.' at index 0.
'          Group 0: 'This is one sentence.' at index 0.
'             Capture 0: 'This is one sentence.' at 0.
'          Group 1: 'sentence' at index 12.
'             Capture 0: 'This ' at 0.
'             Capture 1: 'is ' at 5.
'             Capture 2: 'one ' at 8.
'             Capture 3: 'sentence' at 12.
'
'       Match: 'This is another.' at index 22.
'          Group 0: 'This is another.' at index 22.
'             Capture 0: 'This is another.' at 22.
'          Group 1: 'another' at index 30.
'             Capture 0: 'This ' at 22.
'             Capture 1: 'is ' at 27.
'             Capture 2: 'another' at 30.
```

When you use subexpressions only to apply quantifiers to them, and you are not interested in the captured text, you should disable group captures. For example, the `(?:subexpression)` language element prevents the group to which it applies from capturing matched substrings. In the following example, the regular expression pattern from the previous example is changed to `\b(?:\w+[;,]?\s?)+[.?!]`. As the output shows, it prevents the regular expression engine from populating the GroupCollection and CaptureCollection collections.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is one sentence. This is another.";
      string pattern = @"\b(?:\w+[;,]?\s?)+[.?!]";

      foreach (Match match in Regex.Matches(input, pattern)) {
         Console.WriteLine("Match: '{0}' at index {1}.",
                           match.Value, match.Index);
         int grpCtr = 0;
         foreach (Group grp in match.Groups) {
            Console.WriteLine("   Group {0}: '{1}' at index {2}.",
                              grpCtr, grp.Value, grp.Index);
            int capCtr = 0;
            foreach (Capture cap in grp.Captures) {
               Console.WriteLine("      Capture {0}: '{1}' at {2}.",
                                 capCtr, cap.Value, cap.Index);
               capCtr++;
            }
            grpCtr++;
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Match: 'This is one sentence.' at index 0.
//          Group 0: 'This is one sentence.' at index 0.
//             Capture 0: 'This is one sentence.' at 0.
//
//       Match: 'This is another.' at index 22.
//          Group 0: 'This is another.' at index 22.
//             Capture 0: 'This is another.' at 22.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "This is one sentence. This is another."
      Dim pattern As String = "\b(?:\w+[;,]?\s?)+[.?!]"

      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("Match: '{0}' at index {1}.",
                           match.Value, match.Index)
         Dim grpCtr As Integer = 0
         For Each grp As Group In match.Groups
            Console.WriteLine("   Group {0}: '{1}' at index {2}.",
                              grpCtr, grp.Value, grp.Index)
            Dim capCtr As Integer = 0
            For Each cap As Capture In grp.Captures
               Console.WriteLine("      Capture {0}: '{1}' at {2}.",
                                 capCtr, cap.Value, cap.Index)
               capCtr += 1
            Next
            grpCtr += 1
         Next
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays the following output:
'       Match: 'This is one sentence.' at index 0.
'          Group 0: 'This is one sentence.' at index 0.
'             Capture 0: 'This is one sentence.' at 0.
'
'       Match: 'This is another.' at index 22.
'          Group 0: 'This is another.' at index 22.
'             Capture 0: 'This is another.' at 22.
```

You can disable captures in one of the following ways:

- Use the `(?:subexpression)` language element. This element prevents the capture of matched substrings in the group to which it applies. It does not disable substring captures in any nested groups.

- Use the ExplicitCapture option. It disables all unnamed or implicit captures in the regular expression pattern. When you use this option, only substrings that match named groups defined with the `(?<name>subexpression)` language element can be captured. The ExplicitCapture flag can be passed to the `options` parameter of a Regex class constructor or to the `options` parameter of a Regex static matching method.

- Use the `n` option in the `(?imnsx)` language element. This option disables all unnamed or implicit captures from the point in the regular expression pattern at which the element appears. Captures are disabled either until the end of the pattern or until the `(-n)` option enables unnamed or implicit captures. For more information, see Miscellaneous Constructs.

- Use the `n` option in the `(?imnsx:subexpression)` language element. This option disables all unnamed or implicit captures in `subexpression`. Captures by any unnamed or implicit nested capturing groups are disabled as well.

Back to top

Related Topics

TITLE	DESCRIPTION
Details of Regular Expression Behavior	Examines the implementation of the regular expression engine in .NET. The topic focuses on the flexibility of regular expressions and explains the developer's responsibility for ensuring the efficient and robust operation of the regular expression engine.
Backtracking	Explains what backtracking is and how it affects regular expression performance, and examines language elements that provide alternatives to backtracking.
Regular Expression Language - Quick Reference	Describes the elements of the regular expression language in .NET and provides links to detailed documentation for each language element.

The Regular Expression Object Model

9/6/2018 • 28 minutes to read • Edit Online

This topic describes the object model used in working with .NET regular expressions. It contains the following sections:

- The Regular Expression Engine
- The MatchCollection and Match Objects
- The Group Collection
- The Captured Group
- The Capture Collection
- The Individual Capture

The Regular Expression Engine

The regular expression engine in .NET is represented by the Regex class. The regular expression engine is responsible for parsing and compiling a regular expression, and for performing operations that match the regular expression pattern with an input string. The engine is the central component in the .NET regular expression object model.

You can use the regular expression engine in either of two ways:

- By calling the static methods of the Regex class. The method parameters include the input string and the regular expression pattern. The regular expression engine caches regular expressions that are used in static method calls, so repeated calls to static regular expression methods that use the same regular expression offer relatively good performance.
- By instantiating a Regex object, by passing a regular expression to the class constructor. In this case, the Regex object is immutable (read-only) and represents a regular expression engine that is tightly coupled with a single regular expression. Because regular expressions used by Regex instances are not cached, you should not instantiate a Regex object multiple times with the same regular expression.

You can call the methods of the Regex class to perform the following operations:

- Determine whether a string matches a regular expression pattern.
- Extract a single match or the first match.
- Extract all matches.
- Replace a matched substring.
- Split a single string into an array of strings.

These operations are described in the following sections.

Matching a Regular Expression Pattern

The Regex.IsMatch method returns `true` if the string matches the pattern, or `false` if it does not. The IsMatch method is often used to validate string input. For example, the following code ensures that a string matches a valid social security number in the United States.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] values = { "111-22-3333", "111-2-3333"};
      string pattern = @"^\d{3}-\d{2}-\d{4}$";
      foreach (string value in values) {
         if (Regex.IsMatch(value, pattern))
            Console.WriteLine("{0} is a valid SSN.", value);
         else
            Console.WriteLine("{0}: Invalid", value);
      }
   }
}
// The example displays the following output:
//       111-22-3333 is a valid SSN.
//       111-2-3333: Invalid
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim values() As String = { "111-22-3333", "111-2-3333"}
      Dim pattern As String = "^\d{3}-\d{2}-\d{4}$"
      For Each value As String In values
         If Regex.IsMatch(value, pattern) Then
            Console.WriteLine("{0} is a valid SSN.", value)
         Else
            Console.WriteLine("{0}: Invalid", value)
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'       111-22-3333 is a valid SSN.
'       111-2-3333: Invalid
```

The regular expression pattern `^\d{3}-\d{2}-\d{4}$` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`^`	Match the beginning of the input string.
`\d{3}`	Match three decimal digits.
`-`	Match a hyphen.
`\d{2}`	Match two decimal digits.
`-`	Match a hyphen.
`\d{4}`	Match four decimal digits.
`$`	Match the end of the input string.

Extracting a Single Match or the First Match

The Regex.Match method returns a Match object that contains information about the first substring that matches a regular expression pattern. If the `Match.Success` property returns `true`, indicating that a match was found, you can retrieve information about subsequent matches by calling the Match.NextMatch method. These method calls can continue until the `Match.Success` property returns `false`. For example, the following code uses the Regex.Match(String, String) method to find the first occurrence of a duplicated word in a string. It then calls the Match.NextMatch method to find any additional occurrences. The example examines the `Match.Success` property after each method call to determine whether the current match was successful and whether a call to the Match.NextMatch method should follow.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is a a farm that that raises dairy cattle.";
      string pattern = @"\b(\w+)\W+(\1)\b";
      Match match = Regex.Match(input, pattern);
      while (match.Success)
      {
         Console.WriteLine("Duplicate '{0}' found at position {1}.",
                           match.Groups[1].Value, match.Groups[2].Index);
         match = match.NextMatch();
      }
   }
}
// The example displays the following output:
//       Duplicate 'a' found at position 10.
//       Duplicate 'that' found at position 22.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "This is a a farm that that raises dairy cattle."
      Dim pattern As String = "\b(\w+)\W+(\1)\b"
      Dim match As Match = Regex.Match(input, pattern)
      Do While match.Success
         Console.WriteLine("Duplicate '{0}' found at position {1}.", _
                           match.Groups(1).Value, match.Groups(2).Index)
         match = match.NextMatch()
      Loop
   End Sub
End Module
' The example displays the following output:
'       Duplicate 'a' found at position 10.
'       Duplicate 'that' found at position 22.
```

The regular expression pattern `\b(\w+)\W+(\1)\b` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match on a word boundary.
`(\w+)`	Match one or more word characters. This is the first capturing group.
`\W+`	Match one or more non-word characters.

PATTERN	DESCRIPTION
`(\1)`	Match the first captured string. This is the second capturing group.
`\b`	End the match on a word boundary.

Extracting All Matches

The Regex.Matches method returns a MatchCollection object that contains information about all matches that the regular expression engine found in the input string. For example, the previous example could be rewritten to call the Matches method instead of the Match and NextMatch methods.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is a a farm that that raises dairy cattle.";
      string pattern = @"\b(\w+)\W+(\1)\b";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Duplicate '{0}' found at position {1}.",
                           match.Groups[1].Value, match.Groups[2].Index);
   }
}
// The example displays the following output:
//       Duplicate 'a' found at position 10.
//       Duplicate 'that' found at position 22.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "This is a a farm that that raises dairy cattle."
      Dim pattern As String = "\b(\w+)\W+(\1)\b"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("Duplicate '{0}' found at position {1}.", _
                           match.Groups(1).Value, match.Groups(2).Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       Duplicate 'a' found at position 10.
'       Duplicate 'that' found at position 22.
```

Replacing a Matched Substring

The Regex.Replace method replaces each substring that matches the regular expression pattern with a specified string or regular expression pattern, and returns the entire input string with replacements. For example, the following code adds a U.S. currency symbol before a decimal number in a string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\d+\.\d{2}\b";
      string replacement = "$$$&";
      string input = "Total Cost: 103.64";
      Console.WriteLine(Regex.Replace(input, pattern, replacement));
   }
}
// The example displays the following output:
//       Total Cost: $103.64
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b\d+\.\d{2}\b"
      Dim replacement As String = "$$$&"
      Dim input As String = "Total Cost: 103.64"
      Console.WriteLine(Regex.Replace(input, pattern, replacement))
   End Sub
End Module
' The example displays the following output:
'       Total Cost: $103.64
```

The regular expression pattern `\b\d+\.\d{2}\b` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`\d+`	Match one or more decimal digits.
`\.`	Match a period.
`\d{2}`	Match two decimal digits.
`\b`	End the match at a word boundary.

The replacement pattern `$$$&` is interpreted as shown in the following table.

PATTERN	REPLACEMENT STRING
`$$`	The dollar sign ($) character.
`$&`	The entire matched substring.

Splitting a Single String into an Array of Strings

The Regex.Split method splits the input string at the positions defined by a regular expression match. For example, the following code places the items in a numbered list into a string array.

```csharp
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "1. Eggs 2. Bread 3. Milk 4. Coffee 5. Tea";
      string pattern = @"\b\d{1,2}\.\s";
      foreach (string item in Regex.Split(input, pattern))
      {
         if (! String.IsNullOrEmpty(item))
            Console.WriteLine(item);
      }
   }
}
// The example displays the following output:
//       Eggs
//       Bread
//       Milk
//       Coffee
//       Tea
```

```vb
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "1. Eggs 2. Bread 3. Milk 4. Coffee 5. Tea"
      Dim pattern As String = "\b\d{1,2}\.\s"
      For Each item As String In Regex.Split(input, pattern)
         If Not String.IsNullOrEmpty(item) Then
            Console.WriteLine(item)
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'       Eggs
'       Bread
'       Milk
'       Coffee
'       Tea
```

The regular expression pattern `\b\d{1,2}\.\s` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`\d{1,2}`	Match one or two decimal digits.
`\.`	Match a period.
`\s`	Match a white-space character.

The MatchCollection and Match Objects

Regex methods return two objects that are part of the regular expression object model: the MatchCollection object, and the Match object.

The Match Collection

The Regex.Matches method returns a MatchCollection object that contains Match objects that represent all the matches that the regular expression engine found, in the order in which they occur in the input string. If there are no matches, the method returns a MatchCollection object with no members. The MatchCollection.Item[Int32] property lets you access individual members of the collection by index, from zero to one less than the value of the MatchCollection.Count property. Item[Int32] is the collection's indexer (in C#) and default property (in Visual Basic).

By default, the call to the Regex.Matches method uses lazy evaluation to populate the MatchCollection object. Access to properties that require a fully populated collection, such as the MatchCollection.Count and MatchCollection.Item[Int32] properties, may involve a performance penalty. As a result, we recommend that you access the collection by using the IEnumerator object that is returned by the MatchCollection.GetEnumerator method. Individual languages provide constructs, such as `For Each` in Visual Basic and `foreach` in C#, that wrap the collection's IEnumerator interface.

The following example uses the Regex.Matches(String) method to populate a MatchCollection object with all the matches found in an input string. The example enumerates the collection, copies the matches to a string array, and records the character positions in an integer array.

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
       MatchCollection matches;
       List<string> results = new List<string>();
       List<int> matchposition = new List<int>();

       // Create a new Regex object and define the regular expression.
       Regex r = new Regex("abc");
       // Use the Matches method to find all matches in the input string.
       matches = r.Matches("123abc4abcd");
       // Enumerate the collection to retrieve all matches and positions.
       foreach (Match match in matches)
       {
          // Add the match string to the string array.
           results.Add(match.Value);
           // Record the character position where the match was found.
           matchposition.Add(match.Index);
       }
       // List the results.
       for (int ctr = 0; ctr < results.Count; ctr++)
         Console.WriteLine("'{0}' found at position {1}.",
                           results[ctr], matchposition[ctr]);
   }
}
// The example displays the following output:
//       'abc' found at position 3.
//       'abc' found at position 7.
```

```
Imports System.Collections.Generic
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
       Dim matches As MatchCollection
       Dim results As New List(Of String)
       Dim matchposition As New List(Of Integer)

       ' Create a new Regex object and define the regular expression.
       Dim r As New Regex("abc")
       ' Use the Matches method to find all matches in the input string.
       matches = r.Matches("123abc4abcd")
       ' Enumerate the collection to retrieve all matches and positions.
       For Each match As Match In matches
          ' Add the match string to the string array.
           results.Add(match.Value)
           ' Record the character position where the match was found.
           matchposition.Add(match.Index)
       Next
       ' List the results.
       For ctr As Integer = 0 To results.Count - 1
         Console.WriteLine("'{0}' found at position {1}.", _
                           results(ctr), matchposition(ctr))
       Next
   End Sub
End Module
' The example displays the following output:
'       'abc' found at position 3.
'       'abc' found at position 7.
```

The Match

The Match class represents the result of a single regular expression match. You can access Match objects in two ways:

- By retrieving them from the MatchCollection object that is returned by the Regex.Matches method. To retrieve individual Match objects, iterate the collection by using a `foreach` (in C#) or `For Each`...`Next` (in Visual Basic) construct, or use the MatchCollection.Item[Int32] property to retrieve a specific Match object either by index or by name. You can also retrieve individual Match objects from the collection by iterating the collection by index, from zero to one less that the number of objects in the collection. However, this method does not take advantage of lazy evaluation, because it accesses the MatchCollection.Count property.

  The following example retrieves individual Match objects from a MatchCollection object by iterating the collection using the `foreach` or `For Each`...`Next` construct. The regular expression simply matches the string "abc" in the input string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "abc";
      string input = "abc123abc456abc789";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("{0} found at position {1}.",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       abc found at position 0.
//       abc found at position 6.
//       abc found at position 12.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "abc"
      Dim input As String = "abc123abc456abc789"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("{0} found at position {1}.", _
                           match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       abc found at position 0.
'       abc found at position 6.
'       abc found at position 12.
```

- By calling the Regex.Match method, which returns a Match object that represents the first match in a string or a portion of a string. You can determine whether the match has been found by retrieving the value of the `Match.Success` property. To retrieve Match objects that represent subsequent matches, call the Match.NextMatch method repeatedly, until the `Success` property of the returned Match object is `false`.

  The following example uses the Regex.Match(String, String) and Match.NextMatch methods to match the string "abc" in the input string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "abc";
      string input = "abc123abc456abc789";
      Match match = Regex.Match(input, pattern);
      while (match.Success)
      {
         Console.WriteLine("{0} found at position {1}.",
                           match.Value, match.Index);
         match = match.NextMatch();
      }
   }
}
// The example displays the following output:
//       abc found at position 0.
//       abc found at position 6.
//       abc found at position 12.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "abc"
      Dim input As String = "abc123abc456abc789"
      Dim match As Match = Regex.Match(input, pattern)
      Do While match.Success
         Console.WriteLine("{0} found at position {1}.", _
                           match.Value, match.Index)
         match = match.NextMatch()
      Loop
   End Sub
End Module
' The example displays the following output:
'       abc found at position 0.
'       abc found at position 6.
'       abc found at position 12.
```

Two properties of the Match class return collection objects:

- The Match.Groups property returns a GroupCollection object that contains information about the substrings that match capturing groups in the regular expression pattern.

- The `Match.Captures` property returns a CaptureCollection object that is of limited use. The collection is not populated for a Match object whose `Success` property is `false`. Otherwise, it contains a single Capture object that has the same information as the Match object.

For more information about these objects, see The Group Collection and The Capture Collection sections later in this topic.

Two additional properties of the Match class provide information about the match. The `Match.Value` property returns the substring in the input string that matches the regular expression pattern. The `Match.Index` property returns the zero-based starting position of the matched string in the input string.

The Match class also has two pattern-matching methods:

- The Match.NextMatch method finds the match after the match represented by the current Match object, and returns a Match object that represents that match.

- The Match.Result method performs a specified replacement operation on the matched string and returns the result.

The following example uses the Match.Result method to prepend a $ symbol and a space before every number that includes two fractional digits.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\d+(,\d{3})*\.\d{2}\b";
      string input = "16.32\n194.03\n1,903,672.08";

      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Result("$$ $&"));
   }
}
// The example displays the following output:
//       $ 16.32
//       $ 194.03
//       $ 1,903,672.08
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b\d+(,\d{3})*\.\d{2}\b"
      Dim input As String = "16.32" + vbCrLf + "194.03" + vbCrLf + "1,903,672.08"

      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine(match.Result("$$ $&"))
      Next
   End Sub
End Module
' The example displays the following output:
'       $ 16.32
'       $ 194.03
'       $ 1,903,672.08
```

The regular expression pattern `\b\d+(,\d{3})*\.\d{2}\b` is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`\d+`	Match one or more decimal digits.
`(,\d{3})*`	Match zero or more occurrences of a comma followed by three decimal digits.
`\.`	Match the decimal point character.
`\d{2}`	Match two decimal digits.
`\b`	End the match at a word boundary.

The replacement pattern `$$ $&` indicates that the matched substring should be replaced by a dollar sign ($) symbol (the `$$` pattern), a space, and the value of the match (the `$&` pattern).

Back to top

The Group Collection

The Match.Groups property returns a GroupCollection object that contains Group objects that represent captured groups in a single match. The first Group object in the collection (at index 0) represents the entire match. Each object that follows represents the results of a single capturing group.

You can retrieve individual Group objects in the collection by using the GroupCollection.Item[String] property. You can retrieve unnamed groups by their ordinal position in the collection, and retrieve named groups either by name or by ordinal position. Unnamed captures appear first in the collection, and are indexed from left to right in the order in which they appear in the regular expression pattern. Named captures are indexed after unnamed captures, from left to right in the order in which they appear in the regular expression pattern. To determine what numbered groups are available in the collection returned for a particular regular expression matching method, you can call the instance Regex.GetGroupNumbers method. To determine what named groups are available in the collection, you can call the instance Regex.GetGroupNames method. Both methods are particularly useful in general-purpose routines that analyze the matches found by any regular expression.

The GroupCollection.Item[String] property is the indexer of the collection in C# and the collection object's default property in Visual Basic. This means that individual Group objects can be accessed by index (or by name, in the case of named groups) as follows:

```
Group group = match.Groups[ctr];
```

```
Dim group As Group = match.Groups(ctr)
```

The following example defines a regular expression that uses grouping constructs to capture the month, day, and year of a date.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+)\s(\d{1,2}),\s(\d{4})\b";
      string input = "Born: July 28, 1989";
      Match match = Regex.Match(input, pattern);
      if (match.Success)
         for (int ctr = 0; ctr <  match.Groups.Count; ctr++)
            Console.WriteLine("Group {0}: {1}", ctr, match.Groups[ctr].Value);
    }
}
// The example displays the following output:
//       Group 0: July 28, 1989
//       Group 1: July
//       Group 2: 28
//       Group 3: 1989
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b(\w+)\s(\d{1,2}),\s(\d{4})\b"
      Dim input As String = "Born: July 28, 1989"
      Dim match As Match = Regex.Match(input, pattern)
      If match.Success Then
         For ctr As Integer = 0 To match.Groups.Count - 1
            Console.WriteLine("Group {0}: {1}", ctr, match.Groups(ctr).Value)
         Next
      End If
   End Sub
End Module
' The example displays the following output:
'       Group 0: July 28, 1989
'       Group 1: July
'       Group 2: 28
'       Group 3: 1989
```

The regular expression pattern `\b(\w+)\s(\d{1,2}),\s(\d{4})\b` is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`(\w+)`	Match one or more word characters. This is the first capturing group.
`\s`	Match a white-space character.
`(\d{1,2})`	Match one or two decimal digits. This is the second capturing group.
`,`	Match a comma.
`\s`	Match a white-space character.
`(\d{4})`	Match four decimal digits. This is the third capturing group.
`\b`	End the match on a word boundary.

Back to top

The Captured Group

The Group class represents the result from a single capturing group. Group objects that represent the capturing groups defined in a regular expression are returned by the Item[String] property of the GroupCollection object returned by the Match.Groups property. The Item[String] property is the indexer (in C#) and the default property (in Visual Basic) of the Group class. You can also retrieve individual members by iterating the collection using the `foreach` or `For Each` construct. For an example, see the previous section.

The following example uses nested grouping constructs to capture substrings into groups. The regular expression pattern `(a(b))c` matches the string "abc". It assigns the substring "ab" to the first capturing group, and the substring "b" to the second capturing group.

```csharp
List<int> matchposition = new List<int>();
List<string> results = new List<string>();
// Define substrings abc, ab, b.
Regex r = new Regex("(a(b))c");
Match m = r.Match("abdabc");
for (int i = 0; m.Groups[i].Value != ""; i++)
{
   // Add groups to string array.
   results.Add(m.Groups[i].Value);
   // Record character position.
   matchposition.Add(m.Groups[i].Index);
}

// Display the capture groups.
for (int ctr = 0; ctr < results.Count; ctr++)
   Console.WriteLine("{0} at position {1}",
                     results[ctr], matchposition[ctr]);
// The example displays the following output:
//       abc at position 3
//       ab at position 3
//       b at position 4
```

```vb
 Dim matchposition As New List(Of Integer)
 Dim results As New List(Of String)
 ' Define substrings abc, ab, b.
 Dim r As New Regex("(a(b))c")
 Dim m As Match = r.Match("abdabc")
 Dim i As Integer = 0
 While Not (m.Groups(i).Value = "")
    ' Add groups to string array.
    results.Add(m.Groups(i).Value)
    ' Record character position.
    matchposition.Add(m.Groups(i).Index)
     i += 1
 End While

 ' Display the capture groups.
 For ctr As Integer = 0 to results.Count - 1
    Console.WriteLine("{0} at position {1}", _
                      results(ctr), matchposition(ctr))
 Next
' The example displays the following output:
'       abc at position 3
'       ab at position 3
'       b at position 4
```

The following example uses named grouping constructs to capture substrings from a string that contains data in the format "DATANAME:VALUE", which the regular expression splits at the colon (:).

```csharp
Regex r = new Regex("^(?<name>\\w+):(?<value>\\w+)");
Match m = r.Match("Section1:119900");
Console.WriteLine(m.Groups["name"].Value);
Console.WriteLine(m.Groups["value"].Value);
// The example displays the following output:
//       Section1
//       119900
```

```
Dim r As New Regex("^(?<name>\w+):(?<value>\w+)")
Dim m As Match = r.Match("Section1:119900")
Console.WriteLine(m.Groups("name").Value)
Console.WriteLine(m.Groups("value").Value)
' The example displays the following output:
'       Section1
'       119900
```

The regular expression pattern `^(?<name>\w+):(?<value>\w+)` is defined as shown in the following table.

PATTERN	DESCRIPTION
`^`	Begin the match at the beginning of the input string.
`(?<name>\w+)`	Match one or more word characters. The name of this capturing group is `name`.
`:`	Match a colon.
`(?<value>\w+)`	Match one or more word characters. The name of this capturing group is `value`.

The properties of the Group class provide information about the captured group: The `Group.Value` property contains the captured substring, the `Group.Index` property indicates the starting position of the captured group in the input text, the `Group.Length` property contains the length of the captured text, and the `Group.Success` property indicates whether a substring matched the pattern defined by the capturing group.

Applying quantifiers to a group (for more information, see Quantifiers) modifies the relationship of one capture per capturing group in two ways:

- If the `*` or `*?` quantifier (which specifies zero or more matches) is applied to a group, a capturing group may not have a match in the input string. When there is no captured text, the properties of the Group object are set as shown in the following table.

GROUP PROPERTY	VALUE
`Success`	`false`
`Value`	String.Empty
`Length`	0

  The following example provides an illustration. In the regular expression pattern `aaa(bbb)*ccc`, the first capturing group (the substring "bbb") can be matched zero or more times. Because the input string "aaaccc" matches the pattern, the capturing group does not have a match.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "aaa(bbb)*ccc";
      string input = "aaaccc";
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match value: {0}", match.Value);
      if (match.Groups[1].Success)
         Console.WriteLine("Group 1 value: {0}", match.Groups[1].Value);
      else
         Console.WriteLine("The first capturing group has no match.");
   }
}
// The example displays the following output:
//       Match value: aaaccc
//       The first capturing group has no match.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "aaa(bbb)*ccc"
      Dim input As String = "aaaccc"
      Dim match As Match = Regex.Match(input, pattern)
      Console.WriteLine("Match value: {0}", match.Value)
      If match.Groups(1).Success Then
         Console.WriteLine("Group 1 value: {0}", match.Groups(1).Value)
      Else
         Console.WriteLine("The first capturing group has no match.")
     End If
   End Sub
End Module
' The example displays the following output:
'       Match value: aaaccc
'       The first capturing group has no match.
```

- Quantifiers can match multiple occurrences of a pattern that is defined by a capturing group. In this case, the `Value` and `Length` properties of a Group object contain information only about the last captured substring. For example, the following regular expression matches a single sentence that ends in a period. It uses two grouping constructs: The first captures individual words along with a white-space character; the second captures individual words. As the output from the example shows, although the regular expression succeeds in capturing an entire sentence, the second capturing group captures only the last word.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b((\w+)\s?)+\.";
      string input = "This is a sentence. This is another sentence.";
      Match match = Regex.Match(input, pattern);
      if (match.Success)
      {
         Console.WriteLine("Match: " + match.Value);
         Console.WriteLine("Group 2: " + match.Groups[2].Value);
      }
   }
}
// The example displays the following output:
//       Match: This is a sentence.
//       Group 2: sentence
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b((\w+)\s?)+\."
      Dim input As String = "This is a sentence. This is another sentence."
      Dim match As Match = Regex.Match(input, pattern)
      If match.Success Then
         Console.WriteLine("Match: " + match.Value)
         Console.WriteLine("Group 2: " + match.Groups(2).Value)
      End If
   End Sub
End Module
' The example displays the following output:
'       Match: This is a sentence.
'       Group 2: sentence
```

Back to top

The Capture Collection

The Group object contains information only about the last capture. However, the entire set of captures made by a capturing group is still available from the CaptureCollection object that is returned by the Group.Captures property. Each member of the collection is a Capture object that represents a capture made by that capturing group, in the order in which they were captured (and, therefore, in the order in which the captured strings were matched from left to right in the input string). You can retrieve individual Capture objects from the collection in either of two ways:

- By iterating through the collection using a construct such as `foreach` (in C#) or `For Each` (in Visual Basic).
- By using the CaptureCollection.Item[Int32] property to retrieve a specific object by index. The Item[Int32] property is the CaptureCollection object's default property (in Visual Basic) or indexer (in C#).

If a quantifier is not applied to a capturing group, the CaptureCollection object contains a single Capture object that is of little interest, because it provides information about the same match as its Group object. If a quantifier is applied to a capturing group, the CaptureCollection object contains all captures made by the capturing group, and the last member of the collection represents the same capture as the Group object.

For example, if you use the regular expression pattern `((a(b))c)+` (where the + quantifier specifies one or more matches) to capture matches from the string "abcabcabc", the CaptureCollection object for each Group object contains three members.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "((a(b))c)+";
      string input = "abcabcabc";

      Match match = Regex.Match(input, pattern);
      if (match.Success)
      {
         Console.WriteLine("Match: '{0}' at position {1}",
                           match.Value, match.Index);
         GroupCollection groups = match.Groups;
         for (int ctr = 0; ctr < groups.Count; ctr++) {
            Console.WriteLine("   Group {0}: '{1}' at position {2}",
                              ctr, groups[ctr].Value, groups[ctr].Index);
            CaptureCollection captures = groups[ctr].Captures;
            for (int ctr2 = 0; ctr2 < captures.Count; ctr2++) {
               Console.WriteLine("      Capture {0}: '{1}' at position {2}",
                                 ctr2, captures[ctr2].Value, captures[ctr2].Index);
            }
         }
      }
   }
}
// The example displays the following output:
//       Match: 'abcabcabc' at position 0
//          Group 0: 'abcabcabc' at position 0
//             Capture 0: 'abcabcabc' at position 0
//          Group 1: 'abc' at position 6
//             Capture 0: 'abc' at position 0
//             Capture 1: 'abc' at position 3
//             Capture 2: 'abc' at position 6
//          Group 2: 'ab' at position 6
//             Capture 0: 'ab' at position 0
//             Capture 1: 'ab' at position 3
//             Capture 2: 'ab' at position 6
//          Group 3: 'b' at position 7
//             Capture 0: 'b' at position 1
//             Capture 1: 'b' at position 4
//             Capture 2: 'b' at position 7
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "((a(b))c)+"
      Dim input As STring = "abcabcabc"

      Dim match As Match = Regex.Match(input, pattern)
      If match.Success Then
         Console.WriteLine("Match: '{0}' at position {1}", _
                           match.Value, match.Index)
         Dim groups As GroupCollection = match.Groups
         For ctr As Integer = 0 To groups.Count - 1
            Console.WriteLine("   Group {0}: '{1}' at position {2}", _
                              ctr, groups(ctr).Value, groups(ctr).Index)
            Dim captures As CaptureCollection = groups(ctr).Captures
            For ctr2 As Integer = 0 To captures.Count - 1
               Console.WriteLine("      Capture {0}: '{1}' at position {2}", _
                                 ctr2, captures(ctr2).Value, captures(ctr2).Index)
            Next
         Next
      End If
   End Sub
End Module
' The example dosplays the following output:
'       Match: 'abcabcabc' at position 0
'          Group 0: 'abcabcabc' at position 0
'             Capture 0: 'abcabcabc' at position 0
'          Group 1: 'abc' at position 6
'             Capture 0: 'abc' at position 0
'             Capture 1: 'abc' at position 3
'             Capture 2: 'abc' at position 6
'          Group 2: 'ab' at position 6
'             Capture 0: 'ab' at position 0
'             Capture 1: 'ab' at position 3
'             Capture 2: 'ab' at position 6
'          Group 3: 'b' at position 7
'             Capture 0: 'b' at position 1
'             Capture 1: 'b' at position 4
'             Capture 2: 'b' at position 7
```

The following example uses the regular expression `(Abc)+` to find one or more consecutive runs of the string "Abc" in the string "XYZAbcAbcAbcXYZAbcAb". The example illustrates the use of the Group.Captures property to return multiple groups of captured substrings.

```
      int counter;
      Match m;
      CaptureCollection cc;
      GroupCollection gc;

      // Look for groupings of "Abc".
      Regex r = new Regex("(Abc)+");
      // Define the string to search.
      m = r.Match("XYZAbcAbcAbcXYZAbcAb");
      gc = m.Groups;

      // Display the number of groups.
      Console.WriteLine("Captured groups = " + gc.Count.ToString());

      // Loop through each group.
      for (int i=0; i < gc.Count; i++)
      {
         cc = gc[i].Captures;
         counter = cc.Count;

         // Display the number of captures in this group.
         Console.WriteLine("Captures count = " + counter.ToString());

         // Loop through each capture in the group.
         for (int ii = 0; ii < counter; ii++)
         {
            // Display the capture and its position.
            Console.WriteLine(cc[ii] + "   Starts at character " +
                 cc[ii].Index);
         }
      }
   }
// The example displays the following output:
//       Captured groups = 2
//       Captures count = 1
//       AbcAbcAbc   Starts at character 3
//       Captures count = 3
//       Abc   Starts at character 3
//       Abc   Starts at character 6
//       Abc   Starts at character 9
```

```
Dim counter As Integer
Dim m As Match
Dim cc As CaptureCollection
Dim gc As GroupCollection

' Look for groupings of "Abc".
Dim r As New Regex("(Abc)+")
' Define the string to search.
m = r.Match("XYZAbcAbcAbcXYZAbcAb")
gc = m.Groups

' Display the number of groups.
Console.WriteLine("Captured groups = " & gc.Count.ToString())

' Loop through each group.
Dim i, ii As Integer
For i = 0 To gc.Count - 1
    cc = gc(i).Captures
    counter = cc.Count

    ' Display the number of captures in this group.
    Console.WriteLine("Captures count = " & counter.ToString())

    ' Loop through each capture in the group.
    For ii = 0 To counter - 1
        ' Display the capture and its position.
        Console.WriteLine(cc(ii).ToString() _
            & "   Starts at character " & cc(ii).Index.ToString())
    Next ii
Next i
' The example displays the following output:
'       Captured groups = 2
'       Captures count = 1
'       AbcAbcAbc   Starts at character 3
'       Captures count = 3
'       Abc   Starts at character 3
'       Abc   Starts at character 6
'       Abc   Starts at character 9
```

Back to top

The Individual Capture

The Capture class contains the results from a single subexpression capture. The Capture.Value property contains the matched text, and the Capture.Index property indicates the zero-based position in the input string at which the matched substring begins.

The following example parses an input string for the temperature of selected cities. A comma (",") is used to separate a city and its temperature, and a semicolon (";") is used to separate each city's data. The entire input string represents a single match. In the regular expression pattern `((\w+(\s\w+)*),(\d+);)+`, which is used to parse the string, the city name is assigned to the second capturing group, and the temperature is assigned to the fourth capturing group.

```csharp
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "Miami,78;Chicago,62;New York,67;San Francisco,59;Seattle,58;";
      string pattern = @"((\w+(\s\w+)*),(\d+);)+";
      Match match = Regex.Match(input, pattern);
      if (match.Success)
      {
         Console.WriteLine("Current temperatures:");
         for (int ctr = 0; ctr < match.Groups[2].Captures.Count; ctr++)
            Console.WriteLine("{0,-20} {1,3}", match.Groups[2].Captures[ctr].Value,
                              match.Groups[4].Captures[ctr].Value);
      }
   }
}
// The example displays the following output:
//       Current temperatures:
//       Miami                78
//       Chicago              62
//       New York             67
//       San Francisco        59
```

```vb
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "Miami,78;Chicago,62;New York,67;San Francisco,59;Seattle,58;"
      Dim pattern As String = "((\w+(\s\w+)*),(\d+);)+"
      Dim match As Match = Regex.Match(input, pattern)
      If match.Success Then
         Console.WriteLine("Current temperatures:")
         For ctr As Integer = 0 To match.Groups(2).Captures.Count - 1
            Console.WriteLine("{0,-20} {1,3}", match.Groups(2).Captures(ctr).Value, _
                              match.Groups(4).Captures(ctr).Value)
         Next
      End If
   End Sub
End Module
' The example displays the following output:
'       Current temperatures:
'       Miami                78
'       Chicago              62
'       New York             67
'       San Francisco        59
```

The regular expression is defined as shown in the following table.

PATTERN	DESCRIPTION
`\w+`	Match one or more word characters.
`(\s\w+)*`	Match zero or more occurrences of a white-space character followed by one or more word characters. This pattern matches multi-word city names. This is the third capturing group.

PATTERN	DESCRIPTION
`(\w+(\s\w+)*)`	Match one or more word characters followed by zero or more occurrences of a white-space character and one or more word characters. This is the second capturing group.
`,`	Match a comma.
`(\d+)`	Match one or more digits. This is the fourth capturing group.
`;`	Match a semicolon.
`((\w+(\s\w+)*),(\d+);)+`	Match the pattern of a word followed by any additional words followed by a comma, one or more digits, and a semicolon, one or more times. This is the first capturing group.

See also

- System.Text.RegularExpressions
- .NET Regular Expressions
- Regular Expression Language - Quick Reference

Details of Regular Expression Behavior

5/2/2018 • 18 minutes to read • Edit Online

The .NET Framework regular expression engine is a backtracking regular expression matcher that incorporates a traditional Nondeterministic Finite Automaton (NFA) engine such as that used by Perl, Python, Emacs, and Tcl. This distinguishes it from faster, but more limited, pure regular expression Deterministic Finite Automaton (DFA) engines such as those found in awk, egrep, or lex. This also distinguishes it from standardized, but slower, POSIX NFAs. The following section describes the three types of regular expression engines, and explains why regular expressions in the .NET Framework are implemented by using a traditional NFA engine.

Benefits of the NFA Engine

When DFA engines perform pattern matching, their processing order is driven by the input string. The engine begins at the beginning of the input string and proceeds sequentially to determine whether the next character matches the regular expression pattern. They can guarantee to match the longest string possible. Because they never test the same character twice, DFA engines do not support backtracking. However, because a DFA engine contains only finite state, it cannot match a pattern with backreferences, and because it does not construct an explicit expansion, it cannot capture subexpressions.

Unlike DFA engines, when traditional NFA engines perform pattern matching, their processing order is driven by the regular expression pattern. As it processes a particular language element, the engine uses greedy matching; that is, it matches as much of the input string as it possibly can. But it also saves its state after successfully matching a subexpression. If a match eventually fails, the engine can return to a saved state so it can try additional matches. This process of abandoning a successful subexpression match so that later language elements in the regular expression can also match is known as *backtracking*. NFA engines use backtracking to test all possible expansions of a regular expression in a specific order and accept the first match. Because a traditional NFA engine constructs a specific expansion of the regular expression for a successful match, it can capture subexpression matches and matching backreferences. However, because a traditional NFA backtracks, it can visit the same state multiple times if it arrives at the state over different paths. As a result, it can run exponentially slowly in the worst case. Because a traditional NFA engine accepts the first match it finds, it can also leave other (possibly longer) matches undiscovered.

POSIX NFA engines are like traditional NFA engines, except that they continue to backtrack until they can guarantee that they have found the longest match possible. As a result, a POSIX NFA engine is slower than a traditional NFA engine, and when you use a POSIX NFA engine, you cannot favor a shorter match over a longer one by changing the order of the backtracking search.

Traditional NFA engines are favored by programmers because they offer greater control over string matching than either DFA or POSIX NFA engines. Although, in the worst case, they can run slowly, you can steer them to find matches in linear or polynomial time by using patterns that reduce ambiguities and limit backtracking. In other words, although NFA engines trade performance for power and flexibility, in most cases they offer good to acceptable performance if a regular expression is well-written and avoids cases in which backtracking degrades performance exponentially.

> **NOTE**
>
> For information about the performance penalty caused by excessive backtracking and ways to craft a regular expression to work around them, see Backtracking.

.NET Framework Engine Capabilities

To take advantage of the benefits of a traditional NFA engine, the .NET Framework regular expression engine includes a complete set of constructs to enable programmers to steer the backtracking engine. These constructs can be used to find matches faster or to favor specific expansions over others.

Other features of the .NET Framework regular expression engine include the following:

- Lazy quantifiers: `??`, `*?`, `+?`, `{` *n* `,` *m* `}?`. These constructs tell the backtracking engine to search the minimum number of repetitions first. In contrast, ordinary greedy quantifiers try to match the maximum number of repetitions first. The following example illustrates the difference between the two. A regular expression matches a sentence that ends in a number, and a capturing group is intended to extract that number. The regular expression `.+(\d+)\.` includes the greedy quantifier `.+`, which causes the regular expression engine to capture only the last digit of the number. In contrast, the regular expression `.+?(\d+)\.` includes the lazy quantifier `.+?`, which causes the regular expression engine to capture the entire number.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string greedyPattern = @".+(\d+)\.";
      string lazyPattern = @".+?(\d+)\.";
      string input = "This sentence ends with the number 107325.";
      Match match;

      // Match using greedy quantifier .+.
      match = Regex.Match(input, greedyPattern);
      if (match.Success)
         Console.WriteLine("Number at end of sentence (greedy): {0}",
                           match.Groups[1].Value);
      else
         Console.WriteLine("{0} finds no match.", greedyPattern);

      // Match using lazy quantifier .+?.
      match = Regex.Match(input, lazyPattern);
      if (match.Success)
         Console.WriteLine("Number at end of sentence (lazy): {0}",
                           match.Groups[1].Value);
      else
         Console.WriteLine("{0} finds no match.", lazyPattern);
   }
}
// The example displays the following output:
//       Number at end of sentence (greedy): 5
//       Number at end of sentence (lazy): 107325
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim greedyPattern As String = ".+(\d+)\."
      Dim lazyPattern As String = ".+?(\d+)\."
      Dim input As String = "This sentence ends with the number 107325."
      Dim match As Match

      ' Match using greedy quantifier .+.
      match = Regex.Match(input, greedyPattern)
      If match.Success Then
         Console.WriteLine("Number at end of sentence (greedy): {0}",
                           match.Groups(1).Value)
      Else
         Console.WriteLine("{0} finds no match.", greedyPattern)
      End If

      ' Match using lazy quantifier .+?.
      match = Regex.Match(input, lazyPattern)
      If match.Success Then
         Console.WriteLine("Number at end of sentence (lazy): {0}",
                           match.Groups(1).Value)
      Else
         Console.WriteLine("{0} finds no match.", lazyPattern)
      End If
   End Sub
End Module
' The example displays the following output:
'       Number at end of sentence (greedy): 5
'       Number at end of sentence (lazy): 107325
```

The greedy and lazy versions of this regular expression are defined as shown in the following table.`

PATTERN	DESCRIPTION
`.+` (greedy quantifier)	Match at least one occurrence of any character. This causes the regular expression engine to match the entire string, and then to backtrack as needed to match the remainder of the pattern.
`.+?` (lazy quantifier)	Match at least one occurrence of any character, but match as few as possible.
`(\d+)`	Match at least one numeric character, and assign it to the first capturing group.
`\.`	Match a period.

For more information about lazy quantifiers, see Quantifiers.

- Positive lookahead: `(?=` *subexpression* `)`. This feature allows the backtracking engine to return to the same spot in the text after matching a subexpression. It is useful for searching throughout the text by verifying multiple patterns that start from the same position. It also allows the engine to verify that a substring exists at the end of the match without including the substring in the matched text. The following example uses positive lookahead to extract the words in a sentence that are not followed by punctuation symbols.

```csharp
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b[A-Z]+\b(?=\P{P})";
      string input = "If so, what comes next?";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       If
//       what
//       comes
```

```vb
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b[A-Z]+\b(?=\P{P})"
      Dim input As String = "If so, what comes next?"
      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
         Console.WriteLine(match.Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       If
'       what
'       comes
```

The regular expression `\b[A-Z]+\b(?=\P{P})` is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`[A-Z]+`	Match any alphabetic character one or more times. Because the Regex.Matches method is called with the RegexOptions.IgnoreCase option, the comparison is case-insensitive.
`\b`	End the match at a word boundary.
`(?=\P{P})`	Look ahead to determine whether the next character is a punctuation symbol. If it is not, the match succeeds.

For more information about positive lookahead assertions, see Grouping Constructs.

- Negative lookahead: `(?!` *subexpression* `)`. This feature adds the ability to match an expression only if a subexpression fails to match. This is particularly powerful for pruning a search, because it is often simpler to provide an expression for a case that should be eliminated than an expression for cases that must be included. For example, it is difficult to write an expression for words that do not begin with "non". The following example uses negative lookahead to exclude them.

```csharp
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?!non)\w+\b";
      string input = "Nonsense is not always non-functional.";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       is
//       not
//       always
//       functional
```

```vb
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "\b(?!non)\w+\b"
      Dim input As String = "Nonsense is not always non-functional."
      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
         Console.WriteLine(match.Value)
      Next
   End Sub
End Module
' The example displays the following output:
'       is
'       not
'       always
'       functional
```

The regular expression pattern `\b(?!non)\w+\b` is defined as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`(?!non)`	Look ahead to ensure that the current string does not begin with "non". If it does, the match fails.
`(\w+)`	Match one or more word characters.
`\b`	End the match at a word boundary.

For more information about negative lookahead assertions, see Grouping Constructs.

- Conditional evaluation: `(?(` *expression* `)` *yes* `|` *no* `)` and `(?(` *name* `)` *yes* `|` *no* `)`, where *expression* is a subexpression to match, *name* is the name of a capturing group, *yes* is the string to match if *expression* is matched or *name* is a valid, non-empty captured group, and *no* is the subexpression to match if *expression* is not matched or *name* is not a valid, non-empty captured group. This feature allows the engine to search by using more than one alternate pattern, depending on the result of a previous subexpression match or the result of a zero-width assertion. This allows a more powerful form of backreference that permits, for example, matching a subexpression based on whether a previous subexpression was matched. The regular expression in the following example matches paragraphs that are intended for both public and internal use.

Paragraphs intended only for internal use begin with a `<PRIVATE>` tag. The regular expression pattern `^(?<Pvt>\<PRIVATE\>\s)?(?(Pvt)((\w+\p{P}?\s)+)|((\w+\p{P}?\s)+))\r?$` uses conditional evaluation to assign the contents of paragraphs intended for public and for internal use to separate capturing groups. These paragraphs can then be handled differently.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "<PRIVATE> This is not for public consumption." + Environment.NewLine +
                     "But this is for public consumption." + Environment.NewLine +
                     "<PRIVATE> Again, this is confidential.\n";
      string pattern = @"^(?<Pvt>\<PRIVATE\>\s)?(?(Pvt)((\w+\p{P}?\s)+)|((\w+\p{P}?\s)+))\r?$";
      string publicDocument = null, privateDocument = null;

      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Multiline))
      {
         if (match.Groups[1].Success) {
            privateDocument += match.Groups[1].Value + "\n";
         }
         else {
            publicDocument += match.Groups[3].Value + "\n";
            privateDocument += match.Groups[3].Value + "\n";
         }
      }

      Console.WriteLine("Private Document:");
      Console.WriteLine(privateDocument);
      Console.WriteLine("Public Document:");
      Console.WriteLine(publicDocument);
   }
}
// The example displays the following output:
//    Private Document:
//    This is not for public consumption.
//    But this is for public consumption.
//    Again, this is confidential.
//
//    Public Document:
//    But this is for public consumption.
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "<PRIVATE> This is not for public consumption." + vbCrLf + _
                            "But this is for public consumption." + vbCrLf + _
                            "<PRIVATE> Again, this is confidential." + vbCrLf
      Dim pattern As String = "^(?<Pvt>\<PRIVATE\>\s)?(?(Pvt)((\w+\p{P}?\s)+)|((\w+\p{P}?\s)+))\r?$"
      Dim publicDocument As String = Nothing
      Dim privateDocument As String = Nothing

      For Each match As Match In Regex.Matches(input, pattern, RegexOptions.Multiline)
         If match.Groups(1).Success Then
            privateDocument += match.Groups(1).Value + vbCrLf
         Else
            publicDocument += match.Groups(3).Value + vbCrLf
            privateDocument += match.Groups(3).Value + vbCrLf
         End If
      Next

      Console.WriteLine("Private Document:")
      Console.WriteLine(privateDocument)
      Console.WriteLine("Public Document:")
      Console.WriteLine(publicDocument)
   End Sub
End Module
' The example displays the following output:
'    Private Document:
'    This is not for public consumption.
'    But this is for public consumption.
'    Again, this is confidential.
'
'    Public Document:
'    But this is for public consumption.
```

The regular expression pattern is defined as shown in the following table.

PATTERN	DESCRIPTION
`^`	Begin the match at the beginning of a line.
`(?<Pvt>\<PRIVATE\>\s)?`	Match zero or one occurrence of the string `<PRIVATE>` followed by a white-space character. Assign the match to a capturing group named `Pvt`.
`(?(Pvt)((\w+\p{P}?\s)+)`	If the `Pvt` capturing group exists, match one or more occurrences of one or more word characters followed by zero or one punctuation separator followed by a white-space character. Assign the substring to the first capturing group.
`|((\w+\p{P}?\s)+))`	If the `Pvt` capturing group does not exist, match one or more occurrences of one or more word characters followed by zero or one punctuation separator followed by a white-space character. Assign the substring to the third capturing group.
`\r?$`	Match the end of a line or the end of the string.

For more information about conditional evaluation, see Alternation Constructs.

- Balancing group definitions: `(?<` *name1* `-` *name2* `>` *subexpression* `)`. This feature allows the regular expression engine to keep track of nested constructs such as parentheses or opening and closing brackets. For an example, see Grouping Constructs.

- Nonbacktracking subexpressions (also known as greedy subexpressions): `(?>` *subexpression* `)`. This feature allows the backtracking engine to guarantee that a subexpression matches only the first match found for that subexpression, as if the expression were running independent of its containing expression. If you do not use this construct, backtracking searches from the larger expression can change the behavior of a subexpression. For example, the regular expression `(a+)\w` matches one or more "a" characters, along with a word character that follows the sequence of "a" characters, and assigns the sequence of "a" characters to the first capturing group, However, if the final character of the input string is also an "a", it is matched by the `\w` language element and is not included in the captured group.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "aaaaa", "aaaaab" };
      string backtrackingPattern = @"(a+)\w";
      Match match;

      foreach (string input in inputs) {
         Console.WriteLine("Input: {0}", input);
         match = Regex.Match(input, backtrackingPattern);
         Console.WriteLine("   Pattern: {0}", backtrackingPattern);
         if (match.Success) {
            Console.WriteLine("      Match: {0}", match.Value);
            Console.WriteLine("      Group 1: {0}", match.Groups[1].Value);
         }
         else {
            Console.WriteLine("      Match failed.");
         }
      }
      Console.WriteLine();
   }
}
// The example displays the following output:
//       Input: aaaaa
//          Pattern: (a+)\w
//             Match: aaaaa
//             Group 1: aaaa
//       Input: aaaaab
//          Pattern: (a+)\w
//             Match: aaaaab
//             Group 1: aaaaa
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim inputs() As String = { "aaaaa", "aaaaab" }
      Dim backtrackingPattern As String = "(a+)\w"
      Dim match As Match

      For Each input As String In inputs
         Console.WriteLine("Input: {0}", input)
         match = Regex.Match(input, backtrackingPattern)
         Console.WriteLine("   Pattern: {0}", backtrackingPattern)
         If match.Success Then
            Console.WriteLine("      Match: {0}", match.Value)
            Console.WriteLine("      Group 1: {0}", match.Groups(1).Value)
         Else
            Console.WriteLine("      Match failed.")
         End If
      Next
      Console.WriteLine()
   End Sub
End Module
' The example displays the following output:
'       Input: aaaaa
'          Pattern: (a+)\w
'             Match: aaaaa
'             Group 1: aaaa
'       Input: aaaaab
'          Pattern: (a+)\w
'             Match: aaaaab
'             Group 1: aaaaa
```

The regular expression `((?>a+))\w` prevents this behavior. Because all consecutive "a" characters are matched without backtracking, the first capturing group includes all consecutive "a" characters. If the "a" characters are not followed by at least one more character other than "a", the match fails.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "aaaaa", "aaaaab" };
      string nonbacktrackingPattern = @"((?>a+))\w";
      Match match;

      foreach (string input in inputs) {
         Console.WriteLine("Input: {0}", input);
         match = Regex.Match(input, nonbacktrackingPattern);
         Console.WriteLine("   Pattern: {0}", nonbacktrackingPattern);
         if (match.Success) {
            Console.WriteLine("      Match: {0}", match.Value);
            Console.WriteLine("      Group 1: {0}", match.Groups[1].Value);
         }
         else {
            Console.WriteLine("      Match failed.");
         }
      }
      Console.WriteLine();
   }
}
// The example displays the following output:
//       Input: aaaaa
//          Pattern: ((?>a+))\w
//             Match failed.
//       Input: aaaaab
//          Pattern: ((?>a+))\w
//             Match: aaaaab
//             Group 1: aaaaa
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim inputs() As String = { "aaaaa", "aaaaab" }
      Dim nonbacktrackingPattern As String = "((?>a+))\w"
      Dim match As Match

      For Each input As String In inputs
         Console.WriteLine("Input: {0}", input)
         match = Regex.Match(input, nonbacktrackingPattern)
         Console.WriteLine("   Pattern: {0}", nonbacktrackingPattern)
         If match.Success Then
            Console.WriteLine("      Match: {0}", match.Value)
            Console.WriteLine("      Group 1: {0}", match.Groups(1).Value)
         Else
            Console.WriteLine("      Match failed.")
         End If
      Next
      Console.WriteLine()
   End Sub
End Module
' The example displays the following output:
'       Input: aaaaa
'          Pattern: ((?>a+))\w
'             Match failed.
'       Input: aaaaab
'          Pattern: ((?>a+))\w
'             Match: aaaaab
'             Group 1: aaaaa
```

For more information about nonbacktracking subexpressions, see Grouping Constructs.

- Right-to-left matching, which is specified by supplying the RegexOptions.RightToLeft option to a Regex class constructor or static instance matching method. This feature is useful when searching from right to left instead of from left to right, or in cases where it is more efficient to begin a match at the right part of the pattern instead of the left. As the following example illustrates, using right-to-left matching can change the behavior of greedy quantifiers. The example conducts two searches for a sentence that ends in a number. The left-to-right search that uses the greedy quantifier `+` matches one of the six digits in the sentence, whereas the right-to-left search matches all six digits. For an description of the regular expression pattern, see the example that illustrates lazy quantifiers earlier in this section.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string greedyPattern = @".+(\d+)\.";
      string input = "This sentence ends with the number 107325.";
      Match match;

      // Match from left-to-right using lazy quantifier .+?.
      match = Regex.Match(input, greedyPattern);
      if (match.Success)
         Console.WriteLine("Number at end of sentence (left-to-right): {0}",
                           match.Groups[1].Value);
      else
         Console.WriteLine("{0} finds no match.", greedyPattern);

      // Match from right-to-left using greedy quantifier .+.
      match = Regex.Match(input, greedyPattern, RegexOptions.RightToLeft);
      if (match.Success)
         Console.WriteLine("Number at end of sentence (right-to-left): {0}",
                           match.Groups[1].Value);
      else
         Console.WriteLine("{0} finds no match.", greedyPattern);
   }
}
// The example displays the following output:
//       Number at end of sentence (left-to-right): 5
//       Number at end of sentence (right-to-left): 107325
```

```vb
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim greedyPattern As String = ".+(\d+)\."
      Dim input As String = "This sentence ends with the number 107325."
      Dim match As Match

      ' Match from left-to-right using lazy quantifier .+?.
      match = Regex.Match(input, greedyPattern)
      If match.Success Then
         Console.WriteLine("Number at end of sentence (left-to-right): {0}",
                           match.Groups(1).Value)
      Else
         Console.WriteLine("{0} finds no match.", greedyPattern)
      End If

      ' Match from right-to-left using greedy quantifier .+.
      match = Regex.Match(input, greedyPattern, RegexOptions.RightToLeft)
      If match.Success Then
         Console.WriteLine("Number at end of sentence (right-to-left): {0}",
                           match.Groups(1).Value)
      Else
         Console.WriteLine("{0} finds no match.", greedyPattern)
      End If
   End Sub
End Module
' The example displays the following output:
'       Number at end of sentence (left-to-right): 5
'       Number at end of sentence (right-to-left): 107325
```

For more information about right-to-left matching, see Regular Expression Options.

- Positive and negative lookbehind: `(?<=` *subexpression* `)` for positive lookbehind, and `(?<!` *subexpression* `)` for negative lookbehind. This feature is similar to lookahead, which is discussed earlier in this topic. Because the regular expression engine allows complete right-to-left matching, regular expressions allow unrestricted lookbehinds. Positive and negative lookbehind can also be used to avoid nesting quantifiers when the nested subexpression is a superset of an outer expression. Regular expressions with such nested quantifiers often offer poor performance. For example, the following example verifies that a string begins and ends with an alphanumeric character, and that any other character in the string is one of a larger subset. It forms a portion of the regular expression used to validate email addresses; for more information, see How to: Verify that Strings Are in Valid Email Format.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "jack.sprat", "dog#", "dog#1", "me.myself",
                          "me.myself!" };
      string pattern = @"^[A-Z0-9]([-!#$%&'.*+/=?^`{}|~\w])*(?<=[A-Z0-9])$";
      foreach (string input in inputs) {
         if (Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase))
            Console.WriteLine("{0}: Valid", input);
         else
            Console.WriteLine("{0}: Invalid", input);
      }
   }
}
// The example displays the following output:
//       jack.sprat: Valid
//       dog#: Invalid
//       dog#1: Valid
//       me.myself: Valid
//       me.myself!: Invalid
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim inputs() As String = { "jack.sprat", "dog#", "dog#1", "me.myself",
                                 "me.myself!" }
      Dim pattern As String = "^[A-Z0-9]([-!#$%&'.*+/=?^`{}|~\w])*(?<=[A-Z0-9])$"
      For Each input As String In inputs
         If Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase) Then
            Console.WriteLine("{0}: Valid", input)
         Else
            Console.WriteLine("{0}: Invalid", input)
         End If
      Next
   End Sub
End Module
' The example displays the following output:
'       jack.sprat: Valid
'       dog#: Invalid
'       dog#1: Valid
'       me.myself: Valid
'       me.myself!: Invalid
```

The regular expression `^[A-Z0-9]([-!#$%&'.*+/=?^ `{}|~\w])*(?<=[A-Z0-9])$` is defined as shown in the following table.

<table>
<tr><th>PATTERN</th><th>DESCRIPTION</th></tr>
<tr><td><code>^</code></td><td>Begin the match at the beginning of the string.</td></tr>
<tr><td><code>[A-Z0-9]</code></td><td>Match any numeric or alphanumeric character. (The comparison is case-insensitive.)</td></tr>
<tr><td><code>([-!#$%&'.+/=?^`{}|~\w])</code></td><td>Match zero or more occurrences of any word character, or any of the following characters: -, !, #, $, %, &, ', ., *, +, /, =, ?, ^, `, {, }, |, or ~.</td></tr>
</table>

PATTERN	DESCRIPTION
`(?<=[A-Z0-9])`	Look behind to the previous character, which must be numeric or alphanumeric. (The comparison is case-insensitive.)
`$`	End the match at the end of the string.

For more information about positive and negative lookbehind, see Grouping Constructs.

Related Topics

TITLE	DESCRIPTION
Backtracking	Provides information about how regular expression backtracking branches to find alternative matches.
Compilation and Reuse	Provides information about compiling and reusing regular expressions to increase performance.
Thread Safety	Provides information about regular expression thread safety and explains when you should synchronize access to regular expression objects.
.NET Framework Regular Expressions	Provides an overview of the programming language aspect of regular expressions.
The Regular Expression Object Model	Provides information and code examples illustrating how to use the regular expression classes.
Regular Expression Examples	Contains code examples that illustrate the use of regular expressions in common applications.
Regular Expression Language - Quick Reference	Provides information about the set of characters, operators, and constructs that you can use to define regular expressions.

Reference

System.Text.RegularExpressions

Backtracking in Regular Expressions

9/6/2018 • 21 minutes to read • Edit Online

Backtracking occurs when a regular expression pattern contains optional quantifiers or alternation constructs, and the regular expression engine returns to a previous saved state to continue its search for a match. Backtracking is central to the power of regular expressions; it makes it possible for expressions to be powerful and flexible, and to match very complex patterns. At the same time, this power comes at a cost. Backtracking is often the single most important factor that affects the performance of the regular expression engine. Fortunately, the developer has control over the behavior of the regular expression engine and how it uses backtracking. This topic explains how backtracking works and how it can be controlled.

> **NOTE**
>
> In general, a Nondeterministic Finite Automaton (NFA) engine like .NET regular expression engine places the responsibility for crafting efficient, fast regular expressions on the developer.

This topic contains the following sections:

Linear Comparison Without Backtracking

If a regular expression pattern has no optional quantifiers or alternation constructs, the regular expression engine executes in linear time. That is, after the regular expression engine matches the first language element in the pattern with text in the input string, it tries to match the next language element in the pattern with the next character or group of characters in the input string. This continues until the match either succeeds or fails. In either case, the regular expression engine advances by one character at a time in the input string.

The following example provides an illustration. The regular expression `e{2}\w\b` looks for two occurrences of the letter "e" followed by any word character followed by a word boundary.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "needing a reed";
      string pattern = @"e{2}\w\b";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("{0} found at position {1}",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       eed found at position 11
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "needing a reed"
      Dim pattern As String = "e{2}\w\b"
      For Each match As Match In Regex.Matches(input, pattern)
         Console.WriteLine("{0} found at position {1}", _
                           match.Value, match.Index)
      Next
   End Sub
End Module
' The example displays the following output:
'       eed found at position 11
```

Although this regular expression includes the quantifier `{2}`, it is evaluated in a linear manner. The regular expression engine does not backtrack because `{2}` is not an optional quantifier; it specifies an exact number and not a variable number of times that the previous subexpression must match. As a result, the regular expression engine tries to match the regular expression pattern with the input string as shown in the following table.

OPERATION	POSITION IN PATTERN	POSITION IN STRING	RESULT
1	e	"needing a reed" (index 0)	No match.
2	e	"eeding a reed" (index 1)	Possible match.
3	e{2}	"eding a reed" (index 2)	Possible match.
4	\w	"ding a reed" (index 3)	Possible match.
5	\b	"ing a reed" (index 4)	Possible match fails.
6	e	"eding a reed" (index 2)	Possible match.
7	e{2}	"ding a reed" (index 3)	Possible match fails.
8	e	"ding a reed" (index 3)	Match fails.
9	e	"ing a reed" (index 4)	No match.
10	e	"ng a reed" (index 5)	No match.
11	e	"g a reed" (index 6)	No match.
12	e	" a reed" (index 7)	No match.
13	e	"a reed" (index 8)	No match.
14	e	" reed" (index 9)	No match.
15	e	"reed" (index 10)	No match
16	e	"eed" (index 11)	Possible match.

OPERATION	POSITION IN PATTERN	POSITION IN STRING	RESULT
17	e{2}	"ed" (index 12)	Possible match.
18	\w	"d" (index 13)	Possible match.
19	\b	"" (index 14)	Match.

If a regular expression pattern includes no optional quantifiers or alternation constructs, the maximum number of comparisons required to match the regular expression pattern with the input string is roughly equivalent to the number of characters in the input string. In this case, the regular expression engine uses 19 comparisons to identify possible matches in this 13-character string. In other words, the regular expression engine runs in near-linear time if it contains no optional quantifiers or alternation constructs.

Back to top

Backtracking with Optional Quantifiers or Alternation Constructs

When a regular expression includes optional quantifiers or alternation constructs, the evaluation of the input string is no longer linear. Pattern matching with an NFA engine is driven by the language elements in the regular expression and not by the characters to be matched in the input string. Therefore, the regular expression engine tries to fully match optional or alternative subexpressions. When it advances to the next language element in the subexpression and the match is unsuccessful, the regular expression engine can abandon a portion of its successful match and return to an earlier saved state in the interest of matching the regular expression as a whole with the input string. This process of returning to a previous saved state to find a match is known as backtracking.

For example, consider the regular expression pattern `.*(es)`, which matches the characters "es" and all the characters that precede it. As the following example shows, if the input string is "Essential services are provided by regular expressions.", the pattern matches the whole string up to and including the "es" in "expressions".

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "Essential services are provided by regular expressions.";
      string pattern = ".*(es)";
      Match m = Regex.Match(input, pattern, RegexOptions.IgnoreCase);
      if (m.Success) {
         Console.WriteLine("'{0}' found at position {1}",
                           m.Value, m.Index);
         Console.WriteLine("'es' found at position {0}",
                           m.Groups[1].Index);
      }
   }
}
//    'Essential services are provided by regular expres' found at position 0
//    'es' found at position 47
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "Essential services are provided by regular expressions."
      Dim pattern As String = ".*(es)"
      Dim m As Match = Regex.Match(input, pattern, RegexOptions.IgnoreCase)
      If m.Success Then
         Console.WriteLine("'{0}' found at position {1}", _
                           m.Value, m.Index)
         Console.WriteLine("'es' found at position {0}", _
                           m.Groups(1).Index)
      End If
   End Sub
End Module
'    'Essential services are provided by regular expres' found at position 0
'    'es' found at position 47
```

To do this, the regular expression engine uses backtracking as follows:

- It matches the `.*` (which matches zero, one, or more occurrences of any character) with the whole input string.
- It attempts to match "e" in the regular expression pattern. However, the input string has no remaining characters available to match.
- It backtracks to its last successful match, "Essential services are provided by regular expressions", and attempts to match "e" with the period at the end of the sentence. The match fails.
- It continues to backtrack to a previous successful match one character at a time until the tentatively matched substring is "Essential services are provided by regular expr". It then compares the "e" in the pattern to the second "e" in "expressions" and finds a match.
- It compares "s" in the pattern to the "s" that follows the matched "e" character (the first "s" in "expressions"). The match is successful.

When you use backtracking, matching the regular expression pattern with the input string, which is 55 characters long, requires 67 comparison operations. Interestingly, if the regular expression pattern included a lazy quantifier, `.*?(es)`, matching the regular expression would require additional comparisons. In this case, instead of having to backtrack from the end of the string to the "r" in "expressions", the regular expression engine would have to backtrack all the way to the beginning of the string to match "Es" and would require 113 comparisons. Generally, if a regular expression pattern has a single alternation construct or a single optional quantifier, the number of comparison operations required to match the pattern is more than twice the number of characters in the input string.

Back to top

Backtracking with Nested Optional Quantifiers

The number of comparison operations required to match a regular expression pattern can increase exponentially if the pattern includes a large number of alternation constructs, if it includes nested alternation constructs, or, most commonly, if it includes nested optional quantifiers. For example, the regular expression pattern `^(a+)+$` is designed to match a complete string that contains one or more "a" characters. The example provides two input strings of identical length, but only the first string matches the pattern. The System.Diagnostics.Stopwatch class is used to determine how long the match operation takes.

```csharp
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "^(a+)+$";
      string[] inputs = { "aaaaaa", "aaaaa!" };
      Regex rgx = new Regex(pattern);
      Stopwatch sw;

      foreach (string input in inputs) {
         sw = Stopwatch.StartNew();
         Match match = rgx.Match(input);
         sw.Stop();
         if (match.Success)
            Console.WriteLine("Matched {0} in {1}", match.Value, sw.Elapsed);
         else
            Console.WriteLine("No match found in {0}", sw.Elapsed);
      }
   }
}
```

```vb
Imports System.Diagnostics
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim pattern As String = "^(a+)+$"
      Dim inputs() As String = { "aaaaaa", "aaaaa!" }
      Dim rgx As New Regex(pattern)
      Dim sw As Stopwatch

      For Each input As String In inputs
         sw = Stopwatch.StartNew()
         Dim match As Match = rgx.Match(input)
         sw.Stop()
         If match.Success Then
            Console.WriteLine("Matched {0} in {1}", match.Value, sw.Elapsed)
         Else
            Console.WriteLine("No match found in {0}", sw.Elapsed)
         End If
      Next
   End Sub
End Module
```

As the output from the example shows, the regular expression engine took about twice as long to find that an input string did not match the pattern as it did to identify a matching string. This is because an unsuccessful match always represents a worst-case scenario. The regular expression engine must use the regular expression to follow all possible paths through the data before it can conclude that the match is unsuccessful, and the nested parentheses create many additional paths through the data. The regular expression engine concludes that the second string did not match the pattern by doing the following:

- It checks that it was at the beginning of the string, and then matches the first five characters in the string with the pattern `a+`. It then determines that there are no additional groups of "a" characters in the string. Finally, it tests for the end of the string. Because one additional character remains in the string, the match fails. This failed match requires 9 comparisons. The regular expression engine also saves state information from its matches of "a" (which we will call match 1), "aa" (match 2), "aaa" (match 3), and "aaaa" (match 4).
- It returns to the previously saved match 4. It determines that there is one additional "a" character to assign

to an additional captured group. Finally, it tests for the end of the string. Because one additional character remains in the string, the match fails. This failed match requires 4 comparisons. So far, a total of 13 comparisons have been performed.

- It returns to the previously saved match 3. It determines that there are two additional "a" characters to assign to an additional captured group. However, the end-of-string test fails. It then returns to match3 and tries to match the two additional "a" characters in two additional captured groups. The end-of-string test still fails. These failed matches require 12 comparisons. So far, a total of 25 comparisons have been performed.

Comparison of the input string with the regular expression continues in this way until the regular expression engine has tried all possible combinations of matches, and then concludes that there is no match. Because of the nested quantifiers, this comparison is an $O(2^n)$ or an exponential operation, where *n* is the number of characters in the input string. This means that in the worst case, an input string of 30 characters requires approximately 1,073,741,824 comparisons, and an input string of 40 characters requires approximately 1,099,511,627,776 comparisons. If you use strings of these or even greater lengths, regular expression methods can take an extremely long time to complete when they process input that does not match the regular expression pattern.

Back to top

Controlling Backtracking

Backtracking lets you create powerful, flexible regular expressions. However, as the previous section showed, these benefits may be coupled with unacceptably poor performance. To prevent excessive backtracking, you should define a time-out interval when you instantiate a Regex object or call a static regular expression matching method. This is discussed in the next section. In addition, .NET supports three regular expression language elements that limit or suppress backtracking and that support complex regular expressions with little or no performance penalty: nonbacktracking subexpressions, lookbehind assertions, and lookahead assertions. For more information about each language element, see Grouping Constructs.

Defining a Time-out Interval

Starting with the .NET Framework 4.5, you can set a time-out value that represents the longest interval the regular expression engine will search for a single match before it abandons the attempt and throws a RegexMatchTimeoutException exception. You specify the time-out interval by supplying a TimeSpan value to the Regex.Regex(String, RegexOptions, TimeSpan) constructor for instance regular expressions. In addition, each static pattern matching method has an overload with a TimeSpan parameter that allows you to specify a time-out value. By default, the time-out interval is set to Regex.InfiniteMatchTimeout and the regular expression engine does not time out.

> **IMPORTANT**
>
> We recommend that you always set a time-out interval if your regular expression relies on backtracking.

A RegexMatchTimeoutException exception indicates that the regular expression engine was unable to find a match within in the specified time-out interval but does not indicate why the exception was thrown. The reason might be excessive backtracking, but it is also possible that the time-out interval was set too low given the system load at the time the exception was thrown. When you handle the exception, you can choose to abandon further matches with the input string or increase the time-out interval and retry the matching operation.

For example, the following code calls the Regex.Regex(String, RegexOptions, TimeSpan) constructor to instantiate a Regex object with a time-out value of one second. The regular expression pattern `(a+)+$`, which matches one or more sequences of one or more "a" characters at the end of a line, is subject to excessive backtracking. If a RegexMatchTimeoutException is thrown, the example increases the time-out value up to a maximum interval of three seconds. After that, it abandons the attempt to match the pattern.

```
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.Security;
using System.Text.RegularExpressions;
using System.Threading;

public class Example
{
   const int MaxTimeoutInSeconds = 3;

   public static void Main()
   {
      string pattern = @"(a+)+$";    // DO NOT REUSE THIS PATTERN.
      Regex rgx = new Regex(pattern, RegexOptions.IgnoreCase, TimeSpan.FromSeconds(1));
      Stopwatch sw = null;

      string[] inputs= { "aa", "aaaa>",
                         "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                         "aaaaaaaaaaaaaaaaaaaaaa>",
                         "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>" };

      foreach (var inputValue in inputs) {
         Console.WriteLine("Processing {0}", inputValue);
         bool timedOut = false;
         do {
            try {
               sw = Stopwatch.StartNew();
               // Display the result.
               if (rgx.IsMatch(inputValue)) {
                  sw.Stop();
                  Console.WriteLine(@"Valid: '{0}' ({1:ss\.fffffff} seconds)",
                                    inputValue, sw.Elapsed);
               }
               else {
                  sw.Stop();
                  Console.WriteLine(@"'{0}' is not a valid string. ({1:ss\.fffff} seconds)",
                                    inputValue, sw.Elapsed);
               }
            }
            catch (RegexMatchTimeoutException e) {
               sw.Stop();
               // Display the elapsed time until the exception.
               Console.WriteLine(@"Timeout with '{0}' after {1:ss\.fffff}",
                                 inputValue, sw.Elapsed);
               Thread.Sleep(1500);       // Pause for 1.5 seconds.

               // Increase the timeout interval and retry.
               TimeSpan timeout = e.MatchTimeout.Add(TimeSpan.FromSeconds(1));
               if (timeout.TotalSeconds > MaxTimeoutInSeconds) {
                  Console.WriteLine("Maximum timeout interval of {0} seconds exceeded.",
                                    MaxTimeoutInSeconds);
                  timedOut = false;
               }
               else {
                  Console.WriteLine("Changing the timeout interval to {0}",
                                    timeout);
                  rgx = new Regex(pattern, RegexOptions.IgnoreCase, timeout);
                  timedOut = true;
               }
            }
         } while (timedOut);
         Console.WriteLine();
      }
   }
}
// The example displays output like the following :
//    Processing aa
```

```
//    Valid: 'aa' (00.0000779 seconds)
//
//    Processing aaaa>
//    'aaaa>' is not a valid string. (00.00005 seconds)
//
//    Processing aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
//    Valid: 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' (00.0000043 seconds)
//
//    Processing aaaaaaaaaaaaaaaaaaaaaa>
//    Timeout with 'aaaaaaaaaaaaaaaaaaaaaa>' after 01.00469
//    Changing the timeout interval to 00:00:02
//    Timeout with 'aaaaaaaaaaaaaaaaaaaaaa>' after 02.01202
//    Changing the timeout interval to 00:00:03
//    Timeout with 'aaaaaaaaaaaaaaaaaaaaaa>' after 03.01043
//    Maximum timeout interval of 3 seconds exceeded.
//
//    Processing aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>
//    Timeout with 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>' after 03.01018
//    Maximum timeout interval of 3 seconds exceeded.
```

```
Imports System.ComponentModel
Imports System.Diagnostics
Imports System.Security
Imports System.Text.RegularExpressions
Imports System.Threading

Module Example
   Const MaxTimeoutInSeconds As Integer = 3

   Public Sub Main()
      Dim pattern As String = "(a+)+$"    ' DO NOT REUSE THIS PATTERN.
      Dim rgx As New Regex(pattern, RegexOptions.IgnoreCase, TimeSpan.FromSeconds(1))
      Dim sw As Stopwatch = Nothing

      Dim inputs() As String = { "aa", "aaaa>",
                                 "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                                 "aaaaaaaaaaaaaaaaaaaaaa>",
                                 "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>" }

      For Each inputValue In inputs
         Console.WriteLine("Processing {0}", inputValue)
         Dim timedOut As Boolean = False
         Do
            Try
               sw = Stopwatch.StartNew()
               ' Display the result.
               If rgx.IsMatch(inputValue) Then
                  sw.Stop()
                  Console.WriteLine("Valid: '{0}' ({1:ss\.fffffff} seconds)",
                                    inputValue, sw.Elapsed)
               Else
                  sw.Stop()
                  Console.WriteLine("'{0}' is not a valid string. ({1:ss\.fffff} seconds)",
                                    inputValue, sw.Elapsed)
               End If
            Catch e As RegexMatchTimeoutException
               sw.Stop()
               ' Display the elapsed time until the exception.
               Console.WriteLine("Timeout with '{0}' after {1:ss\.fffff}",
                                 inputValue, sw.Elapsed)
               Thread.Sleep(1500)       ' Pause for 1.5 seconds.

               ' Increase the timeout interval and retry.
               Dim timeout As TimeSpan = e.MatchTimeout.Add(TimeSpan.FromSeconds(1))
               If timeout.TotalSeconds > MaxTimeoutInSeconds Then
                  Console.WriteLine("Maximum timeout interval of {0} seconds exceeded.",
                                    MaxTimeoutInSeconds)
```

```
                  timedOut = False
               Else
                  Console.WriteLine("Changing the timeout interval to {0}",
                                    timeout)
                  rgx = New Regex(pattern, RegexOptions.IgnoreCase, timeout)
                  timedOut = True
               End If
            End Try
         Loop While timedOut
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays output like the following:
'    Processing aa
'    Valid: 'aa' (00.0000779 seconds)
'
'    Processing aaaa>
'    'aaaa>' is not a valid string. (00.00005 seconds)
'
'    Processing aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
'    Valid: 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' (00.0000043 seconds)
'
'    Processing aaaaaaaaaaaaaaaaaaaaaa>
'    Timeout with 'aaaaaaaaaaaaaaaaaaaaaa>' after 01.00469
'    Changing the timeout interval to 00:00:02
'    Timeout with 'aaaaaaaaaaaaaaaaaaaaaa>' after 02.01202
'    Changing the timeout interval to 00:00:03
'    Timeout with 'aaaaaaaaaaaaaaaaaaaaaa>' after 03.01043
'    Maximum timeout interval of 3 seconds exceeded.
'
'    Processing aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>
'    Timeout with 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>' after 03.01018
'    Maximum timeout interval of 3 seconds exceeded.
```

Nonbacktracking Subexpression

The `(?>` *subexpression* `)` language element suppresses backtracking in a subexpression. It is useful for preventing the performance problems associated with failed matches.

The following example illustrates how suppressing backtracking improves performance when using nested quantifiers. It measures the time required for the regular expression engine to determine that an input string does not match two regular expressions. The first regular expression uses backtracking to attempt to match a string that contains one or more occurrences of one or more hexadecimal digits, followed by a colon, followed by one or more hexadecimal digits, followed by two colons. The second regular expression is identical to the first, except that it disables backtracking. As the output from the example shows, the performance improvement from disabling backtracking is significant.

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "b51:4:1DB:9EE1:5:27d60:f44:D4:cd:E:5:0A5:4a:D24:41Ad:";
      bool matched;
      Stopwatch sw;

      Console.WriteLine("With backtracking:");
      string backPattern = "^(([0-9a-fA-F]{1,4}:)*([0-9a-fA-F]{1,4}))*(::)$";
      sw = Stopwatch.StartNew();
      matched = Regex.IsMatch(input, backPattern);
      sw.Stop();
      Console.WriteLine("Match: {0} in {1}", Regex.IsMatch(input, backPattern), sw.Elapsed);
      Console.WriteLine();

      Console.WriteLine("Without backtracking:");
      string noBackPattern = "^((?>[0-9a-fA-F]{1,4}:)*(?>[0-9a-fA-F]{1,4}))*(::)$";
      sw = Stopwatch.StartNew();
      matched = Regex.IsMatch(input, noBackPattern);
      sw.Stop();
      Console.WriteLine("Match: {0} in {1}", Regex.IsMatch(input, noBackPattern), sw.Elapsed);
   }
}
// The example displays output like the following:
//       With backtracking:
//       Match: False in 00:00:27.4282019
//
//       Without backtracking:
//       Match: False in 00:00:00.0001391
```

```vb
Imports System.Diagnostics
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "b51:4:1DB:9EE1:5:27d60:f44:D4:cd:E:5:0A5:4a:D24:41Ad:"
      Dim matched As Boolean
      Dim sw As Stopwatch

      Console.WriteLine("With backtracking:")
      Dim backPattern As String = "^(([0-9a-fA-F]{1,4}:)*([0-9a-fA-F]{1,4}))*(::)$"
      sw = Stopwatch.StartNew()
      matched = Regex.IsMatch(input, backPattern)
      sw.Stop()
      Console.WriteLine("Match: {0} in {1}", Regex.IsMatch(input, backPattern), sw.Elapsed)
      Console.WriteLine()

      Console.WriteLine("Without backtracking:")
      Dim noBackPattern As String = "^((?>[0-9a-fA-F]{1,4}:)*(?>[0-9a-fA-F]{1,4}))*(::)$"
      sw = Stopwatch.StartNew()
      matched = Regex.IsMatch(input, noBackPattern)
      sw.Stop()
      Console.WriteLine("Match: {0} in {1}", Regex.IsMatch(input, noBackPattern), sw.Elapsed)
   End Sub
End Module
' The example displays the following output:
'       With backtracking:
'       Match: False in 00:00:27.4282019
'
'       Without backtracking:
'       Match: False in 00:00:00.0001391
```

Lookbehind Assertions

.NET includes two language elements, `(?<=` *subexpression* `)` and `(?<!` *subexpression* `)`, that match the previous character or characters in the input string. Both language elements are zero-width assertions; that is, they determine whether the character or characters that immediately precede the current character can be matched by *subexpression*, without advancing or backtracking.

`(?<=` *subexpression* `)` is a positive lookbehind assertion; that is, the character or characters before the current position must match *subexpression*. `(?<!` *subexpression* `)` is a negative lookbehind assertion; that is, the character or characters before the current position must not match *subexpression*. Both positive and negative lookbehind assertions are most useful when *subexpression* is a subset of the previous subexpression.

The following example uses two equivalent regular expression patterns that validate the user name in an email address. The first pattern is subject to poor performance because of excessive backtracking. The second pattern modifies the first regular expression by replacing a nested quantifier with a positive lookbehind assertion. The output from the example displays the execution time of the Regex.IsMatch method.

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      Stopwatch sw;
      string input = "test@contoso.com";
      bool result;

      string pattern = @"^[0-9A-Z]([-.\w]*[0-9A-Z])?@";
      sw = Stopwatch.StartNew();
      result = Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase);
      sw.Stop();
      Console.WriteLine("Match: {0} in {1}", result, sw.Elapsed);

      string behindPattern = @"^[0-9A-Z][-.\w]*(?<=[0-9A-Z])@";
      sw = Stopwatch.StartNew();
      result = Regex.IsMatch(input, behindPattern, RegexOptions.IgnoreCase);
      sw.Stop();
      Console.WriteLine("Match with Lookbehind: {0} in {1}", result, sw.Elapsed);
   }
}
// The example displays output similar to the following:
//       Match: True in 00:00:00.0017549
//       Match with Lookbehind: True in 00:00:00.0000659
```

```
Module Example
   Public Sub Main()
      Dim sw As Stopwatch
      Dim input As String = "test@contoso.com"
      Dim result As Boolean

      Dim pattern As String = "^[0-9A-Z]([-.\w]*[0-9A-Z])?@"
      sw = Stopwatch.StartNew()
      result = Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase)
      sw.Stop()
      Console.WriteLine("Match: {0} in {1}", result, sw.Elapsed)

      Dim behindPattern As String = "^[0-9A-Z][-.\w]*(?<=[0-9A-Z])@"
      sw = Stopwatch.StartNew()
      result = Regex.IsMatch(input, behindPattern, RegexOptions.IgnoreCase)
      sw.Stop()
      Console.WriteLine("Match with Lookbehind: {0} in {1}", result, sw.Elapsed)
   End Sub
End Module
' The example displays output similar to the following:
'       Match: True in 00:00:00.0017549
'       Match with Lookbehind: True in 00:00:00.0000659
```

The first regular expression pattern, `^[0-9A-Z]([-.\w]*[0-9A-Z])*@`, is defined as shown in the following table.

PATTERN	DESCRIPTION
`^`	Start the match at the beginning of the string.
`[0-9A-Z]`	Match an alphanumeric character. This comparison is case-insensitive, because the Regex.IsMatch method is called with the RegexOptions.IgnoreCase option.

PATTERN	DESCRIPTION
`[-.\w]*`	Match zero, one, or more occurrences of a hyphen, period, or word character.
`[0-9A-Z]`	Match an alphanumeric character.
`([-.\w]*[0-9A-Z])*`	Match zero or more occurrences of the combination of zero or more hyphens, periods, or word characters, followed by an alphanumeric character. This is the first capturing group.
`@`	Match an at sign ("@").

The second regular expression pattern, `^[0-9A-Z][-.\w]*(?<=[0-9A-Z])@`, uses a positive lookbehind assertion. It is defined as shown in the following table.

PATTERN	DESCRIPTION
`^`	Start the match at the beginning of the string.
`[0-9A-Z]`	Match an alphanumeric character. This comparison is case-insensitive, because the Regex.IsMatch method is called with the RegexOptions.IgnoreCase option.
`[-.\w]*`	Match zero or more occurrences of a hyphen, period, or word character.
`(?<=[0-9A-Z])`	Look back at the last matched character and continue the match if it is alphanumeric. Note that alphanumeric characters are a subset of the set that consists of periods, hyphens, and all word characters.
`@`	Match an at sign ("@").

Lookahead Assertions

.NET includes two language elements, `(?=` *subexpression* `)` and `(?!` *subexpression* `)`, that match the next character or characters in the input string. Both language elements are zero-width assertions; that is, they determine whether the character or characters that immediately follow the current character can be matched by *subexpression*, without advancing or backtracking.

`(?=` *subexpression* `)` is a positive lookahead assertion; that is, the character or characters after the current position must match *subexpression*. `(?!` *subexpression* `)` is a negative lookahead assertion; that is, the character or characters after the current position must not match *subexpression*. Both positive and negative lookahead assertions are most useful when *subexpression* is a subset of the next subexpression.

The following example uses two equivalent regular expression patterns that validate a fully qualified type name. The first pattern is subject to poor performance because of excessive backtracking. The second modifies the first regular expression by replacing a nested quantifier with a positive lookahead assertion. The output from the example displays the execution time of the Regex.IsMatch method.

```csharp
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "aaaaaaaaaaaaaaaaaaaaaa.";
      bool result;
      Stopwatch sw;

      string pattern = @"^(([A-Z]\w*)+\.)*[A-Z]\w*$";
      sw = Stopwatch.StartNew();
      result = Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase);
      sw.Stop();
      Console.WriteLine("{0} in {1}", result, sw.Elapsed);

      string aheadPattern = @"^((?=[A-Z])\w+\.)*[A-Z]\w*$";
      sw = Stopwatch.StartNew();
      result = Regex.IsMatch(input, aheadPattern, RegexOptions.IgnoreCase);
      sw.Stop();
      Console.WriteLine("{0} in {1}", result, sw.Elapsed);
   }
}
// The example displays the following output:
//       False in 00:00:03.8003793
//       False in 00:00:00.0000866
```

```vb
Imports System.Diagnostics
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "aaaaaaaaaaaaaaaaaaaaaa."
      Dim result As Boolean
      Dim sw As Stopwatch

      Dim pattern As String = "^(([A-Z]\w*)+\.)*[A-Z]\w*$"
      sw = Stopwatch.StartNew()
      result = Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase)
      sw.Stop()
      Console.WriteLine("{0} in {1}", result, sw.Elapsed)

      Dim aheadPattern As String = "^((?=[A-Z])\w+\.)*[A-Z]\w*$"
      sw = Stopwatch.StartNew()
      result = Regex.IsMatch(input, aheadPattern, RegexOptions.IgnoreCase)
      sw.Stop()
      Console.WriteLine("{0} in {1}", result, sw.Elapsed)
   End Sub
End Module
' The example displays the following output:
'       False in 00:00:03.8003793
'       False in 00:00:00.0000866
```

The first regular expression pattern, `^(([A-Z]\w*)+\.)*[A-Z]\w*$`, is defined as shown in the following table.

PATTERN	DESCRIPTION
`^`	Start the match at the beginning of the string.

PATTERN	DESCRIPTION
`([A-Z]\w*)+\.`	Match an alphabetical character (A-Z) followed by zero or more word characters one or more times, followed by a period. This comparison is case-insensitive, because the Regex.IsMatch method is called with the RegexOptions.IgnoreCase option.
`(([A-Z]\w*)+\.)*`	Match the previous pattern zero or more times.
`[A-Z]\w*`	Match an alphabetical character followed by zero or more word characters.
`$`	End the match at the end of the input string.

The second regular expression pattern, `^((?=[A-Z])\w+\.)*[A-Z]\w*$`, uses a positive lookahead assertion. It is defined as shown in the following table.

PATTERN	DESCRIPTION
`^`	Start the match at the beginning of the string.
`(?=[A-Z])`	Look ahead to the first character and continue the match if it is alphabetical (A-Z). This comparison is case-insensitive, because the Regex.IsMatch method is called with the RegexOptions.IgnoreCase option.
`\w+\.`	Match one or more word characters followed by a period.
`((?=[A-Z])\w+\.)*`	Match the pattern of one or more word characters followed by a period zero or more times. The initial word character must be alphabetical.
`[A-Z]\w*`	Match an alphabetical character followed by zero or more word characters.
`$`	End the match at the end of the input string.

Back to top

See also

- .NET Regular Expressions
- Regular Expression Language - Quick Reference
- Quantifiers
- Alternation Constructs
- Grouping Constructs

Compilation and Reuse in Regular Expressions

9/6/2018 • 2 minutes to read • Edit Online

You can optimize the performance of applications that make extensive use of regular expressions by understanding how the regular expression engine compiles expressions and by understanding how regular expressions are cached. This topic discusses both compilation and caching.

Compiled Regular Expressions

By default, the regular expression engine compiles a regular expression to a sequence of internal instructions (these are high-level codes that are different from Microsoft intermediate language, or MSIL). When the engine executes a regular expression, it interprets the internal codes.

If a Regex object is constructed with the RegexOptions.Compiled option, it compiles the regular expression to explicit MSIL code instead of high-level regular expression internal instructions. This allows .NET's just-in-time (JIT) compiler to convert the expression to native machine code for higher performance.

However, generated MSIL cannot be unloaded. The only way to unload code is to unload an entire application domain (that is, to unload all of your application's code.). Effectively, once a regular expression is compiled with the RegexOptions.Compiled option, never releases the resources used by the compiled expression, even if the regular expression was created by a Regex object that is itself released to garbage collection.

You must be careful to limit the number of different regular expressions you compile with the RegexOptions.Compiled option to avoid consuming too many resources. If an application must use a large or unbounded number of regular expressions, each expression should be interpreted, not compiled. However, if a small number of regular expressions are used repeatedly, they should be compiled with RegexOptions.Compiled for better performance. An alternative is to use precompiled regular expressions. You can compile all of your expressions into a reusable DLL by using the CompileToAssembly method. This avoids the need to compile at runtime while still benefiting from the speed of compiled regular expressions.

The Regular Expressions Cache

To improve performance, the regular expression engine maintains an application-wide cache of compiled regular expressions. The cache stores regular expression patterns that are used only in static method calls. (Regular expression patterns supplied to instance methods are not cached.) This avoids the need to reparse an expression into high-level byte code each time it is used.

The maximum number of cached regular expressions is determined by the value of the `static` (`Shared` in Visual Basic) Regex.CacheSize property. By default, the regular expression engine caches up to 15 compiled regular expressions. If the number of compiled regular expressions exceeds the cache size, the least recently used regular expression is discarded and the new regular expression is cached.

Your application can take advantage of precompiled regular expressions in one of the following two ways:

- By using a static method of the Regex object to define the regular expression. If you are using a regular expression pattern that has already been defined in another static method call, the regular expression engine will retrieve it from the cache. If not, the engine will compile the regular expression and add it to the cache.
- By reusing an existing Regex object as long as its regular expression pattern is needed.

Because of the overhead of object instantiation and regular expression compilation, creating and rapidly destroying numerous Regex objects is a very expensive process. For applications that use a large number of different regular expressions, you can optimize performance by using calls to static `Regex` methods and possibly by increasing the size of the regular expression cache.

See also

- .NET Regular Expressions

Thread Safety in Regular Expressions

9/6/2018 • 2 minutes to read • Edit Online

The Regex class itself is thread safe and immutable (read-only). That is, **Regex** objects can be created on any thread and shared between threads; matching methods can be called from any thread and never alter any global state.

However, result objects (**Match** and **MatchCollection**) returned by **Regex** should be used on a single thread. Although many of these objects are logically immutable, their implementations could delay computation of some results to improve performance, and as a result, callers must serialize access to them.

If there is a need to share **Regex** result objects on multiple threads, these objects can be converted to thread-safe instances by calling their synchronized methods. With the exception of enumerators, all regular expression classes are thread safe or can be converted into thread-safe objects by a synchronized method.

Enumerators are the only exception. An application must serialize calls to collection enumerators. The rule is that if a collection can be enumerated on more than one thread simultaneously, you should synchronize enumerator methods on the root object of the collection traversed by the enumerator.

See also

- .NET Regular Expressions

Regular Expression Examples

5/15/2018 • 2 minutes to read • Edit Online

This section contains code examples that illustrate the use of regular expressions in common applications.

> **NOTE**
>
> The System.Web.RegularExpressions namespace contains a number of regular expression objects that implement predefined regular expression patterns for parsing strings from HTML, XML, and ASP.NET documents. For example, the TagRegex class identifies start tags in a string and the CommentRegex class identifies ASP.NET comments in a string.

In This Section

Example: Scanning for HREFs

Provides an example that searches an input string and prints out all the href="..." values and their locations in the string.

Example: Changing Date Formats

Provides an example that replaces dates in the form mm/dd/yy with dates in the form dd-mm-yy.

How to: Extract a Protocol and Port Number from a URL

Provides an example that extracts a protocol and port number from a string that contains a URL. For example, "http://www.contoso.com:8080/letters/readme.html" returns "http:8080".

How to: Strip Invalid Characters from a String

Provides an example that strips invalid non-alphanumeric characters from a string.

How to: Verify that Strings Are in Valid Email Format

Provides an example that verifies that a string is in valid email format.

Reference

System.Text.RegularExpressions

Provides class library reference information for the .NET **System.Text.RegularExpressions** namespace.

Related Sections

.NET Regular Expressions

Provides an overview of the programming language aspect of regular expressions.

The Regular Expression Object Model

Describes the regular expression classes contained in the `System.Text.RegularExpression` namespace and provides examples of their use.

Details of Regular Expression Behavior

Provides information about the capabilities and behavior of .NET regular expressions.

Regular Expression Language - Quick Reference

Provides information on the set of characters, operators, and constructs that you can use to define regular expressions.

Regular Expression Example: Scanning for HREFs

9/6/2018 • 3 minutes to read • Edit Online

The following example searches an input string and displays all the href="..." values and their locations in the string.

The Regex Object

Because the `DumpHRefs` method can be called multiple times from user code, it uses the `static` (`Shared` in Visual Basic) Regex.Match(String, String, RegexOptions) method. This enables the regular expression engine to cache the regular expression and avoids the overhead of instantiating a new Regex object each time the method is called. A Match object is then used to iterate through all matches in the string.

```
private static void DumpHRefs(string inputString)
{
   Match m;
   string HRefPattern = "href\\s*=\\s*(?:[\"'](?<1>[^\"']*)[\"']|(?<1>\\S+))";

   try {
      m = Regex.Match(inputString, HRefPattern,
                      RegexOptions.IgnoreCase | RegexOptions.Compiled,
                      TimeSpan.FromSeconds(1));
      while (m.Success)
      {
         Console.WriteLine("Found href " + m.Groups[1] + " at "
            + m.Groups[1].Index);
         m = m.NextMatch();
      }
   }
   catch (RegexMatchTimeoutException) {
      Console.WriteLine("The matching operation timed out.");
   }
}
```

```
Private Sub DumpHRefs(inputString As String)
   Dim m As Match
   Dim HRefPattern As String = "href\s*=\s*(?:[""'](?<1>[^""']*)[""']|(?<1>\S+))"

   Try
      m = Regex.Match(inputString, HRefPattern, _
                      RegexOptions.IgnoreCase Or RegexOptions.Compiled,
                      TimeSpan.FromSeconds(1))
      Do While m.Success
         Console.WriteLine("Found href {0} at {1}.", _
                           m.Groups(1), m.Groups(1).Index)
         m = m.NextMatch()
      Loop
   Catch e As RegexMatchTimeoutException
      Console.WriteLine("The matching operation timed out.")
   End Try
End Sub
```

The following example then illustrates a call to the `DumpHRefs` method.

```
public static void Main()
{
   string inputString = "My favorite web sites include:</P>" +
                        "<A HREF=\"http://msdn2.microsoft.com\">" +
                        "MSDN Home Page</A></P>" +
                        "<A HREF=\"http://www.microsoft.com\">" +
                        "Microsoft Corporation Home Page</A></P>" +
                        "<A HREF=\"http://blogs.msdn.com/bclteam\">" +
                        ".NET Base Class Library blog</A></P>";
   DumpHRefs(inputString);

}
// The example displays the following output:
//       Found href http://msdn2.microsoft.com at 43
//       Found href http://www.microsoft.com at 102
//       Found href http://blogs.msdn.com/bclteam at 176
```

```
Public Sub Main()
   Dim inputString As String = "My favorite web sites include:</P>" & _
                               "<A HREF=""http://msdn2.microsoft.com"">" & _
                               "MSDN Home Page</A></P>" & _
                               "<A HREF=""http://www.microsoft.com"">" & _
                               "Microsoft Corporation Home Page</A></P>" & _
                               "<A HREF=""http://blogs.msdn.com/bclteam"">" & _
                               ".NET Base Class Library blog</A></P>"
   DumpHRefs(inputString)
End Sub
' The example displays the following output:
'       Found href http://msdn2.microsoft.com at 43
'       Found href http://www.microsoft.com at 102
'       Found href http://blogs.msdn.com/bclteam/) at 176
```

The regular expression pattern `href\s*=\s*(?:["'](?<1>[^"']*)["']|(?<1>\S+))` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`href`	Match the literal string "href". The match is case-insensitive.
`\s*`	Match zero or more white-space characters.
`=`	Match the equals sign.
`\s*`	Match zero or more white-space characters.
`(?:["'](?<1>[^"']*)["']|(?<1>\S+))`	Match one of the following without assigning the result to a captured group: • A quotation mark or apostrophe, followed by zero or more occurrences of any character other than a quotation mark or apostrophe, followed by a quotation mark or apostrophe. The group named `1` is included in this pattern. • One or more non-white-space characters. The group named `1` is included in this pattern.

PATTERN	DESCRIPTION
`(?<1>[^"']*)`	Assign zero or more occurrences of any character other than a quotation mark or apostrophe to the capturing group named `1`.
`(?<1>\S+)`	Assign one or more non-white-space characters to the capturing group named `1`.

Match Result Class

The results of a search are stored in the Match class, which provides access to all the substrings extracted by the search. It also remembers the string being searched and the regular expression being used, so it can call the Match.NextMatch method to perform another search starting where the last one ended.

Explicitly Named Captures

In traditional regular expressions, capturing parentheses are automatically numbered sequentially. This leads to two problems. First, if a regular expression is modified by inserting or removing a set of parentheses, all code that refers to the numbered captures must be rewritten to reflect the new numbering. Second, because different sets of parentheses often are used to provide two alternative expressions for an acceptable match, it might be difficult to determine which of the two expressions actually returned a result.

To address these problems, the Regex class supports the syntax `(?<name>…)` for capturing a match into a specified slot (the slot can be named using a string or an integer; integers can be recalled more quickly). Thus, alternative matches for the same string all can be directed to the same place. In case of a conflict, the last match dropped into a slot is the successful match. (However, a complete list of multiple matches for a single slot is available. See the Group.Captures collection for details.)

See also

- .NET Regular Expressions

Regular Expression Example: Changing Date Formats

9/6/2018 • 2 minutes to read • Edit Online

The following code example uses the Regex.Replace method to replace dates that have the form *mm/dd/yy* with dates that have the form *dd-mm-yy*.

Example

```
static string MDYToDMY(string input)
{
   try {
      return Regex.Replace(input,
            "\\b(?<month>\\d{1,2})/(?<day>\\d{1,2})/(?<year>\\d{2,4})\\b",
            "${day}-${month}-${year}", RegexOptions.None,
            TimeSpan.FromMilliseconds(150));
   }
   catch (RegexMatchTimeoutException) {
      return input;
   }
}
```

```
Function MDYToDMY(input As String) As String
   Try
      Return Regex.Replace(input, _
             "\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b", _
             "${day}-${month}-${year}", RegexOptions.None,
             TimeSpan.FromMilliseconds(150))
    Catch e As RegexMatchTimeoutException
       Return input
    End Try
End Function
```

The following code shows how the `MDYToDMY` method can be called in an application.

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class Class1
{
   public static void Main()
   {
      string dateString = DateTime.Today.ToString("d",
                                        DateTimeFormatInfo.InvariantInfo);
      string resultString = MDYToDMY(dateString);
      Console.WriteLine("Converted {0} to {1}.", dateString, resultString);
   }

   static string MDYToDMY(string input)
   {
      try {
         return Regex.Replace(input,
               "\\b(?<month>\\d{1,2})/(?<day>\\d{1,2})/(?<year>\\d{2,4})\\b",
               "${day}-${month}-${year}", RegexOptions.None,
               TimeSpan.FromMilliseconds(150));
      }
      catch (RegexMatchTimeoutException) {
         return input;
      }
   }

}
// The example displays the following output to the console if run on 8/21/2007:
//      Converted 08/21/2007 to 21-08-2007.
```

```
Imports System.Globalization
Imports System.Text.RegularExpressions

Module DateFormatReplacement
   Public Sub Main()
      Dim dateString As String = Date.Today.ToString("d", _
                                           DateTimeFormatInfo.InvariantInfo)
      Dim resultString As String = MDYToDMY(dateString)
      Console.WriteLine("Converted {0} to {1}.", dateString, resultString)
   End Sub

    Function MDYToDMY(input As String) As String
       Try
          Return Regex.Replace(input, _
                 "\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b", _
                 "${day}-${month}-${year}", RegexOptions.None,
                 TimeSpan.FromMilliseconds(150))
        Catch e As RegexMatchTimeoutException
           Return input
        End Try
    End Function
End Module
' The example displays the following output to the console if run on 8/21/2007:
'      Converted 08/21/2007 to 21-08-2007.
```

Comments

The regular expression pattern `\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b` is interpreted as shown in the following table.

PATTERN	DESCRIPTION
`\b`	Begin the match at a word boundary.
`(?<month>\d{1,2})`	Match one or two decimal digits. This is the `month` captured group.
`/`	Match the slash mark.
`(?<day>\d{1,2})`	Match one or two decimal digits. This is the `day` captured group.
`/`	Match the slash mark.
`(?<year>\d{2,4})`	Match from two to four decimal digits. This is the `year` captured group.
`\b`	End the match at a word boundary.

The pattern `${day}-${month}-${year}` defines the replacement string as shown in the following table.

PATTERN	DESCRIPTION
`$(day)`	Add the string captured by the `day` capturing group.
`-`	Add a hyphen.
`$(month)`	Add the string captured by the `month` capturing group.
`-`	Add a hyphen.
`$(year)`	Add the string captured by the `year` capturing group.

See also

- .NET Regular Expressions

How to: Extract a Protocol and Port Number from a URL

9/6/2018 • 2 minutes to read • Edit Online

The following example extracts a protocol and port number from a URL.

Example

The example uses the Match.Result method to return the protocol followed by a colon followed by the port number.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string url = "http://www.contoso.com:8080/letters/readme.html";

      Regex r = new Regex(@"^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/",
                          RegexOptions.None, TimeSpan.FromMilliseconds(150));
      Match m = r.Match(url);
      if (m.Success)
         Console.WriteLine(r.Match(url).Result("${proto}${port}"));
   }
}
// The example displays the following output:
//       http:8080
```

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim url As String = "http://www.contoso.com:8080/letters/readme.html"
      Dim r As New Regex("^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/",
                         RegexOptions.None, TimeSpan.FromMilliseconds(150))

      Dim m As Match = r.Match(url)
      If m.Success Then
         Console.WriteLine(r.Match(url).Result("${proto}${port}"))
      End If
   End Sub
End Module
' The example displays the following output:
'       http:8080
```

The regular expression pattern `^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/` can be interpreted as shown in the following table.

PATTERN	DESCRIPTION
`^`	Begin the match at the start of the string.

PATTERN	DESCRIPTION
`(?<proto>\w+)`	Match one or more word characters. Name this group `proto`.
`://`	Match a colon followed by two slash marks.
`[^/]+?`	Match one or more occurrences (but as few as possible) of any character other than a slash mark.
`(?<port>:\d+)?`	Match zero or one occurrence of a colon followed by one or more digit characters. Name this group `port`.
`/`	Match a slash mark.

The Match.Result method expands the `${proto}${port}` replacement sequence, which concatenates the value of the two named groups captured in the regular expression pattern. It is a convenient alternative to explicitly concatenating the strings retrieved from the collection object returned by the Match.Groups property.

The example uses the Match.Result method with two substitutions, `${proto}` and `${port}`, to include the captured groups in the output string. You can retrieve the captured groups from the match's GroupCollection object instead, as the following code shows.

```
Console.WriteLine(m.Groups["proto"].Value + m.Groups["port"].Value);
```

```
Console.WriteLine(m.Groups("proto").Value + m.Groups("port").Value)
```

See also

- .NET Regular Expressions

How to: Strip Invalid Characters from a String

9/6/2018 • 2 minutes to read • Edit Online

The following example uses the static Regex.Replace method to strip invalid characters from a string.

Example

You can use the `CleanInput` method defined in this example to strip potentially harmful characters that have been entered into a text field that accepts user input. In this case, `CleanInput` strips out all nonalphanumeric characters except periods (.), at symbols (@), and hyphens (-), and returns the remaining string. However, you can modify the regular expression pattern so that it strips out any characters that should not be included in an input string.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    static string CleanInput(string strIn)
    {
        // Replace invalid characters with empty strings.
        try {
           return Regex.Replace(strIn, @"[^\w\.@-]", "",
                                RegexOptions.None, TimeSpan.FromSeconds(1.5));
        }
        // If we timeout when replacing invalid characters,
        // we should return Empty.
        catch (RegexMatchTimeoutException) {
           return String.Empty;
        }
    }
}
```

```
Imports System.Text.RegularExpressions

Module Example
    Function CleanInput(strIn As String) As String
        ' Replace invalid characters with empty strings.
        Try
           Return Regex.Replace(strIn, "[^\w\.@-]", "")
        ' If we timeout when replacing invalid characters,
        ' we should return String.Empty.
        Catch e As RegexMatchTimeoutException
           Return String.Empty
        End Try
    End Function
End Module
```

The regular expression pattern `[^\w\.@-]` matches any character that is not a word character, a period, an @ symbol, or a hyphen. A word character is any letter, decimal digit, or punctuation connector such as an underscore. Any character that matches this pattern is replaced by String.Empty, which is the string defined by the replacement pattern. To allow additional characters in user input, add those characters to the character class in the regular expression pattern. For example, the regular expression pattern `[^\w\.@-\\%]` also allows a percentage symbol and a backslash in an input string.

See also

- .NET Regular Expressions

How to: Verify that Strings Are in Valid Email Format

9/6/2018 • 7 minutes to read • Edit Online

The following example uses a regular expression to verify that a string is in valid email format.

Example

The example defines an `IsValidEmail` method, which returns `true` if the string contains a valid email address and `false` if it does not, but takes no other action.

To verify that the email address is valid, the `IsValidEmail` method calls the Regex.Replace(String, String, MatchEvaluator) method with the `(@)(.+)$` regular expression pattern to separate the domain name from the email address. The third parameter is a MatchEvaluator delegate that represents the method that processes and replaces the matched text. The regular expression pattern is interpreted as follows.

PATTERN	DESCRIPTION
`(@)`	Match the @ character. This is the first capturing group.
`(.+)`	Match one or more occurrences of any character. This is the second capturing group.
`$`	End the match at the end of the string.

The domain name along with the @ character is passed to the `DomainMapper` method, which uses the IdnMapping class to translate Unicode characters that are outside the US-ASCII character range to Punycode. The method also sets the `invalid` flag to `True` if the IdnMapping.GetAscii method detects any invalid characters in the domain name. The method returns the Punycode domain name preceded by the @ symbol to the `IsValidEmail` method.

The `IsValidEmail` method then calls the Regex.IsMatch(String, String) method to verify that the address conforms to a regular expression pattern.

Note that the `IsValidEmail` method does not perform authentication to validate the email address. It merely determines whether its format is valid for an email address. In addition, the `IsValidEmail` method does not verify that the top-level domain name is a valid domain name listed at the IANA Root Zone Database, which would require a look-up operation. Instead, the regular expression merely verifies that the top-level domain name consists of between two and twenty-four ASCII characters, with alphanumeric first and last characters and the remaining characters being either alphanumeric or a hyphen (-).

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class RegexUtilities
{
   bool invalid = false;

   public bool IsValidEmail(string strIn)
   {
       invalid = false;
       if (String.IsNullOrEmpty(strIn))
          return false;

       // Use IdnMapping class to convert Unicode domain names.
       try {
          strIn = Regex.Replace(strIn, @"(@)(.+)$", this.DomainMapper,
                                RegexOptions.None, TimeSpan.FromMilliseconds(200));
       }
       catch (RegexMatchTimeoutException) {
         return false;
       }

        if (invalid)
           return false;

       // Return true if strIn is in valid email format.
       try {
          return Regex.IsMatch(strIn,
                @"^(?("")("".+?(?<!\\)""@)|(([0-9a-z]((\.(?!\.))|[-!#\$%&'\*\+/=\?\^`\{\}\|~\w])*)(?<=[0-9a-
z])@))" +
                @"(?(\[)(\[(\d{1,3}\.){3}\d{1,3}\])|(([0-9a-z][-0-9a-z]*[0-9a-z]*\.)+[a-z0-9][\-a-z0-9]{0,22}
[a-z0-9]))$",
                RegexOptions.IgnoreCase, TimeSpan.FromMilliseconds(250));
       }
       catch (RegexMatchTimeoutException) {
          return false;
       }
   }

   private string DomainMapper(Match match)
   {
      // IdnMapping class with default property values.
      IdnMapping idn = new IdnMapping();

      string domainName = match.Groups[2].Value;
      try {
         domainName = idn.GetAscii(domainName);
      }
      catch (ArgumentException) {
         invalid = true;
      }
      return match.Groups[1].Value + domainName;
   }
}
```

```
Imports System.Globalization
Imports System.Text.RegularExpressions

Public Class RegexUtilities
   Dim invalid As Boolean = False

   public Function IsValidEmail(strIn As String) As Boolean
       invalid = False
       If String.IsNullOrEmpty(strIn) Then Return False

       ' Use IdnMapping class to convert Unicode domain names.
       Try
          strIn = Regex.Replace(strIn, "(@)(.+)$", AddressOf Me.DomainMapper,
                                RegexOptions.None, TimeSpan.FromMilliseconds(200))
       Catch e As RegexMatchTimeoutException
          Return False
       End Try

       If invalid Then Return False

       ' Return true if strIn is in valid email format.
       Try
          Return Regex.IsMatch(strIn,
                 "^(?("")("".+?(?<!\\)""@)|(([0-9a-z]((\.(?!\.))|[-!#\$%&'\*\+/=\?\^`\{\}\|~\w])*)(?<=[0-9a-
z])@))" +
                 "(?(\[)(\[(\d{1,3}\.){3}\d{1,3}\])|(([0-9a-z][-0-9a-z]*[0-9a-z]*\.)+[a-z0-9][\-a-z0-9]{0,22}
[a-z0-9]))$",
                 RegexOptions.IgnoreCase, TimeSpan.FromMilliseconds(250))
       Catch e As RegexMatchTimeoutException
          Return False
       End Try
   End Function

   Private Function DomainMapper(match As Match) As String
      ' IdnMapping class with default property values.
      Dim idn As New IdnMapping()

      Dim domainName As String = match.Groups(2).Value
      Try
         domainName = idn.GetAscii(domainName)
      Catch e As ArgumentException
         invalid = True
      End Try
      Return match.Groups(1).Value + domainName
   End Function
End Class
```

In this example, the regular expression pattern
``^(?(")(".+?(?<!\\)"@)|(([0-9a-z]((\.(?!\.))|[-!#\$%&'\*\+/=\?\^`{}|~\w])*)(?<=[0-9a-z])@))(?([)([(\d{1,3}.){3}\d{1,3}])|(([0-9a-z][-0-9a-z]*[0-9a-z]*.)+[a-z0-9][-a-z0-9]{0,22}[a-z0-9]))$``
is interpreted as shown in the following table. Note that the regular expression is compiled using the RegexOptions.IgnoreCase flag.

PATTERN	DESCRIPTION
`^`	Begin the match at the start of the string.
`(?(")`	Determine whether the first character is a quotation mark. `(?(")` is the beginning of an alternation construct.

PATTERN	DESCRIPTION
`(?("")("".+?(?<!\\)""@)`	If the first character is a quotation mark, match a beginning quotation mark followed by at least one occurrence of any character, followed by an ending quotation mark. The ending quotation mark must not be preceded by a backslash character (\). `(?<!` is the beginning of a zero-width negative lookbehind assertion. The string should conclude with an at sign (@).
`|(([0-9a-z]`	If the first character is not a quotation mark, match any alphabetic character from a to z or A to Z (the comparison is case insensitive), or any numeric character from 0 to 9.
`(\.(?!\.))`	If the next character is a period, match it. If it is not a period, look ahead to the next character and continue the match. `(?!\.)` is a zero-width negative lookahead assertion that prevents two consecutive periods from appearing in the local part of an email address.
`` |[-!#$%&'*+/=?^`{}|~\w] ``	If the next character is not a period, match any word character or one of the following characters: -!#$%'*+=?^`{}|~.
`` ((.(?!.))|[-!#$%'*+/=?^`{}|~\w])* ``	Match the alternation pattern (a period followed by a non-period, or one of a number of characters) zero or more times.
`@`	Match the @ character.
`(?<=[0-9a-z])`	Continue the match if the character that precedes the @ character is A through Z, a through z, or 0 through 9. The `(?<=[0-9a-z])` construct defines a zero-width positive lookbehind assertion.
`(?(\[)`	Check whether the character that follows @ is an opening bracket.
`(\[(\d{1,3}\.){3}\d{1,3}\])`	If it is an opening bracket, match the opening bracket followed by an IP address (four sets of one to three digits, with each set separated by a period) and a closing bracket.
`|(([0-9a-z][-0-9a-z][0-9a-z].)+`	If the character that follows @ is not an opening bracket, match one alphanumeric character with a value of A-Z, a-z, or 0-9, followed by zero or more occurrences of a hyphen, followed by zero or one alphanumeric character with a value of A-Z, a-z, or 0-9, followed by a period. This pattern can be repeated one or more times, and must be followed by the top-level domain name.
`[a-z0-9][\-a-z0-9]{0,22}[a-z0-9]))`	The top-level domain name must begin and end with an alphanumeric character (a-z, A-Z, and 0-9). It can also include from zero to 22 ASCII characters that are either alphanumeric or hyphens.
`$`	End the match at the end of the string.

Compiling the Code

The `IsValidEmail` and `DomainMapper` methods can be included in a library of regular expression utility methods, or they can be included as private static or instance methods in the application class.

To include them in a regular expression library, either copy and paste the code into a Visual Studio Class Library project, or copy and paste it into a text file and compile it from the command line with a command like the following (assuming that the name of the source code file is RegexUtilities.cs or RegexUtilities.vb:

```
csc /t:library RegexUtilities.cs
```

```
vbc /t:library RegexUtilities.vb
```

You can also use the Regex.CompileToAssembly method to include this regular expression in a regular expression library.

If they are used in a regular expression library, you can call them by using code such as the following:

```
public class Application
{
   public static void Main()
   {
      RegexUtilities util = new RegexUtilities();
      string[] emailAddresses = { "david.jones@proseware.com", "d.j@server1.proseware.com",
                                  "jones@ms1.proseware.com", "j.@server1.proseware.com",
                                  "j@proseware.com9", "js#internal@proseware.com",
                                  "j_9@[129.126.118.1]", "j..s@proseware.com",
                                  "js*@proseware.com", "js@proseware..com",
                                  "js@proseware.com9", "j.s@server1.proseware.com",
                                   "\"j\\\"s\\\"\"@proseware.com", "js@contoso.中国" };

      foreach (var emailAddress in emailAddresses) {
         if (util.IsValidEmail(emailAddress))
            Console.WriteLine("Valid: {0}", emailAddress);
         else
            Console.WriteLine("Invalid: {0}", emailAddress);
      }
   }
}
// The example displays the following output:
//       Valid: david.jones@proseware.com
//       Valid: d.j@server1.proseware.com
//       Valid: jones@ms1.proseware.com
//       Invalid: j.@server1.proseware.com
//       Valid: j@proseware.com9
//       Valid: js#internal@proseware.com
//       Valid: j_9@[129.126.118.1]
//       Invalid: j..s@proseware.com
//       Invalid: js*@proseware.com
//       Invalid: js@proseware..com
//       Valid: js@proseware.com9
//       Valid: j.s@server1.proseware.com
//       Valid: "j\"s\""@proseware.com
//       Valid: js@contoso.ä¸å›½
```

```
Public Class Application
   Public Shared Sub Main()
      Dim util As New RegexUtilities()
      Dim emailAddresses() As String = { "david.jones@proseware.com", "d.j@server1.proseware.com", _
                                         "jones@ms1.proseware.com", "j.@server1.proseware.com", _
                                         "j@proseware.com9", "js#internal@proseware.com", _
                                         "j_9@[129.126.118.1]", "j..s@proseware.com", _
                                         "js*@proseware.com", "js@proseware..com", _
                                         "js@proseware.com9", "j.s@server1.proseware.com",
                                         """j\""s\""""@proseware.com", "js@contoso.中国" }

      For Each emailAddress As String In emailAddresses
         If util.IsValidEmail(emailAddress) Then
            Console.WriteLine("Valid: {0}", emailAddress)
         Else
            Console.WriteLine("Invalid: {0}", emailAddress)
         End If
      Next
   End Sub
End Class
' The example displays the following output:
'       Valid: david.jones@proseware.com
'       Valid: d.j@server1.proseware.com
'       Valid: jones@ms1.proseware.com
'       Invalid: j.@server1.proseware.com
'       Valid: j@proseware.com9
'       Valid: js#internal@proseware.com
'       Valid: j_9@[129.126.118.1]
'       Invalid: j..s@proseware.com
'       Invalid: js*@proseware.com
'       Invalid: js@proseware..com
'       Valid: js@proseware.com9
'       Valid: j.s@server1.proseware.com
'       Valid: "j\"s\""@proseware.com
'       Valid: js@contoso.ä¸å›½
```

Assuming you've created a class library named RegexUtilities.dll that includes your email validation regular expression, you can compile this example in either of the following ways:

- In Visual Studio, by creating a Console Application and adding a reference to RegexUtilities.dll to your project.
- From the command line, by copying and pasting the source code into a text file and compiling it with a command like the following (assuming that the name of the source code file is Example.cs or Example.vb:

```
csc Example.cs /r:RegexUtilities.dll
```

```
vbc Example.vb /r:RegexUtilities.dll
```

See also

- .NET Framework Regular Expressions

Character Encoding in .NET

9/6/2018 • 44 minutes to read • Edit Online

Characters are abstract entities that can be represented in many different ways. A character encoding is a system that pairs each character in a supported character set with some value that represents that character. For example, Morse code is a character encoding that pairs each character in the Roman alphabet with a pattern of dots and dashes that are suitable for transmission over telegraph lines. A character encoding for computers pairs each character in a supported character set with a numeric value that represents that character. A character encoding has two distinct components:

- An encoder, which translates a sequence of characters into a sequence of numeric values (bytes).
- A decoder, which translates a sequence of bytes into a sequence of characters.

Character encoding describes the rules by which an encoder and a decoder operate. For example, the UTF8Encoding class describes the rules for encoding to, and decoding from, 8-bit Unicode Transformation Format (UTF-8), which uses one to four bytes to represent a single Unicode character. Encoding and decoding can also include validation. For example, the UnicodeEncoding class checks all surrogates to make sure they constitute valid surrogate pairs. (A surrogate pair consists of a character with a code point that ranges from U+D800 to U+DBFF followed by a character with a code point that ranges from U+DC00 to U+DFFF.) A fallback strategy determines how an encoder handles invalid characters or how a decoder handles invalid bytes.

WARNING

.NET encoding classes provide a way to store and convert character data. They should not be used to store binary data in string form. Depending on the encoding used, converting binary data to string format with the encoding classes can introduce unexpected behavior and produce inaccurate or corrupted data. To convert binary data to a string form, use the Convert.ToBase64String method.

.NET uses the UTF-16 encoding (represented by the UnicodeEncoding class) to represent characters and strings. Applications that target the common language runtime use encoders to map Unicode character representations supported by the common language runtime to other encoding schemes. They use decoders to map characters from non-Unicode encodings to Unicode.

This topic consists of the following sections:

Encodings in .NET

All character encoding classes in .NET inherit from the System.Text.Encoding class, which is an abstract class that defines the functionality common to all character encodings. To access the individual encoding objects implemented in .NET, do the following:

- Use the static properties of the Encoding class, which return objects that represent the standard character encodings available in .NET (ASCII, UTF-7, UTF-8, UTF-16, and UTF-32). For example, the Encoding.Unicode property returns a UnicodeEncoding object. Each object uses replacement fallback to handle strings that it cannot encode and bytes that it cannot decode. (For more information, see the Replacement Fallback section.)

- Call the encoding's class constructor. Objects for the ASCII, UTF-7, UTF-8, UTF-16, and UTF-32 encodings can be instantiated in this way. By default, each object uses replacement fallback to handle strings that it cannot encode and bytes that it cannot decode, but you can specify that an exception should be thrown instead. (For more information, see the Replacement Fallback and Exception Fallback sections.)

- Call the Encoding.Encoding(Int32) constructor and pass it an integer that represents the encoding. Standard encoding objects use replacement fallback, and code page and double-byte character set (DBCS) encoding objects use best-fit fallback to handle strings that they cannot encode and bytes that they cannot decode. (For more information, see the Best-Fit Fallback section.)

- Call the Encoding.GetEncoding method, which returns any standard, code page, or DBCS encoding available in .NET. Overloads let you specify a fallback object for both the encoder and the decoder.

> **NOTE**
>
> The Unicode Standard assigns a code point (a number) and a name to each character in every supported script. For example, the character "A" is represented by the code point U+0041 and the name "LATIN CAPITAL LETTER A". The Unicode Transformation Format (UTF) encodings define ways to encode that code point into a sequence of one or more bytes. A Unicode encoding scheme simplifies world-ready application development because it allows characters from any character set to be represented in a single encoding. Application developers no longer have to keep track of the encoding scheme that was used to produce characters for a specific language or writing system, and data can be shared among systems internationally without being corrupted.
>
> .NET supports three encodings defined by the Unicode standard: UTF-8, UTF-16, and UTF-32. For more information, see The Unicode Standard at the Unicode home page.

You can retrieve information about all the encodings available in .NET by calling the Encoding.GetEncodings method. .NET supports the character encoding systems listed in the following table.

ENCODING	CLASS	DESCRIPTION	ADVANTAGES/DISADVANTAGES
ASCII	ASCIIEncoding	Encodes a limited range of characters by using the lower seven bits of a byte.	Because this encoding only supports character values from U+0000 through U+007F, in most cases it is inadequate for internationalized applications.

ENCODING	CLASS	DESCRIPTION	ADVANTAGES/DISADVANTAGES
UTF-7	UTF7Encoding	Represents characters as sequences of 7-bit ASCII characters. Non-ASCII Unicode characters are represented by an escape sequence of ASCII characters.	UTF-7 supports protocols such as email and newsgroup protocols. However, UTF-7 is not particularly secure or robust. In some cases, changing one bit can radically alter the interpretation of an entire UTF-7 string. In other cases, different UTF-7 strings can encode the same text. For sequences that include non-ASCII characters, UTF-7 requires more space than UTF-8, and encoding/decoding is slower. Consequently, you should use UTF-8 instead of UTF-7 if possible.
UTF-8	UTF8Encoding	Represents each Unicode code point as a sequence of one to four bytes.	UTF-8 supports 8-bit data sizes and works well with many existing operating systems. For the ASCII range of characters, UTF-8 is identical to ASCII encoding and allows a broader set of characters. However, for Chinese-Japanese-Korean (CJK) scripts, UTF-8 can require three bytes for each character, and can potentially cause larger data sizes than UTF-16. Note that sometimes the amount of ASCII data, such as HTML tags, justifies the increased size for the CJK range.
UTF-16	UnicodeEncoding	Represents each Unicode code point as a sequence of one or two 16-bit integers. Most common Unicode characters require only one UTF-16 code point, although Unicode supplementary characters (U+10000 and greater) require two UTF-16 surrogate code points. Both little-endian and big-endian byte orders are supported.	UTF-16 encoding is used by the common language runtime to represent Char and String values, and it is used by the Windows operating system to represent `WCHAR` values.

ENCODING	CLASS	DESCRIPTION	ADVANTAGES/DISADVANTAGES
UTF-32	UTF32Encoding	Represents each Unicode code point as a 32-bit integer. Both little-endian and big-endian byte orders are supported.	UTF-32 encoding is used when applications want to avoid the surrogate code point behavior of UTF-16 encoding on operating systems for which encoded space is too important. Single glyphs rendered on a display can still be encoded with more than one UTF-32 character.
ANSI/ISO encodings		Provides support for a variety of code pages. On Windows operating systems, code pages are used to support a specific language or group of languages. For a table that lists the code pages supported by .NET, see the Encoding class. You can retrieve an encoding object for a particular code page by calling the Encoding.GetEncoding(Int32) method.	A code page contains 256 code points and is zero-based. In most code pages, code points 0 through 127 represent the ASCII character set, and code points 128 through 255 differ significantly between code pages. For example, code page 1252 provides the characters for Latin writing systems, including English, German, and French. The last 128 code points in code page 1252 contain the accent characters. Code page 1253 provides character codes that are required in the Greek writing system. The last 128 code points in code page 1253 contain the Greek characters. As a result, an application that relies on ANSI code pages cannot store Greek and German in the same text stream unless it includes an identifier that indicates the referenced code page.

ENCODING	CLASS	DESCRIPTION	ADVANTAGES/DISADVANTAGES
Double-byte character set (DBCS) encodings		Supports languages, such as Chinese, Japanese, and Korean, that contain more than 256 characters. In a DBCS, a pair of code points (a double byte) represents each character. The Encoding.IsSingleByte property returns `false` for DBCS encodings. You can retrieve an encoding object for a particular DBCS by calling the Encoding.GetEncoding(Int32) method.	In a DBCS, a pair of code points (a double byte) represents each character. When an application handles DBCS data, the first byte of a DBCS character (the lead byte) is processed in combination with the trail byte that immediately follows it. Because a single pair of double-byte code points can represent different characters depending on the code page, this scheme still does not allow for the combination of two languages, such as Japanese and Chinese, in the same data stream.

These encodings enable you to work with Unicode characters as well as with encodings that are most commonly used in legacy applications. In addition, you can create a custom encoding by defining a class that derives from Encoding and overriding its members.

Platform Notes: .NET Core

By default, .NET Core does not make available any code page encodings other than code page 28591 and the Unicode encodings, such as UTF-8 and UTF-16. However, you can add the code page encodings found in standard Windows apps that target .NET to your app. For complete information, see the CodePagesEncodingProvider topic.

Selecting an Encoding Class

If you have the opportunity to choose the encoding to be used by your application, you should use a Unicode encoding, preferably either UTF8Encoding or UnicodeEncoding. (.NET also supports a third Unicode encoding, UTF32Encoding.)

If you are planning to use an ASCII encoding (ASCIIEncoding), choose UTF8Encoding instead. The two encodings are identical for the ASCII character set, but UTF8Encoding has the following advantages:

- It can represent every Unicode character, whereas ASCIIEncoding supports only the Unicode character values between U+0000 and U+007F.
- It provides error detection and better security.
- It has been tuned to be as fast as possible and should be faster than any other encoding. Even for content that is entirely ASCII, operations performed with UTF8Encoding are faster than operations performed with ASCIIEncoding.

You should consider using ASCIIEncoding only for legacy applications. However, even for legacy applications, UTF8Encoding might be a better choice for the following reasons (assuming default settings):

- If your application has content that is not strictly ASCII and encodes it with ASCIIEncoding, each non-ASCII character encodes as a question mark (?). If the application then decodes this data, the information is lost.
- If your application has content that is not strictly ASCII and encodes it with UTF8Encoding, the result seems unintelligible if interpreted as ASCII. However, if the application then uses a UTF-8 decoder to decode this

data, the data performs a round trip successfully.

In a web application, characters sent to the client in response to a web request should reflect the encoding used on the client. In most cases, you should set the HttpResponse.ContentEncoding property to the value returned by the HttpRequest.ContentEncoding property to display text in the encoding that the user expects.

Using an Encoding Object

An encoder converts a string of characters (most commonly, Unicode characters) to its numeric (byte) equivalent. For example, you might use an ASCII encoder to convert Unicode characters to ASCII so that they can be displayed at the console. To perform the conversion, you call the Encoding.GetBytes method. If you want to determine how many bytes are needed to store the encoded characters before performing the encoding, you can call the GetByteCount method.

The following example uses a single byte array to encode strings in two separate operations. It maintains an index that indicates the starting position in the byte array for the next set of ASCII-encoded bytes. It calls the ASCIIEncoding.GetByteCount(String) method to ensure that the byte array is large enough to accommodate the encoded string. It then calls the ASCIIEncoding.GetBytes(String, Int32, Int32, Byte[], Int32) method to encode the characters in the string.

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      string[] strings= { "This is the first sentence. ",
                          "This is the second sentence. " };
      Encoding asciiEncoding = Encoding.ASCII;

      // Create array of adequate size.
      byte[] bytes = new byte[49];
      // Create index for current position of array.
      int index = 0;

      Console.WriteLine("Strings to encode:");
      foreach (var stringValue in strings) {
         Console.WriteLine("   {0}", stringValue);

         int count = asciiEncoding.GetByteCount(stringValue);
         if (count + index >=  bytes.Length)
            Array.Resize(ref bytes, bytes.Length + 50);

         int written = asciiEncoding.GetBytes(stringValue, 0,
                                              stringValue.Length,
                                              bytes, index);

         index = index + written;
      }
      Console.WriteLine("\nEncoded bytes:");
      Console.WriteLine("{0}", ShowByteValues(bytes, index));
      Console.WriteLine();

      // Decode Unicode byte array to a string.
      string newString = asciiEncoding.GetString(bytes, 0, index);
      Console.WriteLine("Decoded: {0}", newString);
   }

   private static string ShowByteValues(byte[] bytes, int last )
   {
      string returnString = "   ";
      for (int ctr = 0; ctr <= last - 1; ctr++) {
         if (ctr % 20 == 0)
            returnString += "\n   ";
         returnString += String.Format("{0:X2} ", bytes[ctr]);
      }
      return returnString;
   }
}
// The example displays the following output:
//       Strings to encode:
//          This is the first sentence.
//          This is the second sentence.
//
//       Encoded bytes:
//
//          54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
//          6E 74 65 6E 63 65 2E 20 54 68 69 73 20 69 73 20 74 68 65 20
//          73 65 63 6F 6E 64 20 73 65 6E 74 65 6E 63 65 2E 20
//
//       Decoded: This is the first sentence. This is the second sentence.
```

```
Imports System.Text

Module Example
   Public Sub Main()
      Dim strings() As String = { "This is the first sentence. ",
                                  "This is the second sentence. " }
      Dim asciiEncoding As Encoding = Encoding.ASCII

      ' Create array of adequate size.
      Dim bytes(50) As Byte
      ' Create index for current position of array.
      Dim index As Integer = 0

      Console.WriteLine("Strings to encode:")
      For Each stringValue In strings
         Console.WriteLine("   {0}", stringValue)

         Dim count As Integer = asciiEncoding.GetByteCount(stringValue)
         If count + index >=  bytes.Length Then
            Array.Resize(bytes, bytes.Length + 50)
         End If
         Dim written As Integer = asciiEncoding.GetBytes(stringValue, 0,
                                                         stringValue.Length,
                                                         bytes, index)

         index = index + written
      Next
      Console.WriteLine()
      Console.WriteLine("Encoded bytes:")
      Console.WriteLine("{0}", ShowByteValues(bytes, index))
      Console.WriteLine()

      ' Decode Unicode byte array to a string.
      Dim newString As String = asciiEncoding.GetString(bytes, 0, index)
      Console.WriteLine("Decoded: {0}", newString)
   End Sub

   Private Function ShowByteValues(bytes As Byte(), last As Integer) As String
      Dim returnString As String = "   "
      For ctr As Integer = 0 To last - 1
         If ctr Mod 20 = 0 Then returnString += vbCrLf + "   "
         returnString += String.Format("{0:X2} ", bytes(ctr))
      Next
      Return returnString
   End Function
End Module
' The example displays the following output:
'       Strings to encode:
'          This is the first sentence.
'          This is the second sentence.
'
'       Encoded bytes:
'
'          54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
'          6E 74 65 6E 63 65 2E 20 54 68 69 73 20 69 73 20 74 68 65 20
'          73 65 63 6F 6E 64 20 73 65 6E 74 65 6E 63 65 2E 20
'
'       Decoded: This is the first sentence. This is the second sentence.
```

A decoder converts a byte array that reflects a particular character encoding into a set of characters, either in a character array or in a string. To decode a byte array into a character array, you call the Encoding.GetChars method. To decode a byte array into a string, you call the GetString method. If you want to determine how many characters are needed to store the decoded bytes before performing the decoding, you can call the GetCharCount method.

The following example encodes three strings and then decodes them into a single array of characters. It maintains an index that indicates the starting position in the character array for the next set of decoded characters. It calls the GetCharCount method to ensure that the character array is large enough to accommodate all the decoded characters. It then calls the ASCIIEncoding.GetChars(Byte[], Int32, Int32, Char[], Int32) method to decode the byte array.

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      string[] strings = { "This is the first sentence. ",
                           "This is the second sentence. ",
                           "This is the third sentence. " };
      Encoding asciiEncoding = Encoding.ASCII;
      // Array to hold encoded bytes.
      byte[] bytes;
      // Array to hold decoded characters.
      char[] chars = new char[50];
      // Create index for current position of character array.
      int index = 0;

      foreach (var stringValue in strings) {
         Console.WriteLine("String to Encode: {0}", stringValue);
         // Encode the string to a byte array.
         bytes = asciiEncoding.GetBytes(stringValue);
         // Display the encoded bytes.
         Console.Write("Encoded bytes: ");
         for (int ctr = 0; ctr < bytes.Length; ctr++)
            Console.Write(" {0}{1:X2}",
                          ctr % 20 == 0 ? Environment.NewLine : "",
                          bytes[ctr]);
         Console.WriteLine();

         // Decode the bytes to a single character array.
         int count = asciiEncoding.GetCharCount(bytes);
         if (count + index >=  chars.Length)
            Array.Resize(ref chars, chars.Length + 50);

         int written = asciiEncoding.GetChars(bytes, 0,
                                              bytes.Length,
                                              chars, index);
         index = index + written;
         Console.WriteLine();
      }

      // Instantiate a single string containing the characters.
      string decodedString = new string(chars, 0, index - 1);
      Console.WriteLine("Decoded string: ");
      Console.WriteLine(decodedString);
   }
}
// The example displays the following output:
//    String to Encode: This is the first sentence.
//    Encoded bytes:
//    54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
//    6E 74 65 6E 63 65 2E 20
//
//    String to Encode: This is the second sentence.
//    Encoded bytes:
//    54 68 69 73 20 69 73 20 74 68 65 20 73 65 63 6F 6E 64 20 73
//    65 6E 74 65 6E 63 65 2E 20
//
//    String to Encode: This is the third sentence.
//    Encoded bytes:
//    54 68 69 73 20 69 73 20 74 68 65 20 74 68 69 72 64 20 73 65
//    6E 74 65 6E 63 65 2E 20
//
//    Decoded string:
//    This is the first sentence. This is the second sentence. This is the third sentence.
```

```vb
Imports System.Text

Module Example
   Public Sub Main()
      Dim strings() As String = { "This is the first sentence. ",
                                  "This is the second sentence. ",
                                  "This is the third sentence. " }
      Dim asciiEncoding As Encoding = Encoding.ASCII
      ' Array to hold encoded bytes.
      Dim bytes() As Byte
      ' Array to hold decoded characters.
      Dim chars(50) As Char
      ' Create index for current position of character array.
      Dim index As Integer

      For Each stringValue In strings
         Console.WriteLine("String to Encode: {0}", stringValue)
         ' Encode the string to a byte array.
         bytes = asciiEncoding.GetBytes(stringValue)
         ' Display the encoded bytes.
         Console.Write("Encoded bytes: ")
         For ctr As Integer = 0 To bytes.Length - 1
            Console.Write(" {0}{1:X2}", If(ctr Mod 20 = 0, vbCrLf, ""),
                                        bytes(ctr))
         Next
         Console.WriteLine()

         ' Decode the bytes to a single character array.
         Dim count As Integer = asciiEncoding.GetCharCount(bytes)
         If count + index >=  chars.Length Then
            Array.Resize(chars, chars.Length + 50)
         End If
         Dim written As Integer = asciiEncoding.GetChars(bytes, 0,
                                                         bytes.Length,
                                                         chars, index)
         index = index + written
         Console.WriteLine()
      Next

      ' Instantiate a single string containing the characters.
      Dim decodedString As New String(chars, 0, index - 1)
      Console.WriteLine("Decoded string: ")
      Console.WriteLine(decodedString)
   End Sub
End Module
' The example displays the following output:
'    String to Encode: This is the first sentence.
'    Encoded bytes:
'    54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
'    6E 74 65 6E 63 65 2E 20
'
'    String to Encode: This is the second sentence.
'    Encoded bytes:
'    54 68 69 73 20 69 73 20 74 68 65 20 73 65 63 6F 6E 64 20 73
'    65 6E 74 65 6E 63 65 2E 20
'
'    String to Encode: This is the third sentence.
'    Encoded bytes:
'    54 68 69 73 20 69 73 20 74 68 65 20 74 68 69 72 64 20 73 65
'    6E 74 65 6E 63 65 2E 20
'
'    Decoded string:
'    This is the first sentence. This is the second sentence. This is the third sentence.
```

The encoding and decoding methods of a class derived from Encoding are designed to work on a complete set of data; that is, all the data to be encoded or decoded is supplied in a single method call. However, in some cases, data is available in a stream, and the data to be encoded or decoded may be available only from separate read operations. This requires the encoding or decoding operation to remember any saved state from its previous invocation. Methods of classes derived from Encoder and Decoder are able to handle encoding and decoding operations that span multiple method calls.

An Encoder object for a particular encoding is available from that encoding's Encoding.GetEncoder property. A Decoder object for a particular encoding is available from that encoding's Encoding.GetDecoder property. For decoding operations, note that classes derived from Decoder include a Decoder.GetChars method, but they do not have a method that corresponds to Encoding.GetString.

The following example illustrates the difference between using the Encoding.GetChars and Decoder.GetChars methods for decoding a Unicode byte array. The example encodes a string that contains some Unicode characters to a file, and then uses the two decoding methods to decode them ten bytes at a time. Because a surrogate pair occurs in the tenth and eleventh bytes, it is decoded in separate method calls. As the output shows, the Encoding.GetChars method is not able to correctly decode the bytes and instead replaces them with U+FFFD (REPLACEMENT CHARACTER). On the other hand, the Decoder.GetChars method is able to successfully decode the byte array to get the original string.

```
using System;
using System.IO;
using System.Text;

public class Example
{
   public static void Main()
   {
      // Use default replacement fallback for invalid encoding.
      UnicodeEncoding enc = new UnicodeEncoding(true, false, false);

      // Define a string with various Unicode characters.
      string str1 = "AB YZ 19 \uD800\udc05 \u00e4";
      str1 += "Unicode characters. \u00a9 \u010C s \u0062\u0308";
      Console.WriteLine("Created original string...\n");

      // Convert string to byte array.
      byte[] bytes = enc.GetBytes(str1);

      FileStream fs = File.Create(@".\characters.bin");
      BinaryWriter bw = new BinaryWriter(fs);
      bw.Write(bytes);
      bw.Close();

      // Read bytes from file.
      FileStream fsIn = File.OpenRead(@".\characters.bin");
      BinaryReader br = new BinaryReader(fsIn);

      const int count = 10;            // Number of bytes to read at a time.
      byte[] bytesRead = new byte[10]; // Buffer (byte array).
      int read;                        // Number of bytes actually read.
      string str2 = String.Empty;      // Decoded string.

      // Try using Encoding object for all operations.
      do {
         read = br.Read(bytesRead, 0, count);
         str2 += enc.GetString(bytesRead, 0, read);
      } while (read == count);
      br.Close();
      Console.WriteLine("Decoded string using UnicodeEncoding.GetString()...");
      CompareForEquality(str1, str2);
      Console.WriteLine();

      // Use Decoder for all operations.
      fsIn = File.OpenRead(@".\characters.bin");
      br = new BinaryReader(fsIn);
```

```csharp
      Decoder decoder = enc.GetDecoder();
      char[] chars = new char[50];
      int index = 0;                   // Next character to write in array.
      int written = 0;                 // Number of chars written to array.
      do {
         read = br.Read(bytesRead, 0, count);
         if (index + decoder.GetCharCount(bytesRead, 0, read) - 1 >= chars.Length)
            Array.Resize(ref chars, chars.Length + 50);

         written = decoder.GetChars(bytesRead, 0, read, chars, index);
         index += written;
      } while (read == count);
      br.Close();
      // Instantiate a string with the decoded characters.
      string str3 = new String(chars, 0, index);
      Console.WriteLine("Decoded string using UnicodeEncoding.Decoder.GetString()...");
      CompareForEquality(str1, str3);
   }

   private static void CompareForEquality(string original, string decoded)
   {
      bool result = original.Equals(decoded);
      Console.WriteLine("original = decoded: {0}",
                        original.Equals(decoded, StringComparison.Ordinal));
      if (! result) {
         Console.WriteLine("Code points in original string:");
         foreach (var ch in original)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
         Console.WriteLine();

         Console.WriteLine("Code points in decoded string:");
         foreach (var ch in decoded)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//    Created original string...
//
//    Decoded string using UnicodeEncoding.GetString()...
//    original = decoded: False
//    Code points in original string:
//    0041 0042 0020 0059 005A 0020 0031 0039 0020 D800 DC05 0020 00E4 0055 006E 0069 0063 006F
//    0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E 0020 00A9 0020 010C
//    0020 0073 0020 0062 0308
//    Code points in decoded string:
//    0041 0042 0020 0059 005A 0020 0031 0039 0020 FFFD FFFD 0020 00E4 0055 006E 0069 0063 006F
//    0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E 0020 00A9 0020 010C
//    0020 0073 0020 0062 0308
//
//    Decoded string using UnicodeEncoding.Decoder.GetString()...
//    original = decoded: True
```

```vb
Imports System.IO
Imports System.Text

Module Example
   Public Sub Main()
      ' Use default replacement fallback for invalid encoding.
      Dim enc As New UnicodeEncoding(True, False, False)

      ' Define a string with various Unicode characters.
      Dim str1 As String = String.Format("AB YZ 19 {0}{1} {2}",
                                         ChrW(&hD800), ChrW(&hDC05), ChrW(&h00e4))
      str1 += String.Format("Unicode characters. {0} {1} s {2}{3}",
                            ChrW(&h00a9), ChrW(&h010C), ChrW(&h0062), ChrW(&h0308))
```

```
    Console.WriteLine("Created original string...")
    Console.WriteLine()

    ' Convert string to byte array.
    Dim bytes() As Byte = enc.GetBytes(str1)

    Dim fs As FileStream = File.Create(".\characters.bin")
    Dim bw As New BinaryWriter(fs)
    bw.Write(bytes)
    bw.Close()

    ' Read bytes from file.
    Dim fsIn As FileStream = File.OpenRead(".\characters.bin")
    Dim br As New BinaryReader(fsIn)

    Const count As Integer = 10      ' Number of bytes to read at a time.
    Dim bytesRead(9) As Byte         ' Buffer (byte array).
    Dim read As Integer              ' Number of bytes actually read.
    Dim str2 As String = ""          ' Decoded string.

    ' Try using Encoding object for all operations.
    Do
       read = br.Read(bytesRead, 0, count)
       str2 += enc.GetString(bytesRead, 0, read)
    Loop While read = count
    br.Close()
    Console.WriteLine("Decoded string using UnicodeEncoding.GetString()...")
    CompareForEquality(str1, str2)
    Console.WriteLine()

    ' Use Decoder for all operations.
    fsIn = File.OpenRead(".\characters.bin")
    br = New BinaryReader(fsIn)
    Dim decoder As Decoder = enc.GetDecoder()
    Dim chars(50) As Char
    Dim index As Integer = 0         ' Next character to write in array.
    Dim written As Integer = 0       ' Number of chars written to array.
    Do
       read = br.Read(bytesRead, 0, count)
       If index + decoder.GetCharCount(bytesRead, 0, read) - 1 >= chars.Length Then
          Array.Resize(chars, chars.Length + 50)
       End If
       written = decoder.GetChars(bytesRead, 0, read, chars, index)
       index += written
    Loop While read = count
    br.Close()
    ' Instantiate a string with the decoded characters.
    Dim str3 As New String(chars, 0, index)
    Console.WriteLine("Decoded string using UnicodeEncoding.Decoder.GetString()...")
    CompareForEquality(str1, str3)
 End Sub

 Private Sub CompareForEquality(original As String, decoded As String)
    Dim result As Boolean = original.Equals(decoded)
    Console.WriteLine("original = decoded: {0}",
                      original.Equals(decoded, StringComparison.Ordinal))
    If Not result Then
       Console.WriteLine("Code points in original string:")
       For Each ch In original
          Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
       Next
       Console.WriteLine()

       Console.WriteLine("Code points in decoded string:")
       For Each ch In decoded
          Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
       Next
       Console.WriteLine()
    End If
```

```
    End Sub
End Module
' The example displays the following output:
'    Created original string...
'
'    Decoded string using UnicodeEncoding.GetString()...
'    original = decoded: False
'    Code points in original string:
'    0041 0042 0020 0059 005A 0020 0031 0039 0020 D800 DC05 0020 00E4 0055 006E 0069 0063 006F
'    0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E 0020 00A9 0020 010C
'    0020 0073 0020 0062 0308
'    Code points in decoded string:
'    0041 0042 0020 0059 005A 0020 0031 0039 0020 FFFD FFFD 0020 00E4 0055 006E 0069 0063 006F
'    0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E 0020 00A9 0020 010C
'    0020 0073 0020 0062 0308
'
'    Decoded string using UnicodeEncoding.Decoder.GetString()...
'    original = decoded: True
```

Choosing a Fallback Strategy

When a method tries to encode or decode a character but no mapping exists, it must implement a fallback strategy that determines how the failed mapping should be handled. There are three types of fallback strategies:

- Best-fit fallback
- Replacement fallback
- Exception fallback

> **IMPORTANT**
>
> The most common problems in encoding operations occur when a Unicode character cannot be mapped to a particular code page encoding. The most common problems in decoding operations occur when invalid byte sequences cannot be translated into valid Unicode characters. For these reasons, you should know which fallback strategy a particular encoding object uses. Whenever possible, you should specify the fallback strategy used by an encoding object when you instantiate the object.

Best-Fit Fallback

When a character does not have an exact match in the target encoding, the encoder can try to map it to a similar character. (Best-fit fallback is mostly an encoding rather than a decoding issue. There are very few code pages that contain characters that cannot be successfully mapped to Unicode.) Best-fit fallback is the default for code page and double-byte character set encodings that are retrieved by the Encoding.GetEncoding(Int32) and Encoding.GetEncoding(String) overloads.

> **NOTE**
>
> In theory, the Unicode encoding classes provided in .NET (UTF8Encoding, UnicodeEncoding, and UTF32Encoding) support every character in every character set, so they can be used to eliminate best-fit fallback issues.

Best-fit strategies vary for different code pages. For example, for some code pages, full-width Latin characters map to the more common half-width Latin characters. For other code pages, this mapping is not made. Even under an aggressive best-fit strategy, there is no imaginable fit for some characters in some encodings. For example, a Chinese ideograph has no reasonable mapping to code page 1252. In this case, a replacement string is used. By default, this string is just a single QUESTION MARK (U+003F).

> **NOTE**
>
> Best-fit strategies are not documented in detail. However, several code pages are documented at the Unicode Consortium's website. Please review the **readme.txt** file in that folder for a description of how to interpret the mapping files.

The following example uses code page 1252 (the Windows code page for Western European languages) to illustrate best-fit mapping and its drawbacks. The Encoding.GetEncoding(Int32) method is used to retrieve an encoding object for code page 1252. By default, it uses a best-fit mapping for Unicode characters that it does not support. The example instantiates a string that contains three non-ASCII characters - CIRCLED LATIN CAPITAL LETTER S (U+24C8), SUPERSCRIPT FIVE (U+2075), and INFINITY (U+221E) - separated by spaces. As the output from the example shows, when the string is encoded, the three original non-space characters are replaced by QUESTION MARK (U+003F), DIGIT FIVE (U+0035), and DIGIT EIGHT (U+0038). DIGIT EIGHT is a particularly poor replacement for the unsupported INFINITY character, and QUESTION MARK indicates that no mapping was available for the original character.

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      // Get an encoding for code page 1252 (Western Europe character set).
      Encoding cp1252 = Encoding.GetEncoding(1252);

      // Define and display a string.
      string str = "\u24c8 \u2075 \u221e";
      Console.WriteLine("Original string: " + str);
      Console.Write("Code points in string: ");
      foreach (var ch in str)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine("\n");

      // Encode a Unicode string.
      Byte[] bytes = cp1252.GetBytes(str);
      Console.Write("Encoded bytes: ");
      foreach (byte byt in bytes)
         Console.Write("{0:X2} ", byt);
      Console.WriteLine("\n");

      // Decode the string.
      string str2 = cp1252.GetString(bytes);
      Console.WriteLine("String round-tripped: {0}", str.Equals(str2));
      if (! str.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
      }
   }
}
// The example displays the following output:
//       Original string: Ⓢ ⁵ ∞
//       Code points in string: 24C8 0020 2075 0020 221E
//
//       Encoded bytes: 3F 20 35 20 38
//
//       String round-tripped: False
//       ? 5 8
//       003F 0020 0035 0020 0038
```

```vb
Imports System.Text

Module Example
   Public Sub Main()
      ' Get an encoding for code page 1252 (Western Europe character set).
      Dim cp1252 As Encoding = Encoding.GetEncoding(1252)

      ' Define and display a string.
      Dim str As String = String.Format("{0} {1} {2}", ChrW(&h24c8), ChrW(&H2075), ChrW(&h221E))
      Console.WriteLine("Original string: " + str)
      Console.Write("Code points in string: ")
      For Each ch In str
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
      Next
      Console.WriteLine()
      Console.WriteLine()

      ' Encode a Unicode string.
      Dim bytes() As Byte = cp1252.GetBytes(str)
      Console.Write("Encoded bytes: ")
      For Each byt In bytes
         Console.Write("{0:X2} ", byt)
      Next
      Console.WriteLine()
      Console.WriteLine()

      ' Decode the string.
      Dim str2 As String = cp1252.GetString(bytes)
      Console.WriteLine("String round-tripped: {0}", str.Equals(str2))
      If Not str.Equals(str2) Then
         Console.WriteLine(str2)
         For Each ch In str2
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
         Next
      End If
   End Sub
End Module
' The example displays the following output:
'       Original string: Ⓢ ⁵ ∞
'       Code points in string: 24C8 0020 2075 0020 221E
'
'       Encoded bytes: 3F 20 35 20 38
'
'       String round-tripped: False
'       ? 5 8
'       003F 0020 0035 0020 0038
```

Best-fit mapping is the default behavior for an Encoding object that encodes Unicode data into code page data, and there are legacy applications that rely on this behavior. However, most new applications should avoid best-fit behavior for security reasons. For example, applications should not put a domain name through a best-fit encoding.

> **NOTE**
>
> You can also implement a custom best-fit fallback mapping for an encoding. For more information, see the Implementing a Custom Fallback Strategy section.

If best-fit fallback is the default for an encoding object, you can choose another fallback strategy when you retrieve an Encoding object by calling the Encoding.GetEncoding(Int32, EncoderFallback, DecoderFallback) or Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) overload. The following section includes an example that replaces each character that cannot be mapped to code page 1252 with an asterisk (*).

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding cp1252r = Encoding.GetEncoding(1252,
                                  new EncoderReplacementFallback("*"),
                                  new DecoderReplacementFallback("*"));

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      foreach (var ch in str1)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine();

      byte[] bytes = cp1252r.GetBytes(str1);
      string str2 = cp1252r.GetString(bytes);
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
      if (! str1.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//       Round-trip: False
//       * * *
//       002A 0020 002A 0020 002A
```

```vb
Imports System.Text

Module Example
   Public Sub Main()
      Dim cp1252r As Encoding = Encoding.GetEncoding(1252,
                                        New EncoderReplacementFallback("*"),
                                        New DecoderReplacementFallback("*"))

      Dim str1 As String = String.Format("{0} {1} {2}", ChrW(&h24C8), ChrW(&h2075), ChrW(&h221E))
      Console.WriteLine(str1)
      For Each ch In str1
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
      Next
      Console.WriteLine()

      Dim bytes() As Byte = cp1252r.GetBytes(str1)
      Dim str2 As String = cp1252r.GetString(bytes)
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2))
      If Not str1.Equals(str2) Then
         Console.WriteLine(str2)
         For Each ch In str2
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
         Next
         Console.WriteLine()
      End If
   End Sub
End Module
' The example displays the following output:
'       Ⓢ ⁵ ∞
'       24C8 0020 2075 0020 221E
'       Round-trip: False
'       * * *
'       002A 0020 002A 0020 002A
```

Replacement Fallback

When a character does not have an exact match in the target scheme, but there is no appropriate character that it can be mapped to, the application can specify a replacement character or string. This is the default behavior for the Unicode decoder, which replaces any two-byte sequence that it cannot decode with REPLACEMENT_CHARACTER (U+FFFD). It is also the default behavior of the ASCIIEncoding class, which replaces each character that it cannot encode or decode with a question mark. The following example illustrates character replacement for the Unicode string from the previous example. As the output shows, each character that cannot be decoded into an ASCII byte value is replaced by 0x3F, which is the ASCII code for a question mark.

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding enc = Encoding.ASCII;

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      foreach (var ch in str1)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine("\n");

      // Encode the original string using the ASCII encoder.
      byte[] bytes = enc.GetBytes(str1);
      Console.Write("Encoded bytes: ");
      foreach (var byt in bytes)
         Console.Write("{0:X2} ", byt);
      Console.WriteLine("\n");

      // Decode the ASCII bytes.
      string str2 = enc.GetString(bytes);
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
      if (! str1.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//
//       Encoded bytes: 3F 20 3F 20 3F
//
//       Round-trip: False
//       ? ? ?
//       003F 0020 003F 0020 003F
```

```
Imports System.Text

Module Example
   Public Sub Main()
      Dim enc As Encoding = Encoding.Ascii

      Dim str1 As String = String.Format("{0} {1} {2}", ChrW(&h24C8), ChrW(&h2075), ChrW(&h221E))
      Console.WriteLine(str1)
      For Each ch In str1
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
      Next
      Console.WriteLine()
      Console.WriteLine()

      ' Encode the original string using the ASCII encoder.
      Dim bytes() As Byte = enc.GetBytes(str1)
      Console.Write("Encoded bytes: ")
      For Each byt In bytes
         Console.Write("{0:X2} ", byt)
      Next
      Console.WriteLine()
      Console.WriteLine()

      ' Decode the ASCII bytes.
      Dim str2 As String = enc.GetString(bytes)
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2))
      If Not str1.Equals(str2) Then
         Console.WriteLine(str2)
         For Each ch In str2
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
         Next
         Console.WriteLine()
      End If
   End Sub
End Module
' The example displays the following output:
'       Ⓢ ⁵ ∞
'       24C8 0020 2075 0020 221E
'
'       Encoded bytes: 3F 20 3F 20 3F
'
'       Round-trip: False
'       ? ? ?
'       003F 0020 003F 0020 003F
```

.NET includes the EncoderReplacementFallback and DecoderReplacementFallback classes, which substitute a replacement string if a character does not map exactly in an encoding or decoding operation. By default, this replacement string is a question mark, but you can call a class constructor overload to choose a different string. Typically, the replacement string is a single character, although this is not a requirement. The following example changes the behavior of the code page 1252 encoder by instantiating an EncoderReplacementFallback object that uses an asterisk (*) as a replacement string.

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding cp1252r = Encoding.GetEncoding(1252,
                                  new EncoderReplacementFallback("*"),
                                  new DecoderReplacementFallback("*"));

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      foreach (var ch in str1)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine();

      byte[] bytes = cp1252r.GetBytes(str1);
      string str2 = cp1252r.GetString(bytes);
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
      if (! str1.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//       Round-trip: False
//       * * *
//       002A 0020 002A 0020 002A
```

```
Imports System.Text

Module Example
   Public Sub Main()
      Dim cp1252r As Encoding = Encoding.GetEncoding(1252,
                                    New EncoderReplacementFallback("*"),
                                    New DecoderReplacementFallback("*"))

      Dim str1 As String = String.Format("{0} {1} {2}", ChrW(&h24C8), ChrW(&h2075), ChrW(&h221E))
      Console.WriteLine(str1)
      For Each ch In str1
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
      Next
      Console.WriteLine()

      Dim bytes() As Byte = cp1252r.GetBytes(str1)
      Dim str2 As String = cp1252r.GetString(bytes)
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2))
      If Not str1.Equals(str2) Then
         Console.WriteLine(str2)
         For Each ch In str2
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
         Next
         Console.WriteLine()
      End If
   End Sub
End Module
' The example displays the following output:
'       Ⓢ ⁵ ∞
'       24C8 0020 2075 0020 221E
'       Round-trip: False
'       * * *
'       002A 0020 002A 0020 002A
```

NOTE

You can also implement a replacement class for an encoding. For more information, see the Implementing a Custom Fallback Strategy section.

In addition to QUESTION MARK (U+003F), the Unicode REPLACEMENT CHARACTER (U+FFFD) is commonly used as a replacement string, particularly when decoding byte sequences that cannot be successfully translated into Unicode characters. However, you are free to choose any replacement string, and it can contain multiple characters.

Exception Fallback

Instead of providing a best-fit fallback or a replacement string, an encoder can throw an EncoderFallbackException if it is unable to encode a set of characters, and a decoder can throw a DecoderFallbackException if it is unable to decode a byte array. To throw an exception in encoding and decoding operations, you supply an EncoderExceptionFallback object and a DecoderExceptionFallback object, respectively, to the Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) method. The following example illustrates exception fallback with the ASCIIEncoding class.

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding enc = Encoding.GetEncoding("us-ascii",
                                          new EncoderExceptionFallback(),
                                          new DecoderExceptionFallback());
```

```
      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      foreach (var ch in str1)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine("\n");

      // Encode the original string using the ASCII encoder.
      byte[] bytes = {};
      try {
         bytes = enc.GetBytes(str1);
         Console.Write("Encoded bytes: ");
         foreach (var byt in bytes)
            Console.Write("{0:X2} ", byt);

         Console.WriteLine();
      }
      catch (EncoderFallbackException e) {
         Console.Write("Exception: ");
         if (e.IsUnknownSurrogate())
            Console.WriteLine("Unable to encode surrogate pair 0x{0:X4} 0x{1:X3} at index {2}.",
                              Convert.ToUInt16(e.CharUnknownHigh),
                              Convert.ToUInt16(e.CharUnknownLow),
                              e.Index);
         else
            Console.WriteLine("Unable to encode 0x{0:X4} at index {1}.",
                              Convert.ToUInt16(e.CharUnknown),
                              e.Index);
         return;
      }
      Console.WriteLine();

      // Decode the ASCII bytes.
      try {
         string str2 = enc.GetString(bytes);
         Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
         if (! str1.Equals(str2)) {
            Console.WriteLine(str2);
            foreach (var ch in str2)
               Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

            Console.WriteLine();
         }
      }
      catch (DecoderFallbackException e) {
         Console.Write("Unable to decode byte(s) ");
         foreach (byte unknown in e.BytesUnknown)
            Console.Write("0x{0:X2} ");

         Console.WriteLine("at index {0}", e.Index);
      }
   }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//
//       Exception: Unable to encode 0x24C8 at index 0.
```

```vb
Imports System.Text

Module Example
   Public Sub Main()
      Dim enc As Encoding = Encoding.GetEncoding("us-ascii",
                                                 New EncoderExceptionFallback(),
                                                 New DecoderExceptionFallback())

      Dim str1 As String = String.Format("{0} {1} {2}", ChrW(&h24C8), ChrW(&h2075), ChrW(&h221E))
      Console.WriteLine(str1)
      For Each ch In str1
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
      Next
      Console.WriteLine()
      Console.WriteLine()

      ' Encode the original string using the ASCII encoder.
      Dim bytes() As Byte = {}
      Try
         bytes = enc.GetBytes(str1)
         Console.Write("Encoded bytes: ")
         For Each byt In bytes
            Console.Write("{0:X2} ", byt)
         Next
         Console.WriteLine()
      Catch e As EncoderFallbackException
         Console.Write("Exception: ")
         If e.IsUnknownSurrogate() Then
            Console.WriteLine("Unable to encode surrogate pair 0x{0:X4} 0x{1:X3} at index {2}.",
                              Convert.ToUInt16(e.CharUnknownHigh),
                              Convert.ToUInt16(e.CharUnknownLow),
                              e.Index)
         Else
            Console.WriteLine("Unable to encode 0x{0:X4} at index {1}.",
                              Convert.ToUInt16(e.CharUnknown),
                              e.Index)
         End If
         Exit Sub
      End Try
      Console.WriteLine()

      ' Decode the ASCII bytes.
      Try
         Dim str2 As String = enc.GetString(bytes)
         Console.WriteLine("Round-trip: {0}", str1.Equals(str2))
         If Not str1.Equals(str2) Then
            Console.WriteLine(str2)
            For Each ch In str2
               Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
            Next
            Console.WriteLine()
         End If
      Catch e As DecoderFallbackException
         Console.Write("Unable to decode byte(s) ")
         For Each unknown As Byte In e.BytesUnknown
            Console.Write("0x{0:X2} ")
         Next
         Console.WriteLine("at index {0}", e.Index)
      End Try
   End Sub
End Module
' The example displays the following output:
'       Ⓢ ⁵ ∞
'       24C8 0020 2075 0020 221E
'
'       Exception: Unable to encode 0x24C8 at index 0.
```

NOTE

You can also implement a custom exception handler for an encoding operation. For more information, see the Implementing a Custom Fallback Strategy section.

The EncoderFallbackException and DecoderFallbackException objects provide the following information about the condition that caused the exception:

- The EncoderFallbackException object includes an IsUnknownSurrogate method, which indicates whether the character or characters that cannot be encoded represent an unknown surrogate pair (in which case, the method returns `true`) or an unknown single character (in which case, the method returns `false`). The characters in the surrogate pair are available from the EncoderFallbackException.CharUnknownHigh and EncoderFallbackException.CharUnknownLow properties. The unknown single character is available from the EncoderFallbackException.CharUnknown property. The EncoderFallbackException.Index property indicates the position in the string at which the first character that could not be encoded was found.
- The DecoderFallbackException object includes a BytesUnknown property that returns an array of bytes that cannot be decoded. The DecoderFallbackException.Index property indicates the starting position of the unknown bytes.

Although the EncoderFallbackException and DecoderFallbackException objects provide adequate diagnostic information about the exception, they do not provide access to the encoding or decoding buffer. Therefore, they do not allow invalid data to be replaced or corrected within the encoding or decoding method.

Implementing a Custom Fallback Strategy

In addition to the best-fit mapping that is implemented internally by code pages, .NET includes the following classes for implementing a fallback strategy:

- Use EncoderReplacementFallback and EncoderReplacementFallbackBuffer to replace characters in encoding operations.
- Use DecoderReplacementFallback and DecoderReplacementFallbackBuffer to replace characters in decoding operations.
- Use EncoderExceptionFallback and EncoderExceptionFallbackBuffer to throw an EncoderFallbackException when a character cannot be encoded.
- Use DecoderExceptionFallback and DecoderExceptionFallbackBuffer to throw a DecoderFallbackException when a character cannot be decoded.

In addition, you can implement a custom solution that uses best-fit fallback, replacement fallback, or exception fallback, by following these steps:

1. Derive a class from EncoderFallback for encoding operations, and from DecoderFallback for decoding operations.
2. Derive a class from EncoderFallbackBuffer for encoding operations, and from DecoderFallbackBuffer for decoding operations.
3. For exception fallback, if the predefined EncoderFallbackException and DecoderFallbackException classes do not meet your needs, derive a class from an exception object such as Exception or ArgumentException.

Deriving from EncoderFallback or DecoderFallback

To implement a custom fallback solution, you must create a class that inherits from EncoderFallback for encoding operations, and from DecoderFallback for decoding operations. Instances of these classes are passed to the

Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) method and serve as the intermediary between the encoding class and the fallback implementation.

When you create a custom fallback solution for an encoder or decoder, you must implement the following members:

- The EncoderFallback.MaxCharCount or DecoderFallback.MaxCharCount property, which returns the maximum possible number of characters that the best-fit, replacement, or exception fallback can return to replace a single character. For a custom exception fallback, its value is zero.

- The EncoderFallback.CreateFallbackBuffer or DecoderFallback.CreateFallbackBuffer method, which returns your custom EncoderFallbackBuffer or DecoderFallbackBuffer implementation. The method is called by the encoder when it encounters the first character that it is unable to successfully encode, or by the decoder when it encounters the first byte that it is unable to successfully decode.

Deriving from EncoderFallbackBuffer or DecoderFallbackBuffer

To implement a custom fallback solution, you must also create a class that inherits from EncoderFallbackBuffer for encoding operations, and from DecoderFallbackBuffer for decoding operations. Instances of these classes are returned by the CreateFallbackBuffer method of the EncoderFallback and DecoderFallback classes. The EncoderFallback.CreateFallbackBuffer method is called by the encoder when it encounters the first character that it is not able to encode, and the DecoderFallback.CreateFallbackBuffer method is called by the decoder when it encounters one or more bytes that it is not able to decode. The EncoderFallbackBuffer and DecoderFallbackBuffer classes provide the fallback implementation. Each instance represents a buffer that contains the fallback characters that will replace the character that cannot be encoded or the byte sequence that cannot be decoded.

When you create a custom fallback solution for an encoder or decoder, you must implement the following members:

- The EncoderFallbackBuffer.Fallback or DecoderFallbackBuffer.Fallback method. EncoderFallbackBuffer.Fallback is called by the encoder to provide the fallback buffer with information about the character that it cannot encode. Because the character to be encoded may be a surrogate pair, this method is overloaded. One overload is passed the character to be encoded and its index in the string. The second overload is passed the high and low surrogate along with its index in the string. The DecoderFallbackBuffer.Fallback method is called by the decoder to provide the fallback buffer with information about the bytes that it cannot decode. This method is passed an array of bytes that it cannot decode, along with the index of the first byte. The fallback method should return `true` if the fallback buffer can supply a best-fit or replacement character or characters; otherwise, it should return `false`. For an exception fallback, the fallback method should throw an exception.

- The EncoderFallbackBuffer.GetNextChar or DecoderFallbackBuffer.GetNextChar method, which is called repeatedly by the encoder or decoder to get the next character from the fallback buffer. When all fallback characters have been returned, the method should return U+0000.

- The EncoderFallbackBuffer.Remaining or DecoderFallbackBuffer.Remaining property, which returns the number of characters remaining in the fallback buffer.

- The EncoderFallbackBuffer.MovePrevious or DecoderFallbackBuffer.MovePrevious method, which moves the current position in the fallback buffer to the previous character.

- The EncoderFallbackBuffer.Reset or DecoderFallbackBuffer.Reset method, which reinitializes the fallback buffer.

If the fallback implementation is a best-fit fallback or a replacement fallback, the classes derived from EncoderFallbackBuffer and DecoderFallbackBuffer also maintain two private instance fields: the exact number of characters in the buffer; and the index of the next character in the buffer to return.

An EncoderFallback Example

An earlier example used replacement fallback to replace Unicode characters that did not correspond to ASCII characters with an asterisk (*). The following example uses a custom best-fit fallback implementation instead to provide a better mapping of non-ASCII characters.

The following code defines a class named `CustomMapper` that is derived from EncoderFallback to handle the best-fit mapping of non-ASCII characters. Its `CreateFallbackBuffer` method returns a `CustomMapperFallbackBuffer` object, which provides the EncoderFallbackBuffer implementation. The `CustomMapper` class uses a Dictionary<TKey,TValue> object to store the mappings of unsupported Unicode characters (the key value) and their corresponding 8-bit characters (which are stored in two consecutive bytes in a 64-bit integer). To make this mapping available to the fallback buffer, the `CustomMapper` instance is passed as a parameter to the `CustomMapperFallbackBuffer` class constructor. Because the longest mapping is the string "INF" for the Unicode character U+221E, the `MaxCharCount` property returns 3.

```
public class CustomMapper : EncoderFallback
{
   public string DefaultString;
   internal Dictionary<ushort, ulong> mapping;

   public CustomMapper() : this("*")
   {
   }

   public CustomMapper(string defaultString)
   {
      this.DefaultString = defaultString;

      // Create table of mappings
      mapping = new Dictionary<ushort, ulong>();
      mapping.Add(0x24C8, 0x53);
      mapping.Add(0x2075, 0x35);
      mapping.Add(0x221E, 0x49004E0046);
   }

   public override EncoderFallbackBuffer CreateFallbackBuffer()
   {
      return new CustomMapperFallbackBuffer(this);
   }

   public override int MaxCharCount
   {
      get { return 3; }
   }
}
```

```
Public Class CustomMapper : Inherits EncoderFallback
   Public DefaultString As String
   Friend mapping As Dictionary(Of UShort, ULong)

   Public Sub New()
      Me.New("?")
   End Sub

   Public Sub New(ByVal defaultString As String)
      Me.DefaultString = defaultString

      ' Create table of mappings
      mapping = New Dictionary(Of UShort, ULong)
      mapping.Add(&H24C8, &H53)
      mapping.Add(&H2075, &H35)
      mapping.Add(&H221E, &H49004E0046)
   End Sub

   Public Overrides Function CreateFallbackBuffer() As System.Text.EncoderFallbackBuffer
      Return New CustomMapperFallbackBuffer(Me)
   End Function

   Public Overrides ReadOnly Property MaxCharCount As Integer
      Get
         Return 3
      End Get
   End Property
End Class
```

The following code defines the `CustomMapperFallbackBuffer` class, which is derived from EncoderFallbackBuffer. The dictionary that contains best-fit mappings and that is defined in the `CustomMapper` instance is available from its class constructor. Its `Fallback` method returns `true` if any of the Unicode characters that the ASCII encoder cannot encode are defined in the mapping dictionary; otherwise, it returns `false`. For each fallback, the private `count` variable indicates the number of characters that remain to be returned, and the private `index` variable indicates the position in the string buffer, `charsToReturn`, of the next character to return.

```
public class CustomMapperFallbackBuffer : EncoderFallbackBuffer
{
   int count = -1;                   // Number of characters to return
   int index = -1;                   // Index of character to return
   CustomMapper fb;
   string charsToReturn;

   public CustomMapperFallbackBuffer(CustomMapper fallback)
   {
      this.fb = fallback;
   }

   public override bool Fallback(char charUnknownHigh, char charUnknownLow, int index)
   {
      // Do not try to map surrogates to ASCII.
      return false;
   }

   public override bool Fallback(char charUnknown, int index)
   {
      // Return false if there are already characters to map.
      if (count >= 1) return false;

      // Determine number of characters to return.
      charsToReturn = String.Empty;

      ushort key = Convert.ToUInt16(charUnknown);
      if (fb.mapping.ContainsKey(key)) {
```

```
            byte[] bytes = BitConverter.GetBytes(fb.mapping[key]);
            int ctr = 0;
            foreach (var byt in bytes) {
               if (byt > 0) {
                  ctr++;
                  charsToReturn += (char) byt;
               }
            }
            count = ctr;
         }
         else {
            // Return default.
            charsToReturn = fb.DefaultString;
            count = 1;
         }
         this.index = charsToReturn.Length - 1;

         return true;
      }

      public override char GetNextChar()
      {
         // We'll return a character if possible, so subtract from the count of chars to return.
         count--;
         // If count is less than zero, we've returned all characters.
         if (count < 0)
            return '\u0000';

         this.index--;
         return charsToReturn[this.index + 1];
      }

      public override bool MovePrevious()
      {
         // Original: if count >= -1 and pos >= 0
         if (count >= -1) {
            count++;
            return true;
         }
         else {
            return false;
         }
      }

      public override int Remaining
      {
         get { return count < 0 ? 0 : count; }
      }

      public override void Reset()
      {
         count = -1;
         index = -1;
      }
   }
```

```
Public Class CustomMapperFallbackBuffer : Inherits EncoderFallbackBuffer

   Dim count As Integer = -1        ' Number of characters to return
   Dim index As Integer = -1        ' Index of character to return
   Dim fb As CustomMapper
   Dim charsToReturn As String

   Public Sub New(ByVal fallback As CustomMapper)
      MyBase.New()
      Me.fb = fallback
   End Sub
```

```
   Public Overloads Overrides Function Fallback(ByVal charUnknownHigh As Char, ByVal charUnknownLow As Char,
ByVal index As Integer) As Boolean
      ' Do not try to map surrogates to ASCII.
      Return False
   End Function

   Public Overloads Overrides Function Fallback(ByVal charUnknown As Char, ByVal index As Integer) As Boolean
      ' Return false if there are already characters to map.
      If count >= 1 Then Return False

      ' Determine number of characters to return.
      charsToReturn = String.Empty

      Dim key As UShort = Convert.ToUInt16(charUnknown)
      If fb.mapping.ContainsKey(key) Then
         Dim bytes() As Byte = BitConverter.GetBytes(fb.mapping.Item(key))
         Dim ctr As Integer
         For Each byt In bytes
            If byt > 0 Then
               ctr += 1
               charsToReturn += Chr(byt)
            End If
         Next
         count = ctr
      Else
         ' Return default.
         charsToReturn = fb.DefaultString
         count = 1
      End If
      Me.index = charsToReturn.Length - 1

      Return True
   End Function

   Public Overrides Function GetNextChar() As Char
      ' We'll return a character if possible, so subtract from the count of chars to return.
      count -= 1
      ' If count is less than zero, we've returned all characters.
      If count < 0 Then Return ChrW(0)

      Me.index -= 1
      Return charsToReturn(Me.index + 1)
   End Function

   Public Overrides Function MovePrevious() As Boolean
      ' Original: if count >= -1 and pos >= 0
      If count >= -1 Then
         count += 1
         Return True
      Else
         Return False
      End If
   End Function

   Public Overrides ReadOnly Property Remaining As Integer
      Get
         Return If(count < 0, 0, count)
      End Get
   End Property

   Public Overrides Sub Reset()
      count = -1
      index = -1
   End Sub
End Class
```

The following code then instantiates the `CustomMapper` object and passes an instance of it to the

Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) method. The output indicates that the best-fit fallback implementation successfully handles the three non-ASCII characters in the original string.

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

class Program
{
   static void Main()
   {
      Encoding enc = Encoding.GetEncoding("us-ascii", new CustomMapper(), new DecoderExceptionFallback());

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      for (int ctr = 0; ctr <= str1.Length - 1; ctr++) {
         Console.Write("{0} ", Convert.ToUInt16(str1[ctr]).ToString("X4"));
         if (ctr == str1.Length - 1)
            Console.WriteLine();
      }
      Console.WriteLine();

      // Encode the original string using the ASCII encoder.
      byte[] bytes = enc.GetBytes(str1);
      Console.Write("Encoded bytes: ");
      foreach (var byt in bytes)
         Console.Write("{0:X2} ", byt);

      Console.WriteLine("\n");

      // Decode the ASCII bytes.
      string str2 = enc.GetString(bytes);
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
      if (! str1.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

         Console.WriteLine();
      }
   }
}
```

```
Imports System.Text
Imports System.Collections.Generic

Module Module1

   Sub Main()
      Dim enc As Encoding = Encoding.GetEncoding("us-ascii", New CustomMapper(), New
DecoderExceptionFallback())

      Dim str1 As String = String.Format("{0} {1} {2}", ChrW(&H24C8), ChrW(&H2075), ChrW(&H221E))
      Console.WriteLine(str1)
      For ctr As Integer = 0 To str1.Length - 1
         Console.Write("{0} ", Convert.ToUInt16(str1(ctr)).ToString("X4"))
         If ctr = str1.Length - 1 Then Console.WriteLine()
      Next
      Console.WriteLine()

      ' Encode the original string using the ASCII encoder.
      Dim bytes() As Byte = enc.GetBytes(str1)
      Console.Write("Encoded bytes: ")
      For Each byt In bytes
         Console.Write("{0:X2} ", byt)
      Next
      Console.WriteLine()
      Console.WriteLine()

      ' Decode the ASCII bytes.
      Dim str2 As String = enc.GetString(bytes)
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2))
      If Not str1.Equals(str2) Then
         Console.WriteLine(str2)
         For Each ch In str2
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
         Next
         Console.WriteLine()
      End If
   End Sub
End Module
```

See also

- Encoder
- Decoder
- DecoderFallback
- Encoding
- EncoderFallback
- Globalization and Localization

Parsing Strings in .NET

5/2/2018 • 2 minutes to read • Edit Online

A parsing operation converts a string that represents a .NET base type into that base type. For example, a parsing operation is used to convert a string to a floating-point number or to a date and time value. The method most commonly used to perform a parsing operation is the `Parse` method. Because parsing is the reverse operation of formatting (which involves converting a base type into its string representation), many of the same rules and conventions apply. Just as formatting uses an object that implements the IFormatProvider interface to provide culture-sensitive formatting information, parsing also uses an object that implements the IFormatProvider interface to determine how to interpret a string representation. For more information, see Formatting Types.

In This Section

Parsing Numeric Strings
Describes how to convert strings into .NET numeric types.

Parsing Date and Time Strings
Describes how to convert strings into .NET **DateTime** types.

Parsing Other Strings
Describes how to convert strings into **Char**, **Boolean**, and **Enum** types.

Related Sections

Formatting Types
Describes basic formatting concepts like format specifiers and format providers.

Type Conversion in .NET
Describes how to convert types.

Base Types
Describes common operations that you can perform on .NET base types.

Parsing Numeric Strings in NET

9/6/2018 • 8 minutes to read • Edit Online

All numeric types have two static parsing methods, `Parse` and `TryParse`, that you can use to convert the string representation of a number into a numeric type. These methods enable you to parse strings that were produced by using the format strings documented in Standard Numeric Format Strings and Custom Numeric Format Strings. By default, the `Parse` and `TryParse` methods can successfully convert strings that contain integral decimal digits only to integer values. They can successfully convert strings that contain integral and fractional decimal digits, group separators, and a decimal separator to floating-point values. The `Parse` method throws an exception if the operation fails, whereas the `TryParse` method returns `false`.

Parsing and Format Providers

Typically, the string representations of numeric values differ by culture. Elements of numeric strings such as currency symbols, group (or thousands) separators, and decimal separators all vary by culture. Parsing methods either implicitly or explicitly use a format provider that recognizes these culture-specific variations. If no format provider is specified in a call to the `Parse` or `TryParse` method, the format provider associated with the current thread culture (the NumberFormatInfo object returned by the NumberFormatInfo.CurrentInfo property) is used.

A format provider is represented by an IFormatProvider implementation. This interface has a single member, the GetFormat method, whose single parameter is a Type object that represents the type to be formatted. This method returns the object that provides formatting information. .NET supports the following two IFormatProvider implementations for parsing numeric strings:

- A CultureInfo object whose CultureInfo.GetFormat method returns a NumberFormatInfo object that provides culture-specific formatting information.
- A NumberFormatInfo object whose NumberFormatInfo.GetFormat method returns itself.

The following example tries to convert each string in an array to a Double value. It first tries to parse the string by using a format provider that reflects the conventions of the English (United States) culture. If this operation throws a FormatException, it tries to parse the string by using a format provider that reflects the conventions of the French (France) culture.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string[] values = { "1,304.16", "$1,456.78", "1,094", "152",
                          "123,45 €", "1 304,16", "Ae9f" };
      double number;
      CultureInfo culture = null;

      foreach (string value in values) {
         try {
            culture = CultureInfo.CreateSpecificCulture("en-US");
            number = Double.Parse(value, culture);
            Console.WriteLine("{0}: {1} --> {2}", culture.Name, value, number);
         }
         catch (FormatException) {
            Console.WriteLine("{0}: Unable to parse '{1}'.",
                              culture.Name, value);
            culture = CultureInfo.CreateSpecificCulture("fr-FR");
            try {
               number = Double.Parse(value, culture);
               Console.WriteLine("{0}: {1} --> {2}", culture.Name, value, number);
            }
            catch (FormatException) {
               Console.WriteLine("{0}: Unable to parse '{1}'.",
                                 culture.Name, value);
            }
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//    en-US: 1,304.16 --> 1304.16
//
//    en-US: Unable to parse '$1,456.78'.
//    fr-FR: Unable to parse '$1,456.78'.
//
//    en-US: 1,094 --> 1094
//
//    en-US: 152 --> 152
//
//    en-US: Unable to parse '123,45 €'.
//    fr-FR: Unable to parse '123,45 €'.
//
//    en-US: Unable to parse '1 304,16'.
//    fr-FR: 1 304,16 --> 1304.16
//
//    en-US: Unable to parse 'Ae9f'.
//    fr-FR: Unable to parse 'Ae9f'.
```

```
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim values() As String = { "1,304.16", "$1,456.78", "1,094", "152",
                                 "123,45 €", "1 304,16", "Ae9f" }
      Dim number As Double
      Dim culture As CultureInfo = Nothing

      For Each value As String In values
         Try
            culture = CultureInfo.CreateSpecificCulture("en-US")
            number = Double.Parse(value, culture)
            Console.WriteLine("{0}: {1} --> {2}", culture.Name, value, number)
         Catch e As FormatException
            Console.WriteLine("{0}: Unable to parse '{1}'.",
                              culture.Name, value)
            culture = CultureInfo.CreateSpecificCulture("fr-FR")
            Try
               number = Double.Parse(value, culture)
               Console.WriteLine("{0}: {1} --> {2}", culture.Name, value, number)
            Catch ex As FormatException
               Console.WriteLine("{0}: Unable to parse '{1}'.",
                                 culture.Name, value)
            End Try
         End Try
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays the following output:
'    en-US: 1,304.16 --> 1304.16
'
'    en-US: Unable to parse '$1,456.78'.
'    fr-FR: Unable to parse '$1,456.78'.
'
'    en-US: 1,094 --> 1094
'
'    en-US: 152 --> 152
'
'    en-US: Unable to parse '123,45 €'.
'    fr-FR: Unable to parse '123,45 €'.
'
'    en-US: Unable to parse '1 304,16'.
'    fr-FR: 1 304,16 --> 1304.16
'
'    en-US: Unable to parse 'Ae9f'.
'    fr-FR: Unable to parse 'Ae9f'.
```

Parsing and NumberStyles Values

The style elements (such as white space, group separators, and decimal separator) that the parse operation can handle are defined by a NumberStyles enumeration value. By default, strings that represent integer values are parsed by using the NumberStyles.Integer value, which permits only numeric digits, leading and trailing white space, and a leading sign. Strings that represent floating-point values are parsed using a combination of the NumberStyles.Float and NumberStyles.AllowThousands values; this composite style permits decimal digits along with leading and trailing white space, a leading sign, a decimal separator, a group separator, and an exponent. By calling an overload of the `Parse` or `TryParse` method that includes a parameter of type NumberStyles and setting one or more NumberStyles flags, you can control the style elements that can be present in the string for the parse operation to succeed.

For example, a string that contains a group separator cannot be converted to an Int32 value by using the

Int32.Parse(String) method. However, the conversion succeeds if you use the NumberStyles.AllowThousands flag, as the following example illustrates.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string value = "1,304";
      int number;
      IFormatProvider provider = CultureInfo.CreateSpecificCulture("en-US");
      if (Int32.TryParse(value, out number))
         Console.WriteLine("{0} --> {1}", value, number);
      else
         Console.WriteLine("Unable to convert '{0}'", value);

      if (Int32.TryParse(value, NumberStyles.Integer | NumberStyles.AllowThousands,
                        provider, out number))
         Console.WriteLine("{0} --> {1}", value, number);
      else
         Console.WriteLine("Unable to convert '{0}'", value);
   }
}
// The example displays the following output:
//       Unable to convert '1,304'
//       1,304 --> 1304
```

```
Imports System.Globalization

Module Example
   Public Sub Main()
      Dim value As String = "1,304"
      Dim number As Integer
      Dim provider As IFormatProvider = CultureInfo.CreateSpecificCulture("en-US")
      If Int32.TryParse(value, number) Then
         Console.WriteLine("{0} --> {1}", value, number)
      Else
         Console.WriteLine("Unable to convert '{0}'", value)
      End If

      If Int32.TryParse(value, NumberStyles.Integer Or NumberStyles.AllowThousands,
                        provider, number) Then
         Console.WriteLine("{0} --> {1}", value, number)
      Else
         Console.WriteLine("Unable to convert '{0}'", value)
      End If
   End Sub
End Module
' The example displays the following output:
'       Unable to convert '1,304'
'       1,304 --> 1304
```

WARNING

The parse operation always uses the formatting conventions of a particular culture. If you do not specify a culture by passing a CultureInfo or NumberFormatInfo object, the culture associated with the current thread is used.

The following table lists the members of the NumberStyles enumeration and describes the effect that they have on the parsing operation.

NUMBERSTYLES VALUE	EFFECT ON THE STRING TO BE PARSED
NumberStyles.None	Only numeric digits are permitted.
NumberStyles.AllowDecimalPoint	The decimal separator and fractional digits are permitted. For integer values, only zero is permitted as a fractional digit. Valid decimal separators are determined by the NumberFormatInfo.NumberDecimalSeparator or NumberFormatInfo.CurrencyDecimalSeparator property.
NumberStyles.AllowExponent	The "e" or "E" character can be used to indicate exponential notation. See NumberStyles for additional information.
NumberStyles.AllowLeadingWhite	Leading white space is permitted.
NumberStyles.AllowTrailingWhite	Trailing white space is permitted.
NumberStyles.AllowLeadingSign	A positive or negative sign can precede numeric digits.
NumberStyles.AllowTrailingSign	A positive or negative sign can follow numeric digits.
NumberStyles.AllowParentheses	Parentheses can be used to indicate negative values.
NumberStyles.AllowThousands	The group separator is permitted. The group separator character is determined by the NumberFormatInfo.NumberGroupSeparator or NumberFormatInfo.CurrencyGroupSeparator property.
NumberStyles.AllowCurrencySymbol	The currency symbol is permitted. The currency symbol is defined by the NumberFormatInfo.CurrencySymbol property.
NumberStyles.AllowHexSpecifier	The string to be parsed is interpreted as a hexadecimal number. It can include the hexadecimal digits 0-9, A-F, and a-f. This flag can be used only to parse integer values.

In addition, the NumberStyles enumeration provides the following composite styles, which include multiple NumberStyles flags.

COMPOSITE NUMBERSTYLES VALUE	INCLUDES MEMBERS
NumberStyles.Integer	Includes the NumberStyles.AllowLeadingWhite, NumberStyles.AllowTrailingWhite, and NumberStyles.AllowLeadingSign styles. This is the default style used to parse integer values.
NumberStyles.Number	Includes the NumberStyles.AllowLeadingWhite, NumberStyles.AllowTrailingWhite, NumberStyles.AllowLeadingSign, NumberStyles.AllowTrailingSign, NumberStyles.AllowDecimalPoint, and NumberStyles.AllowThousands styles.

COMPOSITE NUMBERSTYLES VALUE	INCLUDES MEMBERS
NumberStyles.Float	Includes the NumberStyles.AllowLeadingWhite, NumberStyles.AllowTrailingWhite, NumberStyles.AllowLeadingSign, NumberStyles.AllowDecimalPoint, and NumberStyles.AllowExponent styles.
NumberStyles.Currency	Includes all styles except NumberStyles.AllowExponent and NumberStyles.AllowHexSpecifier.
NumberStyles.Any	Includes all styles except NumberStyles.AllowHexSpecifier.
NumberStyles.HexNumber	Includes the NumberStyles.AllowLeadingWhite, NumberStyles.AllowTrailingWhite, and NumberStyles.AllowHexSpecifier styles.

Parsing and Unicode Digits

The Unicode standard defines code points for digits in various writing systems. For example, code points from U+0030 to U+0039 represent the basic Latin digits 0 through 9, code points from U+09E6 to U+09EF represent the Bangla digits 0 through 9, and code points from U+FF10 to U+FF19 represent the Fullwidth digits 0 through 9. However, the only numeric digits recognized by parsing methods are the basic Latin digits 0-9 with code points from U+0030 to U+0039. If a numeric parsing method is passed a string that contains any other digits, the method throws a FormatException.

The following example uses the Int32.Parse method to parse strings that consist of digits in different writing systems. As the output from the example shows, the attempt to parse the basic Latin digits succeeds, but the attempt to parse the Fullwidth, Arabic-Indic, and Bangla digits fails.

```
using System;

public class Example
{
   public static void Main()
   {
      string value;
      // Define a string of basic Latin digits 1-5.
      value = "\u0031\u0032\u0033\u0034\u0035";
      ParseDigits(value);

      // Define a string of Fullwidth digits 1-5.
      value = "\uFF11\uFF12\uFF13\uFF14\uFF15";
      ParseDigits(value);

      // Define a string of Arabic-Indic digits 1-5.
      value = "\u0661\u0662\u0663\u0664\u0665";
      ParseDigits(value);

      // Define a string of Bangla digits 1-5.
      value = "\u09e7\u09e8\u09e9\u09ea\u09eb";
      ParseDigits(value);
   }

   static void ParseDigits(string value)
   {
      try {
         int number = Int32.Parse(value);
         Console.WriteLine("'{0}' --> {1}", value, number);
      }
      catch (FormatException) {
         Console.WriteLine("Unable to parse '{0}'.", value);
      }
   }
}
// The example displays the following output:
//       '12345' --> 12345
//       Unable to parse '１２３４５'.
//       Unable to parse '١٢٣٤٥'.
//       Unable to parse '■■■■■'.
```

```
Module Example
   Public Sub Main()
      Dim value As String
      ' Define a string of basic Latin digits 1-5.
      value = ChrW(&h31) + ChrW(&h32) + ChrW(&h33) + ChrW(&h34) + ChrW(&h35)
      ParseDigits(value)

      ' Define a string of Fullwidth digits 1-5.
      value = ChrW(&hff11) + ChrW(&hff12) + ChrW(&hff13) + ChrW(&hff14) + ChrW(&hff15)
      ParseDigits(value)

      ' Define a string of Arabic-Indic digits 1-5.
      value = ChrW(&h661) + ChrW(&h662) + ChrW(&h663) + ChrW(&h664) + ChrW(&h665)
      ParseDigits(value)

      ' Define a string of Bangla digits 1-5.
      value = ChrW(&h09e7) + ChrW(&h09e8) + ChrW(&h09e9) + ChrW(&h09ea) + ChrW(&h09eb)
      ParseDigits(value)
   End Sub

   Sub ParseDigits(value As String)
      Try
         Dim number As Integer = Int32.Parse(value)
         Console.WriteLine("'{0}' --> {1}", value, number)
      Catch e As FormatException
         Console.WriteLine("Unable to parse '{0}'.", value)
      End Try
   End Sub
End Module
' The example displays the following output:
'       '12345' --> 12345
'       Unable to parse '１２３４５'.
'       Unable to parse '١٢٣٤٥'.
'       Unable to parse '■■■■■'.
```

See also

- NumberStyles
- Parsing Strings
- Formatting Types

Parsing Date and Time Strings in .NET

9/6/2018 • 6 minutes to read • Edit Online

Parsing strings to convert them to DateTime objects requires you to specify information about how the dates and times are represented as text. Different cultures use different orders for day, month, and year. Some time representations use a 24-hour clock, others specify "AM" and "PM." Some applications need only the date. Others need only the time. Still others need to specify both the date and the time. The methods that convert strings to DateTime objects enable you to provide detailed information about the formats you expect and the elements of a date and time your application needs. There are three subtasks to correctly converting text into a DateTime:

1. You must specify the expected format of the text representing a date and time.
2. You may specify the culture for the format of a date time.
3. You may specify how missing components in the text representation are set in the date and time.

The Parse and TryParse methods convert many common representations of a date and time. The ParseExact and TryParseExact methods convert a string representation that conforms to the pattern specified by a date and time format string. (See the articles on standard date and time format strings and custom date and time format strings for details.)

The current DateTimeFormatInfo object provides more control over how text should be interpreted as a date and time. Properties of a DateTimeFormatInfo describe the date and time separators, and the names of months, days, and eras, and the format for the "AM" and "PM" designations. The current thread culture provides a DateTimeFormatInfo that represents the current culture. If you want a specific culture or custom settings, you specify the IFormatProvider parameter of a parsing method. For the IFormatProvider parameter, specify a CultureInfo object, which represents a culture, or a DateTimeFormatInfo object.

The text representing a date or time may be missing some information. For example, most people would assume the date "March 12" represents the current year. Similarly, "March 2018" represents the month of March in the year 2018. Text representing time often does only includes hours, minutes, and an AM/PM designation. Parsing methods handle this missing information by using reasonable defaults:

- When only the time is present, the date portion uses the current date.
- When only the date is present, the time portion is midnight.
- When the year isn't specified in a date, the current year is used.
- When the day of the month isn't specified, the first of the month is used.

If the date is present in the string, it must include the month and one of the day or year. If the time is present, it must include the hour, and either the minutes or the AM/PM designator.

You can specify the NoCurrentDateDefault constant to override these defaults. When you use that constant, any missing year, month, or day properties are set to the value `1`. The last example using Parse demonstrates this behavior.

In addition to a date and a time component, the string representation of a date and time can include an offset that indicates how much the time differs from Coordinated Universal Time (UTC). For example, the string "2/14/2007 5:32:00 -7:00" defines a time that is seven hours earlier than UTC. If an offset is omitted from the string representation of a time, parsing returns a DateTime object with its Kind property set to DateTimeKind.Unspecified. If an offset is specified, parsing returns a DateTime object with its Kind property set to DateTimeKind.Local and its value adjusted to the local time zone of your machine. You can modify this behavior by using a DateTimeStyles value with the parsing method.

The format provider is also used to interpret an ambiguous numeric date. It is not clear which components of the date represented by the string "02/03/04" are the month, day, and year. The components are interpreted according to the order of similar date formats in the format provider.

Parse

The following example illustrates the use of the DateTime.Parse method to convert a `string` into a DateTime. This example uses the culture associated with the current thread. If the CultureInfo associated with the current culture cannot parse the input string, a FormatException is thrown.

> **TIP**
>
> All the C# samples in this article run in your browser. Press the **Run** button to see the output. You can also edit them to experiment yourself.

> **NOTE**
>
> These examples are available in the GitHub docs repo for both C# and VB. Or you can download the project as a zipfile for C# or VB.

```
string dateInput = "Jan 1, 2009";
DateTime parsedDate = DateTime.Parse(dateInput);
Console.WriteLine(parsedDate);
// Displays the following output on a system whose culture is en-US:
//       1/1/2009 12:00:00 AM
```

```
Dim MyString As String = "Jan 1, 2009"
Dim MyDateTime As DateTime = DateTime.Parse(MyString)
Console.WriteLine(MyDateTime)
' Displays the following output on a system whose culture is en-US:
'       1/1/2009 12:00:00 AM
```

You can also explicitly define the culture whose formatting conventions are used when you parse a string. You specify one of the standard DateTimeFormatInfo objects returned by the CultureInfo.DateTimeFormat property. The following example uses a format provider to parse a German string into a DateTime. It creates a CultureInfo representing the `de-DE` culture. That `CultureInfo` object ensures successful parsing of this particular string. This precludes whatever setting is in the CurrentCulture of the CurrentThread.

```
CultureInfo MyCultureInfo = new CultureInfo("de-DE");
string MyString = "12 Juni 2008";
DateTime MyDateTime = DateTime.Parse(MyString, MyCultureInfo);
Console.WriteLine(MyDateTime);
// The example displays the following output:
//       6/12/2008 12:00:00 AM
```

```
Dim MyCultureInfo As CultureInfo = New CultureInfo("de-DE")
Dim MyString As String = "12 Juni 2008"
Dim MyDateTime As DateTime = DateTime.Parse(MyString, MyCultureInfo)
Console.WriteLine(MyDateTime)
' The example displays the following output:
'       6/12/2008 12:00:00 AM
```

However, although you can use overloads of the Parse method to specify custom format providers, the method does not support parsing non-standard formats. To parse a date and time expressed in a non-standard format, use the ParseExact method instead.

The following example uses the DateTimeStyles enumeration to specify that the current date and time information should not be added to the DateTime for unspecified fields.

```
CultureInfo MyCultureInfo = new CultureInfo("de-DE");
string MyString = "12 Juni 2008";
DateTime MyDateTime = DateTime.Parse(MyString, MyCultureInfo,
                                     DateTimeStyles.NoCurrentDateDefault);
Console.WriteLine(MyDateTime);
// The example displays the following output if the current culture is en-US:
//      6/12/2008 12:00:00 AM
```

```
Dim MyCultureInfo As CultureInfo = New CultureInfo("de-DE")
Dim MyString As String = "12 Juni 2008"
Dim MyDateTime As DateTime = DateTime.Parse(MyString, MyCultureInfo,
                           DateTimeStyles.NoCurrentDateDefault)
Console.WriteLine(MyDateTime)
' The example displays the following output if the current culture is en-US:
'       6/12/2008 12:00:00 AM
```

ParseExact

The DateTime.ParseExact method converts a string to a DateTime object if it conforms to one of the specified string patterns. When a string that is not one of the forms specified is passed to this method, a FormatException is thrown. You can specify one of the standard date and time format specifiers or a combination of the custom format specifiers. Using the custom format specifiers, it is possible for you to construct a custom recognition string. For an explanation of the specifiers, see the topics on standard date and time format strings and custom date and time format strings.

In the following example, the DateTime.ParseExact method is passed a string object to parse, followed by a format specifier, followed by a CultureInfo object. This ParseExact method can only parse strings that follow the long date pattern in the `en-US` culture.

```
CultureInfo MyCultureInfo = new CultureInfo("en-US");
string[] MyString = { " Friday, April 10, 2009", "Friday, April 10, 2009" };
foreach (string dateString in MyString)
{
    try
    {
        DateTime MyDateTime = DateTime.ParseExact(dateString, "D", MyCultureInfo);
        Console.WriteLine(MyDateTime);
    }
    catch (FormatException)
    {
        Console.WriteLine("Unable to parse '{0}'", dateString);
    }
}
// The example displays the following output:
//       Unable to parse ' Friday, April 10, 2009'
//       4/10/2009 12:00:00 AM
```

```vb
Dim MyCultureInfo As CultureInfo = New CultureInfo("en-US")
Dim MyString() As String = {" Friday, April 10, 2009", "Friday, April 10, 2009"}
For Each dateString As String In MyString
    Try
        Dim MyDateTime As DateTime = DateTime.ParseExact(dateString, "D",
                                                          MyCultureInfo)
        Console.WriteLine(MyDateTime)
    Catch e As FormatException
        Console.WriteLine("Unable to parse '{0}'", dateString)
    End Try
Next
' The example displays the following output:
'       Unable to parse ' Friday, April 10, 2009'
'       4/10/2009 12:00:00 AM
```

Each overload of the Parse and ParseExact methods also has an IFormatProvider parameter that provides culture-specific information about the formatting of the string. This IFormatProvider object is a CultureInfo object that represents a standard culture or a DateTimeFormatInfo object that is returned by the CultureInfo.DateTimeFormat property. ParseExact also uses an additional string or string array argument that defines one or more custom date and time formats.

See also

- Parsing Strings
- Formatting Types
- Type Conversion in .NET
- Standard date and time formats
- Custom date and time format strings

Parsing Other Strings in .NET

9/6/2018 • 2 minutes to read • Edit Online

In addition to numeric and DateTime strings, you can also parse strings that represent the types Char, Boolean, and Enum into data types.

Char

The static parse method associated with the **Char** data type is useful for converting a string that contains a single character into its Unicode value. The following code example parses a string into a Unicode character.

```
String^ MyString1 = "A";
char MyChar = Char::Parse(MyString1);
// MyChar now contains a Unicode "A" character.
```

```
string MyString1 = "A";
char MyChar = Char.Parse(MyString1);
// MyChar now contains a Unicode "A" character.
```

```
Dim MyString1 As String = "A"
Dim MyChar As Char = Char.Parse(MyString1)
' MyChar now contains a Unicode "A" character.
```

Boolean

The **Boolean** data type contains a **Parse** method that you can use to convert a string that represents a Boolean value into an actual **Boolean** type. This method is not case-sensitive and can successfully parse a string containing "True" or "False." The **Parse** method associated with the **Boolean** type can also parse strings that are surrounded by white spaces. If any other string is passed, a FormatException is thrown.

The following code example uses the **Parse** method to convert a string into a Boolean value.

```
String^ MyString2 = "True";
bool MyBool = bool::Parse(MyString2);
// MyBool now contains a True Boolean value.
```

```
string MyString2 = "True";
bool MyBool = bool.Parse(MyString2);
// MyBool now contains a True Boolean value.
```

```
Dim MyString2 As String = "True"
Dim MyBool As Boolean = Boolean.Parse(MyString2)
' MyBool now contains a True Boolean value.
```

Enumeration

You can use the static **Parse** method to initialize an enumeration type to the value of a string. This method accepts the enumeration type you are parsing, the string to parse, and an optional Boolean flag indicating whether or not the parse is case-sensitive. The string you are parsing can contain several values separated by commas, which can be preceded or followed by one or more empty spaces (also called white spaces). When the string contains multiple values, the value of the returned object is the value of all specified values combined with a bitwise OR operation.

The following example uses the **Parse** method to convert a string representation into an enumeration value. The DayOfWeek enumeration is initialized to **Thursday** from a string.

```
String^ MyString3 = "Thursday";
DayOfWeek MyDays = (DayOfWeek)Enum::Parse(DayOfWeek::typeid, MyString3);
Console::WriteLine(MyDays);
// The result is Thursday.
```

```
string MyString3 = "Thursday";
DayOfWeek MyDays = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), MyString3);
Console.WriteLine(MyDays);
// The result is Thursday.
```

```
Dim MyString3 As String = "Thursday"
Dim MyDays As DayOfWeek = CType([Enum].Parse(GetType(DayOfWeek), MyString3), DayOfWeek)
Console.WriteLine("{0:G}", MyDays)
' The result is Thursday.
```

See also

- Parsing Strings
- Formatting Types
- Type Conversion in the .NET