Attributes

You're already familiar with the notion of attributing code elements of a program with modifiers, such as `virtual` or `ref`. These constructs are built into the language. *Attributes* are an extensible mechanism for adding custom information to code elements (assemblies, types, members, return values, and parameters). This extensibility is useful for services that integrate deeply into the type system, without requiring special keywords or constructs in the C# language.

A good scenario for attributes is serialization—the process of converting arbitrary objects to and from a particular format. In this scenario, an attribute on a field can specify the translation between C#'s representation of the field and the format's representation of the field.

Attribute Classes

An attribute is defined by a class that inherits (directly or indirectly) from the abstract class `System.Attribute`. To attach an attribute to a code element, specify the attribclass `System.Attribute`. To attach an attribute to a code element, specify the attribute's type name in square brackets, before the code element. For example, the following attaches the `ObsoleteAttribute` to the `Foo` class:

```
[ObsoleteAttribute]
public class Foo {...}
```


This attribute is recognized by the compiler and will cause compiler warnings if a type or member marked obsolete is referenced. By convention, all attribute types type or member marked obsolete is referenced. By convention, all attribute types end in the word *Attribute*. C# recognizes this and allows you to omit the suffix when attaching an attribute:

```
[Obsolete]
public class Foo {...}
```

`ObsoleteAttribute` is a type declared in the `System` namespace as follows (simplified for brevity):

```
public sealed class ObsoleteAttribute : Attribute {...}
```

The C# language and the .NET Framework include a number of predefined attributes. We describe how to write your own attributes in Chapter 17.

Named and Positional Attribute Parameters

Attributes may have parameters. In the following example, we apply `XmlElementAttribute` to a class. This attribute tells XML serializer (in `System.Xml.Serialization`) how an object is represented in XML and accepts several *attribute* *parameters*. The following attribute maps the `CustomerEntity` class to an XML element named `Customer`, belonging to the *http://oreilly.com* namespace:

```
[XmlElement ("Customer", Namespace="http://oreilly.com")]
public class CustomerEntity { ... }
```

Attribute parameters fall into one of two categories: positional or named. In the preceding example, the first argument is a *positional parameter*; the second is a *named parameter*. Positional parameters correspond to parameters of the attribute type's public constructors. Named parameters correspond to public fields or public properties on the attribute type.

When specifying an attribute, you must include positional parameters that correspond to one of the attribute's constructors. Named parameters are optional.

In Chapter 18, we describe the valid parameter types and rules for their evaluation.

Attribute Targets

Implicitly, the target of an attribute is the code element it immediately precedes, which is typically a type or type member. However, you can also attach attributes to an assembly. This requires that you explicitly specify the attribute's target.

Here is an example of using the `CLSCompliant` attribute to specify CLS compliance for an entire assembly:

```
[assembly:CLSCompliant(true)]
```

Specifying Multiple Attributes

Multiple attributes can be specified for a single code element. Each attribute can be listed either within the same pair of square brackets (separated by a comma) or in separate pairs of square brackets (or a combination of the two). The following three examples are semantically identical:

```
[Serializable, Obsolete, CLSCompliant(false)]
public class Bar {...}
```

```
[Serializable] [Obsolete] [CLSCompliant(false)]
public class Bar {...}
```

```
public class Bar {...}

[Serializable, Obsolete]
[CLSCompliant(false)]
public class Bar {...}
```

Unsafe Code and Pointers

C# supports direct memory manipulation via pointers within blocks of code marked unsafe and compiled with the `/unsafe` compiler option. Pointer types are primarily useful for interoperability with C APIs, but may also be used for accessing memory outside the managed heap or for performance-critical hotspots.

Pointer Basics

For every value type or pointer type *V*, there is a corresponding pointer type *V**. A

For every value type or pointer type *V*, there is a corresponding pointer type *V**. A pointer instance holds the address of a value. This is considered to be of type *V*, but pointer types can be (unsafely) cast to any other pointer type. The main pointer operators are:

Operator	Meaning
`&`	The *address-of* operator returns a pointer to the address of a value
`*`	The *dereference* operator returns the value at the address of a pointer
`->`	The *pointer-to-member* operator is a syntactic shortcut, in which `x->y` is equivalent to `(*x).y`

Unsafe Code

By marking a type, type member, or statement block with the `unsafe` keyword, you're permitted to use pointer types and perform C++ style pointer operations on memory within that scope. Here is an example of using pointers to quickly process a bitmap:

```
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
```

```
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
      *p++ &= 0xFF;
  }
}
```

Unsafe code can run faster than a corresponding safe implementation. In this case, the code would have required a nested loop with array indexing and bounds checking. An unsafe C# method may also be faster than calling an external C function, since there is no overhead associated with leaving the managed execution environment.

The fixed Statement

The `fixed` statement is required to pin a managed object, such as the bitmap in the previous example. During the execution of a program, many objects are allocated and deallocated from the heap. In order to avoid unnecessary waste or fragmentation of memory, the garbage collector moves objects around. Pointing to an object is futile if its address could change while referencing it, so the `fixed` statement tells the garbage collector to "pin" the object and not move it around. This may have an impact on the efficiency of the runtime, so fixed blocks should be used only briefly, and heap allocation should be avoided within the fixed block.

Within a `fixed` statement, you can get a pointer to any value type, an array of value types, or a string. In the case of arrays and strings, the pointer will actually point to the first element, which is a value type.

the first element, which is a value type.

Value types declared inline within reference types require the reference type to be pinned, as follows:

```
class Test
{
  int x;
  static void Main()
  {
    Test test = new Test();
    unsafe
    {
       fixed (int* p = &test.x)   // Pins test
       {
         *p = 9;
       }
       System.Console.WriteLine (test.x);
    }
  }
}
```

We describe the `fixed` statement further in "Mapping a Struct to Unmanaged Mem-

We describe the `fixed` statement further in "Mapping a Struct to Unmanaged Memory" on page 965 in Chapter 25.

The Pointer-to-Member Operator

In addition to the `&` and `*` operators, C# also provides the C++ style `->` operator, which can be used on structs:

```
struct Test
{
  int x;
  unsafe static void Main()
  {
    Test test = new Test();
    Test* p = &test;
```

```
    Test* p = &test;
    p->x = 9;
    System.Console.WriteLine (test.x);
  }
}
```

Arrays

The stackalloc keyword

Memory can be allocated in a block on the stack explicitly using the `stackalloc` keyword. Since it is allocated on the stack, its lifetime is limited to the execution of the method, just as with any other local variable. The block may use the [] operator to index into memory:

```
int* a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
```

```
int* a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
   Console.WriteLine (a[i]);   // Print raw memory
```


Fixed-size buffers

Memory can be allocated in a block within a struct using the `fixed` keyword:

```
unsafe struct UnsafeUnicodeString
{
  public short Length;
  public fixed byte Buffer[30];
}

unsafe class UnsafeClass
```

```
unsafe class UnsafeClass
{
  UnsafeUnicodeString uus;

  public UnsafeClass (string s)
  {
    uus.Length = (short)s.Length;
    fixed (byte* p = uus.Buffer)
      for (int i = 0; i < s.Length; i++)
        p[i] = (byte) s[i];
  }
}
class Test
{
  static void Main() { new UnsafeClass ("Christian Troy"); }
}
```

The fixed keyword is also used in this example to pin the object on the heap that

The `fixed` keyword is also used in this example to pin the object on the heap that contains the buffer (which will be the instance of `UnsafeClass`).

void*

A *void pointer* (`void*`) makes no assumptions about the type of the underlying data and is useful for functions that deal with raw memory. An implicit conversion exists

A *void pointer* (`void*`) makes no assumptions about the type of the underlying data and is useful for functions that deal with raw memory. An implicit conversion exists from any pointer type to `void*`. A `void*` cannot be dereferenced, and arithmetic operations cannot be performed on void pointers. For example:

```
class Test
{
  unsafe static void Main()
  {
    short[ ] a = {1,1,2,3,5,8,13,21,34,55};
      fixed (short* p = a)
      {
        //sizeof returns size of value-type in bytes
        Zap (p, a.Length * sizeof (short));
      }
    foreach (short x in a)
      System.Console.WriteLine (x);   // Prints all zeros
  }
```

```
unsafe static void Zap (void* memory, int byteCount)
```

```
unsafe static void Zap (void* memory, int byteCount)
{
  byte* b = (byte*) memory;
    for (int i = 0; i < byteCount; i++)
      *b++ = 0;
```



```
      }
    }
```

Pointers to Unmanaged Code

Pointers are also useful for accessing data outside the managed heap (such as when interacting with C DLLs or COM), or when dealing with data not in the main memory (such as graphics memory or a storage medium on an embedded device).

Preprocessor Directives

Preprocessor directives supply the compiler with additional information about regions of code. The most common preprocessor directives are the conditional directives, which provide a way to include or exclude regions of code from compilation. For example:

```
#define DEBUG
class MyClass
{
  int x;
  void Foo()
  {
    # if DEBUG
    Console.WriteLine ("Testing: x = {0}", x);
```

```
      Console.WriteLine ("Testing: x = {0}", x);
      # endif
    }
    ...
  }
```

In this class, the statement in `Foo` is compiled as conditionally dependent upon the presence of the `DEBUG` symbol. If we remove the `DEBUG` symbol, the statement is not compiled. Preprocessor symbols can be defined within a source file (as we have done), and they can be passed to the compiler with the `/define:`*`symbol`* command-line option.

With the `#if` and `#elif` directives, you can use the `||`, `&&`, and `!` operators to perform *or*, *and*, and *not* operations on multiple symbols. The following directive instructs the compiler to include the code that follows if the `TESTMODE` symbol is defined and the `DEBUG` symbol is not defined:

```
#if TESTMODE && !DEBUG
...
```

• • •

Bear in mind, however, that you're not building an ordinary C# expression, and the symbols upon which you operate have absolutely no connection to *variables*—static or otherwise.

The `#error` and `#warning` symbols prevent accidental misuse of conditional directives by making the compiler generate a warning or error given an undesirable set of compilation symbols. Table 4-1 lists the preprocessor directives.

Table 4-1. Preprocessor directives

Preprocessor directive	Action
`#define` *`symbol`*	Defines *`symbol`*
`#undef` *`symbol`*	Undefines *`symbol`*
`#if` *`symbol`* [*`operator symbol2`*]...	*`symbol`* to test; *`operator`*s are ==, !=, &&, and || followed by `#else`, `#elif`, and `#endif`
`#else`	Executes code to subsequent `#endif`
`#elif` *`symbol`* [*`operator symbol2`*]	Combines `#else` branch and `#if` test

`#elif` *symbol* [*operator symbol2*]	Combines `#else` branch and `#if` test
`#endif`	Ends conditional directives
`#warning` *text*	*text* of the warning to appear in compiler output
`#error` *text*	*text* of the error to appear in compiler output
`#line [` *number* `["`*file*`"] | hidden]`	*number* specifies the line in source code; *file* is the filename to appear in computer output; `hidden` instructs debuggers to skip over code from this point until the next `#line` directive
`#region` *name*	Marks the beginning of an outline
`#end` *region*	Ends an outline region

#end *region*	Ends an outline region

Conditional Attributes

An attribute decorated with the `Conditional` attribute will be compiled only if a given preprocessor symbol is present. For example:

```
// file1.cs
#define DEBUG
using System;
using System.Diagnostics;
[Conditional("DEBUG")]
public class TestAttribute : Attribute {}
```

```
// file2.cs
#define DEBUG
[Test]
class Foo
{
  [Test]
  string s;
}
```

```
}
```

The compiler will not incorporate the `[Test]` attributes if the DEBUG symbol is in scope for *file2.cs*.

Pragma Warning

The compiler generates a warning when it spots something in your code that seems unintentional. Unlike errors, warnings don't ordinarily prevent your application from compiling.

Compiler warnings can be extremely valuable in spotting bugs. Their usefulness, however, is undermined when you get *false* warnings. In a large application, maintaining a good signal-to-noise ratio is essential if the "real" warnings are to get noticed.

To this effect, the compiler allows you to selectively suppress warnings with the `#pragma warning` directive. In this example, we instruct the compiler not to warn us

#pragma warning directive. In this example, we instruct the compiler not to warn us about the field Message not being used:

```
public class Foo
{
  static void Main() { }

  #pragma warning disable 414
  static string Message = "Hello";
  #pragma warning restore 414
}
```

Omitting the number in the #pragma warning directive disables or restores all warning codes.

If you are thorough in applying this directive, you can compile with

If you are thorough in applying this directive, you can compile with the /warnaserror switch—this tells the compiler to treat any residual warnings as errors.

XML Documentation

A *documentation comment* is a piece of embedded XML that documents a type or member. A documentation comment comes immediately before a type or member declaration, and starts with three slashes:

```
/// <summary>Cancels a running query.</summary>
public void Cancel() { ... }
```

Multiline comments can be done either like this:

```
/// <summary>
/// Cancels a running query
/// </summary>
```

```
    /// </summary>
    public void Cancel() { ... }
```

or like this (notice the extra star at the start):

```
/**
    <summary> Cancels a running query. </summary>
*/
public void Cancel() { ... }
```

If you compile with the `/doc` directive, the compiler extracts and collates documentation comments into a single XML file. This has two main uses:

- If placed in the same folder as the compiled assembly, Visual Studio automatically reads the XML file and uses the information to provide IntelliSense member listings to consumers of the assembly of the same name.

•

Third-party tools (such as Sandcastle and NDoc) can transform XML file into an HTML help file.

Standard XML Documentation Tags

Here are the standard XML tags that Visual Studio and documentation generators recognize:

<summary>

<summary>...</summary>

Indicates the tool tip that IntelliSense should display for the type or member; typically a single phrase or sentence.

Indicates the tool tip that IntelliSense should display for the type or member, typically a single phrase or sentence.

<remarks>

`<remarks>...</remarks>`

Additional text that describes the type or member. Documentation generators pick this up and merge it into the bulk of a type or member's description.

<param>

<code><param name="name">...</param></code>

Explains a parameter on a method.

<returns>

`<returns>...</returns>`

[redacted]

Explains the return value for a method.

`<exception>`

`<exception [cref="type"]>...</exception>`

Lists an exception that a method may throw (`cref` refers to the exception type).

`<permission>`

Lists an exception that a method may throw (cref refers to the exception type).

<permission>

<permission [cref="*type*"]>...</permission>

Indicates an `IPermission` type required by the documented type or member.

<example>

<example>...</example>

Denotes an example (used by documentation generators). This usually contains both description text and source code (source code is typically within a `<c>` or `<code>` tag).

<c>

<c>...</c>

Indicates an inline code snippet. This tag is usually used inside an `<example>` block.

block.

<code>

<code>...</code>

Indicates a multiline code sample. This tag is usually used inside an `<example>` block.

<see>

<see cref="*member*">...</see>

Inserts an inline cross-reference to another type or member. HTML documentation generators typically convert this to a hyperlink. The compiler emits a warning if the type or member name is invalid. To refer to generic types, use curly braces; for example, `cref="Foo{T,U}"`.

<seealso>

curly braces; for example, cref="Foo{T,U}".

<seealso>

```
<seealso cref="member">...</seealso>
```

Cross-references another type or member. Documentation generators typically write this into a separate "See Also" section at the bottom of the page.

<paramref>

```
<paramref name="name"/>
```

References a parameter from within a <summary> or <remarks> tag.

<list>

```
<list type=[ bullet | number | table ]>
  <listheader>
    <term>...</term>
    <description>...</description>
```

```
    <term>...</term>
    <description>...</description>
  </listheader>
  <item>
    <term>...</term>
    <description>...</description>
  </item>
</list>
```

Instructs documentation generators to emit a bulleted, numbered, or table-style list.

```
<para>
```

```
<para>...</para>
```

Instructs documentation generators to format the contents into a separate

Instructs documentation generators to format the contents into a separate paragraph.

`<include>`

Merges an external XML file that contains documentation. The path attribute denotes an XPath query to a specific element in that file.

User-Defined Tags

Little is special about the predefined XML tags recognized by the C# compiler, and you are free to define your own. The only special processing done by the compiler

is on the `<param>` tag (in which it verifies the parameter name and that all the parameters on the method are documented) and the `cref` attribute (in which it verifies that the attribute refers to a real type or member and expands it to a fully qualified type or member ID). The `cref` attribute can also be used in your own tags and is verified and expanded just as it is in the predefined `<exception>`, `<permission>`, `<see>`, and `<seealso>` tags.

<see>, and <seealso> tags.

Type or Member Cross-References

Type names and type or member cross-references are translated into IDs that uniquely define the type or member. These names are composed of a prefix that defines what the ID represents and a signature of the type or member. The member prefixes are:

XML type prefix	ID prefixes applied to...
N	Namespace
T	Type (class, struct, enum, interface, delegate)
F	Field
P	Property (includes indexers)
M	Method (includes special methods)

M	Method (includes special methods)
E	Event
!	Error

The rules describing how the signatures are generated are well documented, although fairly complex.

Here is an example of a type and the IDs that are generated:

```
// Namespaces do not have independent signatures
namespace NS
{
  /// T:NS.MyClass
  class MyClass
  {
    /// F:NS.MyClass.aField
    string aField;
```

```
/// P:NS.MyClass.aProperty
short aProperty {get {...} set {...}}
/// T:NS.MyClass.NestedType
class NestedType {...};
/// M:NS.MyClass.X()
```

```
/// M:NS.MyClass.X()
void X() {...}
```

```
/// M:NS.MyClass.Y(System.Int32,System.Double@,System.Decimal@)
void Y(int p1, ref double p2, out decimal p3) {...}
```



```
/// M:NS.MyClass.Z(System.Char[ ],System.Single[0:,0:])
void Z(char[ ] 1, float[,] p2) {...}
```

```
/// M:NS.MyClass.op_Addition(NS.MyClass,NS.MyClass)
public static MyClass operator+(MyClass c1, MyClass c2) {...}
```

```
/// M:NS.MyClass.op_Implicit(NS.MyClass)~System.Int32
public static implicit operator int(MyClass c) {...}

/// M:NS.MyClass.#ctor
MyClass() {...}
```

```
/// M:NS.MyClass.#ctor
MyClass() {...}

/// M:NS.MyClass.Finalize
~MyClass() {...}

    /// M:NS.MyClass.#cctor
    static MyClass() {...}
  }
}
```


5

Framework Overview

Almost all the capabilities of the .NET Framework are exposed via a vast set of managed types. These types are organized into hierarchical namespaces and packaged into a set of assemblies, which, together with the CLR, comprise the .NET platform.

Some of the .NET types are used directly by the CLR and are essential for the managed hosting environment. These types reside in an assembly called *mscorlib.dll* and include C#'s built-in types, as well as the basic collection classes, types for stream processing, serialization, reflection, threading, and native interoperability.

At a level above this are additional types that "flesh out" the CLR-level functionality, providing features such as XML, networking, and LINQ. These reside in *System.dll*

At a level above this are additional types that "flesh out" the CLR-level functionality, providing features such as XML, networking, and LINQ. These reside in *System.dll*, *System.Xml.dll*, and *System.Core.dll*, and together with *mscorlib* they provide a rich programming environment upon which the rest of the Framework is built. This "core framework" largely defines the scope of the rest of this book.

The remainder of the .NET Framework consists of applied APIs, most of which cover three areas of functionality:

- User interface technologies
- Backend technologies
- Distributed system technologies

Table 5-1 shows the history of compatibility between each version of C#, the CLR,

Table 5-1 shows the history of compatibility between each version of C#, the CLR, and the .NET Framework. Interestingly, C# 3.0 targeted a new Framework version while using the same CLR version as its predecessor. With C# 4.0, the numbers align cleanly again.

Table 5-1. C#, CLR, and .NET Framework versions

C# version	CLR version	Framework versions
1.0	1.0	1.0
1.2	1.1	1.1
2.0	2.0	2.0, 3.0
3.0	2.0 (SP1)	3.5
4.0	4.0	4.0

This chapter skims all key areas of the .NET Framework—starting with the core types covered in this book and finishing with an overview of the applied

This chapter skims all key areas of the .NET Framework—starting with the core types covered in this book and finishing with an overview of the applied technologies.

Assemblies and namespaces in the .NET Framework *cross-cut*. The most extreme examples are *System.Core.dll* and *mscorlib.dll*, both defining types in dozens of namespaces, none of which is prefixed with *mscorlib* or *System.Core*. The less obvious cases are the more confusing ones, however, such as the types in `System.Security.Cryptography`. Most types in this namespace reside in *System.dll*, except for a handful, which reside in *System.Security.dll*. This book's companion website has a complete mapping of Framework namespaces to assemblies.

What's New in .NET Framework 4.0

Framework 4.0 adds the following new features:

- New core types: `BigInteger` (for arbitrarily large numbers), `Complex` (complex numbers), and tuples (Chapter 6)

- New core types: `BigInteger` (for arbitrarily large numbers), `Complex` (complex numbers), and tuples (Chapter 6)
- A new `SortedSet` collection (Chapter 7)
- *Code contracts*, for enabling methods to interact more reliably through mutual obligations and responsibilities (Chapter 13)
- Direct support for memory-mapped files (Chapter 14)
- Lazy file- and directory-I/O methods that return `IEnumerable<T>` instead of arrays (Chapter 14)
- The *Dynamic Language Runtime* (DLR), which is now part of the .NET Framework (Chapter 19)
- *Security transparency*, which makes it easier to secure libraries in partially trusted environments (Chapter 20)
- New threading constructs, including a more robust `Monitor.Enter` overload, `Thread.Yield`, new signaling classes (`Barrier` and `CountdownEvent`), and lazy initialization primitives (Chapter 21)
- *Parallel programming* APIs for leveraging multicore processors, including Parallel LINQ (PLINQ), imperative data and task parallelism constructs,

concurrent collections, and low-latency synchronization and spinning primconcurrent collections, and low-latency synchronization and spinning primitives (Chapter 22)

- Methods for application domain resource monitoring (Chapter 24)

Framework 4.0 also includes enhancements to ASP.NET, including the MVC framework and Dynamic Data, and enhancements to Entity Framework, WPF, WCF, and Workflow. In addition, it includes the new *Managed Extensibility Framework* library, to assist with runtime composition, discovery, and dependency injection.

Many of the core types are defined in the following assemblies: *mscorlib.dll*, *System.dll*, and *System.Core.dll*. The first of these, *mscorlib.dll*, comprises the types required by the runtime environment itself; *System.dll* and *System.Core.dll* contain additional core types required by you as a programmer. The reason the latter two are separate is historical: when the Microsoft team introduced Framework 3.5, they made it *additive* insofar as it ran over the existing CLR 2.0. Therefore, almost all new core types (such as the classes supporting LINQ) went into a new assembly which Microsoft called *System.Core.dll*. With Framework 4.0, separation was maintained to avoid breaking existing applications.

A notable exception is the following types, which in Framework 4.0 have moved from *System.Core.dll* to *mscorlib.dll*:

- The `Action` and `Func` delegates
- `TimeZoneInfo` and associated types
- `System.Threading.LockRecursionException`

Redirections have been set up on these types, so if you ask the reflection API for the "TimeZoneInfo" type in *System.Core.dll*, you'll be given the type in *mscorlib.dll* instead.

rview

The CLR and Core Framework

System Types

The most fundamental types live directly in the `System` namespace. These include C#'s built-in types, the `Exception` base class, the `Enum`, `Array`, and `Delegate` base classes, and `Nullable`, `Type`, `DateTime`, `TimeSpan`, and `Guid`. The `System` namespace also includes types for performing mathematical functions (`Math`), generating random numbers (`Random`), and converting between various types (`Convert` and `BitConverter`).

Chapter 6 describes these types—as well as the interfaces that define standard protocols used across the .NET Framework for such tasks as formatting (`IFormattable`) and order comparison (`IComparable`).

protocols used across the .NET Framework for such tasks as formatting (`IFormattable`) and order comparison (`IComparable`).

The `System` namespace also defines the `IDisposable` interface and the `GC` class for interacting with the garbage collector. These topics are saved for Chapter 12.

Text Processing

The `System.Text` namespace contains the `StringBuilder` class (the editable or *mutable* cousin of `string`), and the types for working with text encodings, such as UTF-8 (`Encoding` and its subtypes). We cover this in Chapter 6.

The `System.Text.RegularExpressions` namespace contains types that perform advanced pattern-based search and replace operations; these are described in Chapter 26.

Collections

Collections

The .NET Framework offers a variety of classes for managing collections of items. These include both list- and dictionary-based structures, and work in conjunction with a set of standard interfaces that unify their common characteristics. All collection types are defined in the following namespaces, covered in Chapter 7:

```
System.Collections                  // Nongeneric collections
System.Collections.Generic          // Generic collections
System.Collections.Specialized      // Strongly typed collections
System.Collections.ObjectModel      // Bases for your own collections
System.Collections.Concurrent       // Thread-safe collection (Chapter 20)
```

Queries

Language Integrated Query (LINQ) was added in Framework 3.5. LINQ allows you to perform type-safe queries over local and remote collections (e.g., SQL Server tables) and is described in Chapters 8 through 10. A big advantage of LINQ is that it presents a consistent querying API across a variety of domains. The types for resolving LINQ queries reside in these namespaces:

```
System.Linq                   // LINQ to Objects and PLINQ
System.Xml.Linq               // LINQ to XML
```

```
System.Linq                  // LINQ to Objects and PLINQ
System.Xml.Linq              // LINQ to XML
System.Data.Linq             // LINQ to SQL
System.Data.Entity           // LINQ to Entities (Entity Framework)
System.Linq.Expressions      // For building expressions manually
```

The LINQ to SQL and Entity Framework APIs leverage lower-level ADO.NET types in the `System.Data` namespace.

XML

XML is used widely within the .NET Framework, and so is supported extensively. Chapter 10 focuses entirely on LINQ to XML—a lightweight XML document object model that can be constructed and queried through LINQ. Chapter 11 describes the older W3C DOM, as well as the performant low-level reader/writer classes and the Framework's support for XML schemas, stylesheets, and XPath. The XML namespaces are:

```
System.Xml                 // XmlReader, XmlWriter + the old W3C DOM
System.Xml.Linq            // The LINQ to XML DOM
System.Xml.Schema          // Support for XSD
System.Xml.XPath           // XPath query language
```

```
System.Xml.Xsl
System.Xml.Serialization
```

```
// Stylesheet support
// Declarative XML serialization for .NET types
```

Diagnostics and Code Contracts

In Chapter 13, we cover .NET's logging and assertion facilities and the new code contracts system in Framework 4.0. We also describe how to interact with other processes, write to the Windows event log, and use performance counters for monitoring. The types for this are defined in and under `System.Diagnostics`.

Streams and I/O

Streams and I/O

The Framework provides a stream-based model for low-level input/output. Streams are typically used to read and write directly to files and network connections, and can be chained or wrapped in decorator streams to add compression or encryption functionality. Chapter 14 describes .NET's stream architecture, as well as the specific support for working with files and directories, compression, isolated storage, pipes, and memory-mapped files. The `Stream` and I/O types are defined in and under the `System.IO` namespace.

Networking

You can directly access standard network protocols such as HTTP, FTP, TCP/IP, and SMTP via the types in `System.Net`. In Chapter 15, we demonstrate how to communicate using each of these protocols, starting with simple tasks such as downloading from a web page, and finishing with using TCP/IP directly to retrieve POP3 email. Here are the namespaces we cover:

System.Net

```
System.Net
System.Net.Mail
System.Net.Sockets

// For sending mail via SMTP
// TCP, UDP, and IP
```

Serialization

The Framework provides several systems for saving and restoring objects to a binary or text representation. Such systems are required for distributed application technologies, such as WCF, Web Services, and Remoting, and also to save and restore objects to a file. In Chapter 16, we cover all three serialization engines: the data contract serializer, the binary serializer, and the XML serializer. The types for serialization reside in the following namespaces:

```
System.Runtime.Serialization
System.Runtime.Serialization.Formatters.Binary
System.Runtime.Serialization.Formatters.SOAP
System.Xml.Serialization
```

Assemblies, Reflection, and Attributes

Assemblies, Reflection, and Attributes

The assemblies into which C# programs compile comprise executable instructions (stored as intermediate language or IL) and metadata, which describes the program's types, members, and attributes. Through reflection, you can inspect

this metadata at runtime, and do such things as dynamically invoke methods. With `Reflection.Emit`, you can construct new code on the fly.

In Chapter 17, we describe the makeup of assemblies and how to sign them, use the global assembly cache and resources, and resolve file references. In Chapter 18, we cover reflection and attributes—describing how to inspect metadata, dynamically invoke functions, write custom attributes, emit new types, and parse raw IL. The types for using reflection and working with assemblies reside in the following namespaces:

```
System
System.Reflection
```

```
System.Reflection
System.Reflection.Emit
```

Dynamic Programming

In Chapter 19, we look at some of the patterns for dynamic programming and leveraging the Dynamic Language Runtime, which is now part of the CLR. We describe how to implement the Visitor pattern, write custom dynamic objects, and interoperate with IronPython. The types for dynamic programming are in `System.Dynamic`.

Security

The .NET Framework provides its own security layer, allowing you to both sandbox other assemblies and be sandboxed yourself. In Chapter 20, we cover code access, role, and identity security, and the new transparency model in CLR 4.0. We then other assemblies and be sandboxed yourself. In Chapter 20, we cover code access, role, and identity security, and the new transparency model in CLR 4.0. We then describe cryptography in the Framework, covering encryption, hashing, and data protection. The types for this are defined in:

```
System.Security
System.Security.Permissions
System.Security.Policy
System.Security.Cryptography
```

Threading and Parallel Programming

Multithreading allows you to execute code in parallel. Chapter 21 explores this subject in detail, describing both the Framework's support for multithreading and the strategies for writing multithreaded applications. In Chapter 22, we cover Framework 4.0's new constructs for leveraging multicore processors, and in Chapter 23, we describe how to use asynchronous methods to write highly concurrent server applications.

we describe how to use asynchronous methods to write highly concurrent server applications.

All types for threading are in and under the `System.Threading` namespace.

Application Domains

The CLR provides an additional level of isolation within a process, called an *application domain*. In Chapter 24, we examine the properties of an application domain with which you can interact, and demonstrate how to create and use additional application domains within the same process for such purposes as unit testing. We

also describe how to use Remoting to communicate with these application domains. The `AppDomain` type is defined in the `System` namespace.

Native and COM Interoperability

You can interoperate with both native and COM code. Native interoperability allows you to call functions in unmanaged DLLs, register callbacks, map data structures, and interoperate with native data types. COM interoperability allows you to call COM types and expose .NET types to COM. The types that support these functions are in `System.Runtime.InteropServices`, and we cover them in Chapter 25.

Applied Technologies

User Interface Technologies

The .NET Framework provides four APIs for user-interface-based applications:

ASP.NET (`System.Web.UI`)
: For writing thin client applications that run over a standard web browser

Silverlight
: For providing a rich user interface inside a web browser

Silverlight
: For providing a rich user interface inside a web browser

Windows Presentation Foundation (`System.Windows`)
: For writing rich client applications

Windows Forms (`System.Windows.Forms`)
: For maintaining legacy rich client applications

In general, a thin client application amounts to a website; a rich client application is a program the end user must download or install on the client computer.

ASP.NET

Applications written using ASP.NET host under Windows IIS and can be accessed from almost any web browser. Here are the advantages of ASP.NET over rich client technologies:

- There is zero deployment at the client end.
- Clients can run a non-Windows platform.
- Updates are easily deployed.

Further, because most of what you write in an ASP.NET application runs on the

Further, because most of what you write in an ASP.NET application runs on the server, you design your data access layer to run in the same application domain—without limiting security or scalability. In contrast, a rich client that does the same is not generally as secure or scalable. (The solution, with the rich client, is to insert a *middle tier* between the client and database. The middle tier runs on a remote application server [often alongside the database server] and communicates with the rich clients via WCF, Web Services, or Remoting.)

In writing your web pages, you can choose between the traditional Web Forms and the new MVC (Model-View-Controller) API. Both build on the ASP.NET

infrastructure. Web Forms has been part of the Framework since its inception; MVC was written much later in response to the success of Ruby on Rails and MonoRail. Although debuting in Framework 4.0, the MVC framework has benefited from having matured for some time as a public beta. It provides, in general, a better programming abstraction than Web Forms; it also allows more control over the generated HTML. What you lose over Web Forms is a designer. This makes Web Forms still a good choice for web pages with predominately static content.

The limitations of ASP.NET are largely a reflection of the limitations of thin client

The limitations of ASP.NET are largely a reflection of the limitations of thin client systems in general:

- A web browser interface significantly restricts what you can do.
- Maintaining state on the client—or on behalf of the client—is cumbersome.

You can improve the interactivity and responsiveness, however, through client-side scripting or technologies such as AJAX: a good resource for this is *http://ajax.asp.net*. Use of AJAX is simplified through the use of libraries such as jQuery.

The types for writing ASP.NET applications are in the `System.Web.UI` namespace and its subnamespaces, and are packed in the *System.Web.dll* assembly.

Silverlight

Silverlight is not technically part of the main .NET Framework: it's a separate Framework that includes a subset of the Framework's core features—plus the ability to run as a web browser plug-in. Its graphics model is essentially a subset of WPF

Framework that includes a subset of the Framework's core features—plus the ability to run as a web browser plug-in. Its graphics model is essentially a subset of WPF and this allows you to leverage existing knowledge in developing Silverlight applications. Silverlight is available as a small cross-platform download for web browsers—much like Macromedia's Flash.

Flash has a far bigger installation base and so dominates in this area. For this reason, Silverlight tends to be used in fringe scenarios—corporate intranets, for example.

Windows Presentation Foundation (WPF)

WPF was introduced in Framework 3.0 for writing rich client applications. The benefits of WPF over its predecessor, Windows Forms, are as follows:

It supports sophisticated graphics, such as arbitrary transformations, 3D

It supports sophisticated graphics, such as arbitrary transformations, 3D rendering, and true transparency.

Its primary measurement unit is not pixel-based, so applications display correctly at any DPI (dots per inch) setting.

It has extensive dynamic layout support, which means you can localize an application without danger of elements overlapping.

Rendering uses DirectX and is fast, taking good advantage of graphics hardware acceleration.

User interfaces can be described declaratively in XAML files that can be maintained independently of the "code-behind" files—this helps to separate appearance from functionality.

WPF's size and complexity, however, make for a steep learning curve.

The types for writing WPF applications are in the `System.Windows` namespace and all subnamespaces except for `System.Windows.Forms`.

Windows Forms

Windows Forms is a rich client API that's as old as the .NET Framework. Compared to WPF, Windows Forms is a relatively simple technology that provides most of the features you need in writing a typical Windows application. It also has significant relevancy in maintaining legacy applications. It has a number of drawbacks, though, compared to WPF:

- Controls are positioned and sized in pixels, making it easy to write applications that break on clients whose DPI settings differ from the developer's.
- The API for drawing nonstandard controls is GDI+, which, although reasonably flexible, is slow in rendering large areas (and without double buffering, flickers horribly).
- Controls lack true transparency.
-

Controls lack true transparency.

Dynamic layout is difficult to get right reliably.

The last point is an excellent reason to favor WPF over Windows Forms—even if you're writing a business application that needs just a user interface and not a "user experience." The layout elements in WPF, such as `Grid`, make it easy to assemble labels and text boxes such that they always align—even after language changing localization—without messy logic and without any flickering. Further, you don't have to bow to the lowest common denominator in screen resolution—WPF layout elements have been designed from the outset to adapt properly to resizing.

On the subject of speed, it was originally thought that graphics card manufacturers would incorporate GDI+ hardware accelerators. This never happened; their focus was instead on DirectX. Consequently, GDI+ is considerably slower than even its predecessor, GDI, let alone WPF.

On the positive side, Windows Forms is relatively simple to learn and has a wealth of support in third-party controls.

The Windows Forms types are in the namespaces `System.Windows.Forms` (in *System.Windows.Forms.dll*) and `System.Drawing` (in *System.Drawing.dll*). The latter also contains the GDI+ types for drawing custom controls.

Backend Technologies

ADO.NET

ADO.NET is the managed data access API. Although the name is derived from the 1990s-era ADO (ActiveX Data Objects), the technology is completely different. ADO.NET contains two major low-level components:

Provider layer

The provider model defines common classes and interfaces for low-level access to database providers. These interfaces comprise connections, commands, adapters, and readers (forward-only, read-only cursors over a database). The Framework ships with native support for Microsoft SQL Server and Oracle and has OLE-DB and ODBC providers.

DataSet model

A DataSet is a structured cache of data. It resembles a primitive in-memory database, which defines SQL constructs such as tables, rows, columns, relationships, constraints, and views. By programming against a cache of data, you can reduce the number of trips to the server, increasing server scalability and the responsiveness of a rich-client user interface. DataSets are serializable and are designed to be sent across the wire between client and server applications.

Sitting above the provider layer are two APIs that offer the ability to query databases via LINQ:

- LINQ to SQL (introduced in Framework 3.5)
- Entity Framework (introduced in Framework 3.5 SP1)

Both technologies include *object/relational mappers* (ORMs), meaning they automatically map objects (based on classes that you define) to rows in the database. This allows you to query those objects via LINQ (instead of writing SQL `select` statements)—and update them without manually writing SQL `insert`/`delete`/`update` statements. This cuts the volume of code in an application's data access layer (particularly the "plumbing" code) and provides strong static type safety. These technologies also avoid the need for DataSets as receptacles of data—although DataSets still provide the unique ability to store and serialize state changes (something particularly useful in multitier applications). You can use LINQ to SQL or Entity Framework in conjunction with DataSets, although the process is somewhat clumsy and DataSets are inherently ungainly. In other words, there's no straight-

Entity Framework in conjunction with DataSets, although the process is somewhat clumsy and DataSets are inherently ungainly. In other words, there's no straightforward out-of-the-box solution for writing *n*-tier applications with Microsoft's ORMs as yet.

LINQ to SQL is simpler and faster than Entity Framework, and tends to produce better SQL. Entity Framework is more flexible in that you can create elaborate mappings between the database and the classes that you query. Entity Framework also has third-party support for databases other than SQL Server.

Windows Workflow

Windows Workflow is a framework for modeling and managing potentially long-running business processes. Workflow targets a standard runtime library, providing consistency and interoperability. Workflow also helps reduce coding for dynamically controlled decision-making trees.

Windows Workflow is not strictly a backend technology—you can use it anywhere (an example is page flow, in the UI).

Workflow came originally with .NET Framework 3.0, with its types defined in the `System.WorkFlow` namespace. Workflow has been substantially revised in Framework 4.0; the new types live in and under the `System.Activities` namespace.

COM+ and MSMQ

The Framework allows you to interoperate with COM+ for services such as distributed transactions, via types in the `System.EnterpriseServices` namespace. It also supports MSMQ (Microsoft Message Queuing) for asynchronous, one-way messaging through types in `System.Messaging`.

Distributed System Technologies

Windows Communication Foundation (WCF)

WCF is a sophisticated communications infrastructure introduced in Framework 3.0. WCF is flexible and configurable enough to make both of its predecessors—Remoting and (.ASMX) Web Services—*mostly* redundant.

3.0. WCF is flexible and configurable enough to make both of its predecessors Remoting and (.ASMX) Web Services—*mostly* redundant.

WCF, Remoting, and Web Services are all alike in that they implement the following basic model in allowing a client and server application to communicate:

-
-
-
-
-

On the server, you indicate what methods you'd like remote client applications to be able to call.

On the client, you specify or infer the *signatures* of the server methods you'd like to call.

On both the server and the client, you choose a transport and communication protocol (in WCF, this is done through a *binding*).

The client establishes a connection to the server.

The client calls a remote method, which executes transparently on the server

The client establishes a connection to the server.

The client calls a remote method, which executes transparently on the server.

FW Overview

WCF further decouples the client and server through service contracts and data contracts. Conceptually, the client sends an (XML or binary) message to an endpoint on a remote *service*, rather than directly invoking a remote *method*. One of the benefits of this decoupling is that clients have no dependency on the .NET platform or on any proprietary communication protocols.

on any proprietary communication protocols.

WCF is highly configurable and provides the most extensive support for standardized messaging protocols, including WS-*. This lets you communicate with parties running different software—possibly on different platforms—while still supporting advanced features such as encryption. Another benefit of WCF is that you can change protocols without needing to change other aspects of your client or server applications.

The types for communicating with WCF are in, and are below, the `System.ServiceModel` namespace.

Remoting and Web Services

Remoting and .ASMX Web Services are WCF's predecessors and are almost redundant in WCF's wake—although Remoting still has a niche in communicating between application domains within the same process (see Chapter 24).

Remoting's functionality is geared toward tightly coupled applications. A typical

Remoting's functionality is geared toward tightly coupled applications. A typical example is when the client and server are both .NET applications written by the same company (or companies sharing common assemblies). Communication typically involves exchanging potentially complex custom .NET objects that the Remoting infrastructure serializes and deserializes without needing intervention.

The functionality of Web Services is geared toward loosely coupled or SOA-style applications. A typical example is a server designed to accept simple SOAP-based messages that originate from clients running a variety of software—on a variety of platforms. Web Services can only use HTTP and SOAP as transport and formatting protocols, and applications are normally hosted under IIS. The benefits of interoperability come at a performance cost—a Web Services application is typically slower, in both execution and development time, than a well-designed Remoting application.

The types for Remoting are in or under `System.Runtime.Remoting`; the types for Web Services are under `System.Web.Services`.

CardSpace

CardSpace is a token-based authentication and identity management protocol designed to simplify password management for end users. The technology has received

CardSpace is a token-based authentication and identity management protocol designed to simplify password management for end users. The technology has received little attention because of the difficulty in porting tokens across machines (*OpenID* is a popular alternative that avoids this problem).

CardSpace builds on open XML standards, and parties can participate independently of Microsoft. A user can hold multiple identities, which are maintained by a third party (the *identity provider*). When a user wants to access a resource at site X, the user authenticates to the identity provider, which then issues a token to site X. This avoids having to provide a password directly to site X and reduces the number of identities that the user needs to maintain.

WCF allows you to specify a CardSpace identity when connecting through a secure HTTP channel, through types in the `System.IdentityModel.Claims` and `System.IdentityModel.Policy` namespaces.

6

Framework Fundamentals

Many of the core facilities that you need when programming are provided not by the C# language, but by types in the .NET Framework. In this chapter, we cover the Framework's role in fundamental programming tasks, such as virtual equality comparison, order comparison, and type conversion. We also cover the basic Framework types, such as `String`, `DateTime`, and `Enum`.

The types in this section reside in the `System` namespace, with the following exceptions:

•
•
•

`StringBuilder` is defined in `System.Text`, as are the types for *text encodings*.

`CultureInfo` and associated types are defined in `System.Globalization`.

`XmlConvert` is defined in `System.Xml`.

String and Text Handling

Char

A C# `char` represents a single Unicode character and aliases the `System.Char` struct. In Chapter 2, we described how to express `char` literals. For example:

```
char c = 'A';
char newLine = '\n';
```

```
char newLine = '\n';
```

`System.Char` defines a range of static methods for working with characters, such as `ToUpper`, `ToLower`, and `IsWhiteSpace`. You can call these through either the `System.Char` type or its `char` alias:

```
Console.WriteLine (System.Char.ToUpper ('c'));    // C
Console.WriteLine (char.IsWhiteSpace ('\t'));     // True
```

`ToUpper` and `ToLower` honor the end user's locale, which can lead to subtle bugs. The following expression evaluates to `false` in Turkey:

```
char.ToUpper ('i') == 'I'
```


because in Turkey, `char.ToUpper ('i')` is `'İ'` (notice the dot on top!). To avoid this problem, `System.Char` (and `System.String`) also provides culture-invariant versions of `ToUpper` and `ToLower` ending with the word *Invariant*. These always apply English culture rules:

```
Console.WriteLine (char.ToUpperInvariant ('i'));
```

```
Console.WriteLine (char.ToUpperInvariant ('i'));
```

```
// I
```

This is a shortcut for:

```
Console.WriteLine (char.ToUpper ('i', CultureInfo.InvariantCulture))
```

For more on locales and culture, see "Formatting and parsing" on page 212.

Most of `char`'s remaining static methods are related to categorizing characters and are listed in Table 6-1.

Table 6-1. Static methods for categorizing characters

Static method	Characters included	Unicode categories included

`IsLetter`	A–Z, a–z, and letters of other alphabets	`UpperCaseLetter`
		`LowerCaseLetter`
		`TitleCaseLetter`
		`ModifierLetter`

		ModifierLetter OtherLetter
IsUpper	Uppercase letters	UpperCaseLetter
IsLower	Lowercase letters	LowerCaseLetter
IsDigit	0–9 plus digits of other alphabets	DecimalDigitNumber
IsLetterOrDigit	Letters plus digits	Sum of IsLetter and IsDigit
IsNumber	All digits plus Unicode fractions and Roman numeral symbols	DecimalDigitNumber LetterNumber OtherNumber
IsSeparator	Space plus all Unicode separator characters	LineSeparator ParagraphSeparator
IsWhiteSpace	All separators plus \n, \r, \t, \f, and \v	LineSeparator ParagraphSeparator
IsPunctuation	Symbols used for punctuation in Western and other alphabets	DashPunctuation ConnectorPunctuation InitialQuotePunctuation FinalQuotePunctuation

		`FinalQuotePunctuation`

`IsSymbol`	Most other printable symbols	`MathSymbol`
		`ModifierSymbol`
		`OtherSymbol`

Static method	Characters included	Unicode categories included
`IsControl`	Nonprintable "control" characters below 0x20, such as `\r`, `\n`, `\t`, `\0`, and characters between 0x7F and 0x9A	(None)

For more granular categorization, `char` provides a static method called `GetUnicodeCategory`; this returns a `UnicodeCategory` enumeration whose members are shown in the rightmost column of Table 6-1.

By explicitly casting from an integer, it's possible to produce a `char` outside the allocated Unicode set. To test a character's validity, call `char.GetUnicodeCategory`: if the result is `UnicodeCategory.OtherNotAssigned`, the character is invalid.

validity, call `char.GetUnicodeCategory`: if the result is `UnicodeCategory.OtherNotAssigned`, the character is invalid.

A `char` is 16 bits wide—enough to represent any Unicode character in the *Basic Multilingual Plane*. To go outside this, you must use surrogate pairs: we describe the methods for doing this in "Text Encodings and Unicode" on page 203.

String

A C# `string` (`== System.String`) is an immutable (unchangeable) sequence of characters. In Chapter 2, we described how to express string literals, perform equality comparisons, and concatenate two strings. This section covers the remaining functions for working with strings, exposed through the static and instance members of the `System.String` class.

Constructing strings

The simplest way to construct a string is to assign a literal, as we saw in Chapter 2:

The simplest way to construct a string is to assign a literal, as we saw in Chapter 2:

```
string s1 = "Hello";
string s2 = "First Line\r\nSecond Line";
string s3 = @"\\server\fileshare\helloworld.cs";
```

To create a repeating sequence of characters you can use `string`'s constructor:

```
Console.Write (new string ('*', 10));      // **********
```

You can also construct a string from a `char` array. The `ToCharArray` method does the reverse:

```
char[] ca = "Hello".ToCharArray();
string s = new string (ca);

// s = "Hello"
```

`string`'s constructor is also overloaded to accept various (unsafe) pointer types, in order to create strings from types such as `char*`.

Null and empty strings

An empty string has a length of zero. To create an empty string, you can use either a literal or the static `string.Empty` field; to test for an empty string, you can either perform an equality comparison or test its `Length` property:

a literal or the static `string.Empty` field; to test for an empty string, you can either perform an equality comparison or test its `Length` property:


```
string empty = "";
Console.WriteLine (empty == "");
Console.WriteLine (empty == string.Empty);
Console.WriteLine (empty.Length == 0);
```

```
// True
// True
// True
```

Because strings are reference types, they can also be `null`:

```
string nullString = null;
```

```
string nullString = null;
Console.WriteLine (nullString == null);
Console.WriteLine (nullString == "");
Console.WriteLine (nullString.Length == 0);
```

```
// True
// False
// NullReferenceException
```

The static `string.IsNullOrEmpty` method is a useful shortcut for testing whether a given string is either null or empty.

Accessing characters within a string

A string's indexer returns a single character at the given index. As with all functions

A string's indexer returns a single character at the given index. As with all functions that operate on strings, this is zero-indexed:

```
string str  = "abcde";
char letter = str[1];
```

```
// letter == 'b'
```

`string` also implements `IEnumerable<char>`, so you can `foreach` over its characters:

```
foreach (char c in "123") Console.Write (c + ",");    // 1,2,3,
```

Searching within strings

The simplest methods for searching within strings are `Contains`, `StartsWith`, and `EndsWith`. These all return `true` or `false`:

```
Console.WriteLine ("quick brown fox".Contains ("brown"));
Console.WriteLine ("quick brown fox".EndsWith ("fox"));
```

```
Console.WriteLine ("quick brown fox".Contains ("brown"));
Console.WriteLine ("quick brown fox".EndsWith ("fox"));
```

```
// True
// True
```

IndexOf is more powerful: it returns the first position of a given character or substring (or -1 if the substring isn't found):

```
Console.WriteLine ("abcde".IndexOf ("cd"));   // 2
```

IndexOf is overloaded to accept a startPosition (an index from which to begin searching) and a StringComparison enum. The latter allows you to perform case-insensitive searches:

```
Console.WriteLine ("abcde".IndexOf ("CD",
                   StringComparison.CurrentCultureIgnoreCase));
```

```
// 2
```

LastIndexOf is like IndexOf, but works backward through the string.

IndexOfAny returns the first matching position of any one of a set of characters:

```
Console.Write ("ab,cd ef".IndexOfAny (new char[] {' ', ','} ));       // 2
Console.Write ("pas5w0rd".IndexOfAny ("0123456789".ToCharArray() ));  // 3
```

LastIndexOfAny does the same in the reverse direction.

Manipulating strings

Because String is immutable, all the methods that "manipulate" a string return a new one, leaving the original untouched (the same goes for when you reassign a string variable).

Substring extracts a portion of a string:

```
string left3 = "12345".Substring (0, 3);     // left3 = "123";
string mid3  = "12345".Substring (1, 3);     // mid3 = "234";
```

If you omit the length, you get the remainder of the string:

```
string end3  = "12345".Substring (2);        // end3 = "345";
```

`Insert` and `Remove` insert or remove characters at a specified position:

```
string s1 = "helloworld".Insert (5, ", ");    // s1 = "hello, world"
string s2 = s1.Remove (5, 2);                 // s2 = "helloworld";
```

`PadLeft` and `PadRight` pad a string to a given length with a specified character (or a space if unspecified):

```
Console.WriteLine ("12345".PadLeft (9, '*'));  // ****12345
Console.WriteLine ("12345".PadLeft (9));       //     12345
```

If the input string is longer than the padding length, the original string is returned unchanged.

`TrimStart` and `TrimEnd` remove specified characters from the beginning or end of a string; `Trim` does both. By default, these functions remove whitespace characters (including spaces, tabs, new lines, and Unicode variations of these):

```
Console.WriteLine ("  abc \t\r\n ".Trim().Length);   // 3
```

```
Console.WriteLine ("  abc \t\r\n ".Trim().Length);   // 3
```

Replace replaces all occurrences of a particular character or substring:

```
Console.WriteLine ("to be done".Replace (" ", " | ") );  // to | be | done
Console.WriteLine ("to be done".Replace (" ", "")    );  // tobedone
```

ToUpper and ToLower return upper- and lowercase versions of the input string. By default, they honor the user's current language settings; ToUpperInvariant and ToLowerInvariant always apply English alphabet rules.

Splitting and joining strings

`Split` takes a sentence and returns an array of words:

```
string[] words = "The quick brown fox".Split();
```

```
foreach (string word in words)
  Console.Write (word + "|");
```

```
// The|quick|brown|fox|
```

By default, `Split` uses whitespace characters as delimiters; it's also overloaded to accept a `params` array of `char` or `string` delimiters. `Split` also optionally accepts a `StringSplitOptions` enum, which has an option to remove empty entries: this is useful when words are separated by several delimiters in a row.

useful when words are separated by several delimiters in a row.

The static Join method does the reverse of Split. It requires a delimiter and string array:

```
string[] words = "The quick brown fox".Split();
string together = string.Join (" ", words);      // The quick brown fox
```

The static Concat method is similar to Join but accepts only a params string array and applies no separator. Concat is exactly equivalent to the + operator (the compiler, in fact, translates + to Concat):

```
string sentence     = string.Concat ("The", " quick", " brown", " fox");
string sameSentence = "The" + " quick" + " brown" + " fox";
```

String.Format and composite format strings

The static Format method provides a convenient way to build strings that embed variables. The embedded variables can be of any type; the Format simply calls ToString on them.

variables. The embedded variables can be of any type; the Format simply calls `ToString` on them.

The master string that includes the embedded variables is called a *composite format string*. When calling `String.Format`, you provide a composite format string followed by each of the embedded variables. For example:

```
string composite = "It's {0} degrees in {1} on this {2} morning";
string s = string.Format (composite, 35, "Perth", DateTime.Now.DayOfWeek);
```

```
// s == "It's 35 degrees in Perth on this Friday morning"
```

(And that's Celsius!)

Each number in curly braces is called a *format item*. The number corresponds to the argument position and is optionally followed by:

-
-

A comma and a *minimum width* to apply

A comma and a *minimum width* to apply

A colon and a *format string*

The minimum width is useful for aligning columns. If the value is negative, the data is left-aligned; otherwise, it's right-aligned. For example:

```
string composite = "Name={0,-20} Credit Limit={1,15:C}";
```

```
Console.WriteLine (string.Format (composite, "Mary", 500));
Console.WriteLine (string.Format (composite, "Elizabeth", 20000));
```

Here's the result:

```
Name=Mary
Name=Elizabeth
```

```
Credit Limit=
Credit Limit=
```

```
Credit Limit=
    $500.00
 $20,000.00
```

The equivalent without using `string.Format` is this:

```
string s = "Name=" + "Mary".PadRight (20) +
           " Credit Limit=" + 500.ToString ("C").PadLeft (15);
```

The credit limit is formatted as currency by virtue of the `"C"` format string. We describe format strings in detail in "Formatting and parsing" on page 212.

A disadvantage of composite format strings is that it's easier to make a mistake that goes unnoticed until runtime—such as having greater or fewer format items than values. Such a mistake is harder to make when the format items and values are together.

is harder to make when the format items and values are together.

Comparing Strings

In comparing two values, the .NET Framework differentiates the concepts of *equality comparison* and *order comparison*. Equality comparison tests whether two instances are semantically the same; order comparison tests which of two (if any) instances comes first when arranging them in ascending or descending sequence.

> Equality comparison is not a *subset* of order comparison; the two systems have different purposes. It's legal, for instance, to have two unequal values in the same ordering position. We resume this topic in "Equality Comparison" on page 245.

For string equality comparison, you can use the `==` operator or one of `string`'s `Equals` methods. The latter are more versatile because they allow you to specify options such as case-insensitivity.