Chapter 6. .NET Fundamentals

Many of the core facilities that you need when programming are provided not by the C# language but by types in the .NET BCL. In this chapter, we cover types that help with fundamental programming tasks, such as virtual equality comparison, order comparison, and type conversion. We also cover the basic .NET 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:

```
char c = 'A';
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'
```

The reason is 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'));    // I
```

This is a shortcut for:

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

For more on locales and culture, see "Formatting and Parsing".

Most of `char`'s remaining static methods are related to categorizing characters. Table 6-1 lists these.

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` `OtherLetter`
`IsUpper`	Uppercase letters	`UpperCaseLetter`
`IsLower`	Lowercase letters	`LowerCaseLetter`
`IsDigit`	0–9 plus digits of other alphabets	`DecimalDigitNumber`
`IsLetterOrDigit`	Letters plus digits	(`IsLetter`, `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`

Static method	**Characters included**	**Unicode categories included**
`IsPunctuation`	Symbols used for punctuation in Western and other alphabets	`DashPunctuation` `ConnectorPunctuation` `InitialQuotePunctuation` `FinalQuotePunctuation`
`IsSymbol`	Most other printable symbols	`MathSymbol` `ModifierSymbol` `OtherSymbol`
`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.

NOTE

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.

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

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:

```
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:

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

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

```
string nullString = null;
Console.WriteLine (nullString == null);        // True
Console.WriteLine (nullString == "");          // False
Console.WriteLine (nullString.Length == 0);    // 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 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 `StartsWith`, `EndsWith`, and `Contains`. These all return `true` or `false`:

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

These methods are overloaded to let you specify a `StringComparison` enum to control case and culture sensitivity (see "Ordinal versus culture comparison"). The default is to perform a case-sensitive match using rules applicable to the current (localized) culture. The following instead performs a case-insensitive search using the *invariant* culture's rules:

```
"abcdef".StartsWith ("aBc", StringComparison.InvariantCultureIgnoreCase)
```

`IndexOf` 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 also overloaded to accept a `startPosition` (an index from which to begin searching) as well as a `StringComparison` enum:

```
Console.WriteLine ("abcde abcde".IndexOf ("CD", 6,
                   StringComparison.CurrentCultureIgnoreCase));    // 8
```

`LastIndexOf` is like `IndexOf`, but it 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
```

`Replace` replaces all (non-overlapping) 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 uppercase 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` divides a string into pieces:

```
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.

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 (or values) 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:

```
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!)

We can use interpolated string literals to the same effect (see “String Type”). Just precede the string with the `$` symbol and put the expressions in braces:

```
string s = $"It's hot this {DateTime.Now.DayOfWeek} morning";
```

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 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:

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

Here’s the equivalent without using `string.Format`:

```
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”.

Comparing Strings

In comparing two values, .NET 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.

NOTE

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".

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.

WARNING

Another difference is that `==` does not work reliably on strings if the variables are cast to the `object` type. We explain why this is so in "Equality Comparison".

For string order comparison, you can use either the `CompareTo` instance method or the static `Compare` and `CompareOrdinal` methods. These return a positive or negative number, or zero, depending on whether the first value comes after, before, or alongside the second.

Before going into the details of each, we need to examine .NET's underlying string comparison algorithms.

Ordinal versus culture comparison

There are two basic algorithms for string comparison: *ordinal* and *culture sensitive*. Ordinal comparisons interpret characters simply as numbers (according to their numeric Unicode value); culture-sensitive comparisons interpret characters with reference to a particular alphabet. There are two special cultures: the “current culture,” which is based on settings picked up from the computer’s control panel, and the “invariant culture,” which is the same on every computer (and closely matches American culture).

For equality comparison, both ordinal and culture-specific algorithms are useful. For ordering, however, culture-specific comparison is nearly always preferable: to order strings alphabetically, you need an alphabet. Ordinal relies on the numeric Unicode point values, which happen to put English characters in alphabetical order—but even then, not exactly as you might expect. For example, assuming case sensitivity, consider the strings `"Atom"`, `"atom"`, and `"Zamia"`. The invariant culture puts them in the following order:

```
"atom", "Atom", "Zamia"
```

Ordinal arranges them instead as follows:

```
"Atom", "Zamia", "atom"
```

This is because the invariant culture encapsulates an alphabet, which considers uppercase characters adjacent to their lowercase counterparts (aAbBcCdD...). The ordinal algorithm, however, puts all the uppercase characters first, and then all lowercase characters (A...Z, a...z). This is essentially a throwback to the ASCII character set invented in the 1960s.

String equality comparison

Despite ordinal’s limitations, `string`’s `==` operator always performs *ordinal case-sensitive* comparison. The same goes for the instance version of `string.Equals` when called without arguments; this defines the “default” equality comparison behavior for the `string` type.

NOTE

The ordinal algorithm was chosen for `string`’s == and `Equals` functions because it’s both highly efficient and *deterministic*. String equality comparison is considered fundamental and is performed far more frequently than order comparison.

A “strict” notion of equality is also consistent with the general use of the == operator.

The following methods allow culture-aware or case-insensitive comparisons:

```
public bool Equals (string value, StringComparison comparisonType);

public static bool Equals (string a, string b,
                           StringComparison comparisonType);
```

The static version is advantageous in that it still works if one or both of the strings are `null`. `StringComparison` is an `enum` defined as follows:

```
public enum StringComparison
{
  CurrentCulture,              // Case-sensitive
  CurrentCultureIgnoreCase,
  InvariantCulture,            // Case-sensitive
  InvariantCultureIgnoreCase,
  Ordinal,                     // Case-sensitive
  OrdinalIgnoreCase
}
```

For example:

```
Console.WriteLine (string.Equals ("foo", "FOO",
                   StringComparison.OrdinalIgnoreCase));   // True

Console.WriteLine ("ṻ" == "ǖ");                            // False

Console.WriteLine (string.Equals ("ṻ", "ǖ",
                   StringComparison.CurrentCulture));      // ?
```

(The result of the third example is determined by the computer's current language settings.)

String order comparison

`String`'s `CompareTo` instance method performs *culture-sensitive*, *case-sensitive* order comparison. Unlike the `==` operator, `CompareTo` does not use ordinal comparison: for ordering, a culture-sensitive algorithm is much more useful. Here's the method's definition:

```
public int CompareTo (string strB);
```

NOTE

The `CompareTo` instance method implements the generic `IComparable` interface, a standard comparison protocol used across the .NET libraries. This means `string`'s `CompareTo` defines the default ordering behavior of strings in such applications as sorted collections, for instance. For more information on `IComparable`, see "Order Comparison".

For other kinds of comparison, you can call the static `Compare` and `CompareOrdinal` methods:

```
public static int Compare (string strA, string strB,
                           StringComparison comparisonType);

public static int Compare (string strA, string strB, bool ignoreCase,
                           CultureInfo culture);

public static int Compare (string strA, string strB, bool ignoreCase);

public static int CompareOrdinal (string strA, string strB);
```

The last two methods are simply shortcuts for calling the first two methods.

All of the order comparison methods return a positive number, a negative number, or zero depending on whether the first value comes after, before, or alongside the second value:

```
Console.WriteLine ("Boston".CompareTo ("Austin"));    // 1
Console.WriteLine ("Boston".CompareTo ("Boston"));    // 0
Console.WriteLine ("Boston".CompareTo ("Chicago"));   // -1
Console.WriteLine ("Ṻ".CompareTo ("ǖ"));              // 1
Console.WriteLine ("foo".CompareTo ("FOO"));          // -1
```

The following performs a case-insensitive comparison using the current culture:

```
Console.WriteLine (string.Compare ("foo", "FOO", true));   // 0
```

By supplying a `CultureInfo` object, you can plug in any alphabet:

```
// CultureInfo is defined in the System.Globalization namespace

CultureInfo german = CultureInfo.GetCultureInfo ("de-DE");
int i = string.Compare ("Müller", "Muller", false, german);
```

StringBuilder

The `StringBuilder` class (`System.Text` namespace) represents a mutable (editable) string. With a `StringBuilder`, you can `Append`, `Insert`, `Remove`, and `Replace` substrings without replacing the whole `StringBuilder`.

`StringBuilder`'s constructor optionally accepts an initial string value as well as a starting size for its internal capacity (default is 16 characters). If you go beyond this, `StringBuilder` automatically resizes its internal structures to accommodate (at a slight performance cost) up to its maximum capacity (default is `int.MaxValue`).

A popular use of `StringBuilder` is to build up a long string by repeatedly calling `Append`. This approach is much more efficient than repeatedly concatenating ordinary string types:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 50; i++) sb.Append(i).Append(",");
```

To get the final result, call `ToString()`:

```
Console.WriteLine (sb.ToString());

0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
```

`AppendLine` performs an `Append` that adds a new line sequence (`"\r\n"` in Windows). `AppendFormat` accepts a composite format string, just like `String.Format`.

In addition to the `Insert`, `Remove`, and `Replace` methods (`Replace` works like string's `Replace`), `StringBuilder` defines a `Length` property and a writable indexer for getting/setting individual characters.

To clear the contents of a `StringBuilder`, either instantiate a new one or set its `Length` to zero.

WARNING

Setting a `StringBuilder`'s `Length` to zero doesn't shrink its *internal* capacity. So, if the `StringBuilder` previously contained one million characters, it will continue to occupy around two megabytes of memory after zeroing its `Length`. If you want to release the memory, you must create a new `StringBuilder` and allow the old one to drop out of scope (and be garbage-collected).

Text Encodings and Unicode

A *character set* is an allocation of characters, each with a numeric code, or *code point*. There are two character sets in common use: Unicode and ASCII. Unicode has an address space of approximately one million characters, of which about 100,000 are currently allocated. Unicode covers most spoken world languages as well as some historical languages and special symbols. The ASCII set is simply the first 128 characters of the Unicode set, which covers most of what you see on a US-style keyboard. ASCII predates Unicode by 30 years and is still sometimes used for its simplicity and efficiency: each character is represented by one byte.

The .NET type system is designed to work with the Unicode character set. ASCII is implicitly supported, though, by virtue of being a subset of Unicode.

A *text encoding* maps characters from their numeric code point to a binary representation. In .NET, text encodings come into play primarily when dealing with text files or streams. When you read a text file into a string, a *text encoder* translates the file data from binary into the internal Unicode representation that the `char` and `string` types expect. A text encoding can restrict what characters can be represented as well as affect storage efficiency.

There are two categories of text encoding in .NET:

- Those that map Unicode characters to another character set
- Those that use standard Unicode encoding schemes

The first category contains legacy encodings such as IBM's EBCDIC and 8-bit character sets with extended characters in the upper-128 region that were popular prior to Unicode (identified by a code page). The ASCII encoding is also in this category: it encodes the first 128 characters and drops everything else. This category contains the *nonlegacy* GB18030, as well, which is the mandatory standard for applications written in China—or sold to China—since 2000.

In the second category are UTF-8, UTF-16, and UTF-32 (and the obsolete UTF-7). Each differs in space efficiency. UTF-8 is the most space-efficient for most kinds of text: it uses *between one and four bytes* to represent each character. The first 128 characters require only a single byte, making it compatible with ASCII. UTF-8 is the most popular encoding for text files and streams (particularly on the internet), and it is the default for stream input/output (I/O) in .NET (in fact, it's the default for almost everything that implicitly uses an encoding).

UTF-16 uses one or two 16-bit words to represent each character. This is what .NET uses internally to represent characters and strings. Some programs also write files in UTF-16.

UTF-32 is the least space-efficient: it maps each code point directly to 32 bits, so every character consumes four bytes. UTF-32 is rarely used for this reason. It does, however, make random access very easy because every character takes an equal number of bytes.

Obtaining an Encoding object

The `Encoding` class in `System.Text` is the common base type for classes that encapsulate text encodings. There are several subclasses—their purpose is to encapsulate families of encodings with similar features. The most common encodings can be obtained through dedicated static properties on `Encoding`:

Encoding name	Static property on Encoding
UTF-8	`Encoding.UTF8`
UTF-16	`Encoding.Unicode` (*not* `UTF16`)
UTF-32	`Encoding.UTF32`
ASCII	`Encoding.ASCII`

You can obtain other encodings by calling `Encoding.GetEncoding` with a standard Internet Assigned Numbers Authority (IANA) Character Set name:

```
// In .NET 5+ and .NET Core, you must first call RegisterProvider:
Encoding.RegisterProvider (CodePagesEncodingProvider.Instance);

Encoding chinese = Encoding.GetEncoding ("GB18030");
```

The static `GetEncodings` method returns a list of all supported encodings along with their standard IANA names:

```
foreach (EncodingInfo info in Encoding.GetEncodings())
  Console.WriteLine (info.Name);
```

The other way to obtain an encoding is to directly instantiate an encoding class. Doing so allows you to set various options via constructor arguments, including:

- Whether to throw an exception if an invalid byte sequence is encountered when decoding. The default is false.
- Whether to encode/decode UTF-16/UTF-32 with the most significant bytes first (*big endian*) or the least significant bytes first (*little endian*). The default is *little endian*, the standard on the Windows operating system.
- Whether to emit a byte-order mark (a prefix that indicates *endianness*).

Encoding for file and stream I/O

The most common application for an `Encoding` object is to control how text is read and written to a file or stream. For example, the following writes "Testing…" to a file called *data.txt* in UTF-16 encoding:

```
System.IO.File.WriteAllText ("data.txt", "Testing...", Encoding.Unicode);
```

If you omit the final argument, `WriteAllText` applies the ubiquitous UTF-8 encoding.

NOTE

UTF-8 is the default text encoding for all file and stream I/O.

We resume this subject in Chapter 15, in "Stream Adapters".

Encoding to byte arrays

You can also use an `Encoding` object to go to and from a byte array. The `GetBytes` method converts from `string` to `byte[]` with the given encoding; `GetString` converts from `byte[]` to `string`:

```
byte[] utf8Bytes  = System.Text.Encoding.UTF8.GetBytes    ("0123456789");
byte[] utf16Bytes = System.Text.Encoding.Unicode.GetBytes ("0123456789");
byte[] utf32Bytes = System.Text.Encoding.UTF32.GetBytes   ("0123456789");

Console.WriteLine (utf8Bytes.Length);    // 10
Console.WriteLine (utf16Bytes.Length);   // 20
Console.WriteLine (utf32Bytes.Length);   // 40

string original1 = System.Text.Encoding.UTF8.GetString    (utf8Bytes);
string original2 = System.Text.Encoding.Unicode.GetString (utf16Bytes);
string original3 = System.Text.Encoding.UTF32.GetString   (utf32Bytes);

Console.WriteLine (original1);          // 0123456789
Console.WriteLine (original2);          // 0123456789
Console.WriteLine (original3);          // 0123456789
```

UTF-16 and surrogate pairs

Recall that .NET stores characters and strings in UTF-16. Because UTF-16 requires one or two 16-bit words per character, and a `char` is only 16 bits in length, some Unicode characters require two `chars` to represent. This has a couple of consequences:

- A string's `Length` property can be greater than its real character count.
- A single `char` is not always enough to fully represent a Unicode character.

Most applications ignore this because nearly all commonly used characters fit into a section of Unicode called the *Basic Multilingual Plane* (BMP), which requires only one 16-bit word in UTF-16. The BMP covers several dozen world languages and includes more than 30,000 Chinese characters. Excluded are characters of some ancient languages, symbols for musical notation, some less common Chinese characters, and most emojis.

If you need to support two-word characters, the following static methods in `char` convert a 32-bit code point to a string of two `char`s, and back again:

```
string ConvertFromUtf32 (int utf32)
int    ConvertToUtf32   (char highSurrogate, char lowSurrogate)
```

Two-word characters are called *surrogates*. They are easy to spot because each word is in the range 0xD800 to 0xDFFF. You can use the following static methods in `char` to assist:

```
bool IsSurrogate     (char c)
bool IsHighSurrogate (char c)
bool IsLowSurrogate  (char c)
bool IsSurrogatePair (char highSurrogate, char lowSurrogate)
```

The `StringInfo` class in the `System.Globalization` namespace also provides a range of methods and properties for working with two-word characters.

Characters outside the BMP typically require special fonts and have limited operating system support.

Dates and Times

The following immutable structs in the `System` namespace do the job of representing dates and times:

`DateTime`, `DateTimeOffset`, `TimeSpan`, `DateOnly`, `TimeOnly`

C# doesn’t define any special keywords that map to these types.

TimeSpan

A `TimeSpan` represents an interval of time—or a time of the day. In the latter role, it’s simply the “clock” time (without the date), which is equivalent to the time since midnight, assuming no daylight saving transition. A `TimeSpan` has a resolution of 100 ns, has a maximum value of about 10 million days, and can be positive or negative.

There are three ways to construct a `TimeSpan`:

- Through one of the constructors
- By calling one of the static `From`… methods
- By subtracting one `DateTime` from another

Here are the constructors:

```
public TimeSpan (int hours, int minutes, int seconds);
public TimeSpan (int days, int hours, int minutes, int seconds);
public TimeSpan (int days, int hours, int minutes, int seconds,
                                                   int milliseconds);
public TimeSpan (int days, int hours, int minutes, int seconds,
                                      int milliseconds, int microseconds);
public TimeSpan (long ticks);   // Each tick = 100ns
```

The static `From`… methods are more convenient when you want to specify an interval in just a single unit, such as minutes, hours, and so on:

```
public static TimeSpan FromDays (double value);
public static TimeSpan FromHours (double value);
public static TimeSpan FromMinutes (double value);
public static TimeSpan FromSeconds (double value);
public static TimeSpan FromMilliseconds (double value);
public static TimeSpan FromMicroseconds (double value);
```

For example:

```
Console.WriteLine (new TimeSpan (2, 30, 0));     // 02:30:00
Console.WriteLine (TimeSpan.FromHours (2.5));    // 02:30:00
Console.WriteLine (TimeSpan.FromHours (-2.5));   // -02:30:00
```

`TimeSpan` overloads the `<` and `>` operators as well as the `+` and `-` operators. The following expression evaluates to a `TimeSpan` of 2.5 hours:

```
TimeSpan.FromHours(2) + TimeSpan.FromMinutes(30);
```

The next expression evaluates to one second short of 10 days:

```
TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1);   // 9.23:59:59
```

Using this expression, we can illustrate the integer properties `Days`, `Hours`, `Minutes`, `Seconds`, and `Milliseconds`:

```
TimeSpan nearlyTenDays = TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1);

Console.WriteLine (nearlyTenDays.Days);          // 9
Console.WriteLine (nearlyTenDays.Hours);         // 23
Console.WriteLine (nearlyTenDays.Minutes);       // 59
Console.WriteLine (nearlyTenDays.Seconds);       // 59
Console.WriteLine (nearlyTenDays.Milliseconds);  // 0
```

In contrast, the `Total`... properties return values of type `double` describing the entire time span:

```
Console.WriteLine (nearlyTenDays.TotalDays);          // 9.99998842592593
Console.WriteLine (nearlyTenDays.TotalHours);         // 239.999722222222
Console.WriteLine (nearlyTenDays.TotalMinutes);       // 14399.9833333333
Console.WriteLine (nearlyTenDays.TotalSeconds);       // 863999
Console.WriteLine (nearlyTenDays.TotalMilliseconds);  // 863999000
```

The static `Parse` method does the opposite of `ToString`, converting a string to a `TimeSpan`. `TryParse` does the same but returns `false` rather than throwing an exception if the conversion fails. The `XmlConvert` class also provides `TimeSpan`/string conversion methods that follow standard XML formatting protocols.

The default value for a `TimeSpan` is `TimeSpan.Zero`.

`TimeSpan` can also be used to represent the time of the day (the elapsed time since midnight). To obtain the current time of day, call `DateTime.Now.TimeOfDay`.

DateTime and DateTimeOffset

`DateTime` and `DateTimeOffset` are immutable structs for representing a date and, optionally, a time. They have a resolution of 100 ns and a range covering the years 0001 through 9999.

`DateTimeOffset` is functionally similar to `DateTime`. Its distinguishing feature is that it also stores a Coordinated Universal Time (UTC) offset; this allows more meaningful results when comparing values across different time zones.

Choosing between DateTime and DateTimeOffset

`DateTime` and `DateTimeOffset` differ in how they handle time zones. A `DateTime` incorporates a three-state flag indicating whether the `DateTime` is relative to the following:

- The local time on the current computer
- UTC (the modern equivalent of Greenwich Mean Time)
- Unspecified

A `DateTimeOffset` is more specific—it stores the offset from UTC as a `TimeSpan`:

```
July 01 2019 03:00:00 -06:00
```

This influences equality comparisons, which is the main factor in choosing between `DateTime` and `DateTimeOffset`. Specifically:

- `DateTime` ignores the three-state flag in comparisons and considers two values equal if they have the same year, month, day, hour, minute, and so on.
- `DateTimeOffset` considers two values equal if they refer to the *same point in time*.

WARNING

Daylight Saving Time can make this distinction important even if your application doesn't need to handle multiple geographic time zones.

So, `DateTime` considers the following two values different, whereas `DateTimeOffset` considers them equal:

```
July 01 2019 09:00:00 +00:00 (GMT)
July 01 2019 03:00:00 -06:00 (local time, Central America)
```

In most cases, `DateTimeOffset`'s equality logic is preferable. For example, in calculating which of two international events is more recent, a `DateTimeOffset` implicitly gives the correct answer. Similarly, a hacker plotting a Distributed Denial of Service attack would reach for a `DateTimeOffset`! To do the same with `DateTime` requires standardizing on a single time zone (typically UTC) throughout your application. This is problematic for two reasons:

- To be friendly to the end user, UTC `DateTime`s require explicit conversion to local time prior to formatting.
- It's easy to forget and incorporate a local `DateTime`.

`DateTime` is better, though, at specifying a value relative to the local computer at runtime—for example, if you want to schedule an archive at each of your international offices for next Sunday, at 3 A.M. local time (when there's least activity). Here, `DateTime` would be more suitable because it would respect each site's local time.

NOTE

Internally, `DateTimeOffset` uses a short integer to store the UTC offset in minutes. It doesn't store any regional information, so there's nothing present to indicate whether an offset of +08:00, for instance, refers to Singapore time or Perth time.

We revisit time zones and equality comparison in more depth in "Dates and Time Zones".

NOTE

SQL Server 2008 introduced direct support for `DateTimeOffset` through a new data type of the same name.

Constructing a DateTime

`DateTime` defines constructors that accept integers for the year, month, and day—and optionally, the hour, minute, second, millisecond (and microsecond, from .NET 7):

```
public DateTime (int year, int month, int day);

public DateTime (int year, int month, int day,
                 int hour, int minute, int second, int millisecond);
```

If you specify only a date, the time is implicitly set to midnight (0:00).

The `DateTime` constructors also allow you to specify a `DateTimeKind`—an enum with the following values:

```
Unspecified, Local, Utc
```

This corresponds to the three-state flag described in the preceding section. `Unspecified` is the default, and it means that the `DateTime` is time-zone-agnostic. `Local` means relative to the local time zone on the current computer. A local `DateTime` does not include information about *which particular time zone* it refers to, or, unlike `DateTimeOffset`, the numeric offset from UTC.

A `DateTime`’s `Kind` property returns its `DateTimeKind`.

`DateTime`’s constructors are also overloaded to accept a `Calendar` object, as well. This allows you to specify a date using any of the `Calendar` subclasses defined in `System.Globalization`:

```
DateTime d = new DateTime (5767, 1, 1,
                           new System.Globalization.HebrewCalendar());

Console.WriteLine (d);    // 12/12/2006 12:00:00 AM
```

(The formatting of the date in this example depends on your computer’s control panel settings.) A `DateTime` always uses the default Gregorian calendar—this example, a one-time conversion, takes place during construction. To perform computations using another calendar, you must use the methods on the `Calendar` subclass itself.

You can also construct a `DateTime` with a single *ticks* value of type `long`, where *ticks* is the number of 100-ns intervals from midnight 01/01/0001.

For interoperability, `DateTime` provides the static `FromFileTime` and `FromFileTimeUtc` methods for converting from a Windows file time (specified as a `long`) and `FromOADate` for converting from an OLE automation date/time (specified as a `double`).

To construct a `DateTime` from a string, call the static `Parse` or `ParseExact` method. Both methods accept optional flags and format providers; `ParseExact` also accepts a format string. We discuss parsing in greater detail in “Formatting and Parsing”.

Constructing a DateTimeOffset

`DateTimeOffset` has a similar set of constructors. The difference is that you also specify a UTC offset as a `TimeSpan`:

```
public DateTimeOffset (int year, int month, int day,
                       int hour, int minute, int second,
                       TimeSpan offset);

public DateTimeOffset (int year, int month, int day,
                       int hour, int minute, int second, int millisecond,
                       TimeSpan offset);
```

The `TimeSpan` must amount to a whole number of minutes; otherwise, an exception is thrown.

`DateTimeOffset` also has constructors that accept a `Calendar` object, a `long` *ticks* value, and static `Parse` and `ParseExact` methods that accept a string.

You can construct a `DateTimeOffset` from an existing `DateTime` either by using these constructors:

```
public DateTimeOffset (DateTime dateTime);
public DateTimeOffset (DateTime dateTime, TimeSpan offset);
```

or with an implicit cast:

```
DateTimeOffset dt = new DateTime (2000, 2, 3);
```

NOTE

The implicit cast from `DateTime` to `DateTimeOffset` is handy because most of the .NET BCL supports `DateTime`—not `DateTimeOffset`.

If you don't specify an offset, it's inferred from the `DateTime` value using these rules:

- If the `DateTime` has a `DateTimeKind` of `Utc`, the offset is zero.
- If the `DateTime` has a `DateTimeKind` of `Local` or `Unspecified` (the default), the offset is taken from the current local time zone.

To convert in the other direction, `DateTimeOffset` provides three properties that return values of type `DateTime`:

- The `UtcDateTime` property returns a `DateTime` in UTC time.
- The `LocalDateTime` property returns a `DateTime` in the current local time zone (converting it if necessary).
- The `DateTime` property returns a `DateTime` in whatever zone it was specified, with a `Kind` of `Unspecified` (i.e., it returns the UTC time plus the offset).

The current DateTime/DateTimeOffset

Both `DateTime` and `DateTimeOffset` have a static `Now` property that returns the current date and time:

```
Console.WriteLine (DateTime.Now);         // 11/11/2019 1:23:45 PM
Console.WriteLine (DateTimeOffset.Now);   // 11/11/2019 1:23:45 PM -06:00
```

`DateTime` also provides a `Today` property that returns just the date portion:

```
Console.WriteLine (DateTime.Today);       // 11/11/2019 12:00:00 AM
```

The static `UtcNow` property returns the current date and time in UTC:

```
Console.WriteLine (DateTime.UtcNow);        // 11/11/2019 7:23:45 AM
Console.WriteLine (DateTimeOffset.UtcNow);  // 11/11/2019 7:23:45 AM +00:00
```

The precision of all these methods depends on the operating system and is typically in the 10 to 20 ms region.

Working with dates and times

`DateTime` and `DateTimeOffset` provide a similar set of instance properties that return various date/time elements:

```
DateTime dt = new DateTime (2000, 2, 3,
                            10, 20, 30);

Console.WriteLine (dt.Year);         // 2000
Console.WriteLine (dt.Month);        // 2
Console.WriteLine (dt.Day);          // 3
Console.WriteLine (dt.DayOfWeek);    // Thursday
Console.WriteLine (dt.DayOfYear);    // 34

Console.WriteLine (dt.Hour);         // 10
Console.WriteLine (dt.Minute);       // 20
Console.WriteLine (dt.Second);       // 30
Console.WriteLine (dt.Millisecond);  // 0
Console.WriteLine (dt.Ticks);        // 630851700300000000
Console.WriteLine (dt.TimeOfDay);    // 10:20:30  (returns a TimeSpan)
```

`DateTimeOffset` also has an `Offset` property of type `TimeSpan`.

Both types provide the following instance methods to perform computations (most accept an argument of type `double` or `int`):

```
AddYears  AddMonths   AddDays
AddHours  AddMinutes  AddSeconds  AddMilliseconds  AddTicks
```

These all return a new `DateTime` or `DateTimeOffset`, and they take into account such things as leap years. You can pass in a negative value to subtract.

The `Add` method adds a `TimeSpan` to a `DateTime` or `DateTimeOffset`. The `+` operator is overloaded to do the same job:

```
TimeSpan ts = TimeSpan.FromMinutes (90);
Console.WriteLine (dt.Add (ts));
Console.WriteLine (dt + ts);             // same as above
```

You can also subtract a `TimeSpan` from a `DateTime`/`DateTimeOffset` and subtract one `DateTime`/`DateTimeOffset` from another. The latter gives you a `TimeSpan`:

```
DateTime thisYear = new DateTime (2015, 1, 1);
DateTime nextYear = thisYear.AddYears (1);
```

```
TimeSpan oneYear = nextYear - thisYear;
```

Formatting and parsing datetimes

Calling `ToString` on a `DateTime` formats the result as a *short date* (all numbers) followed by a *long time* (including seconds); for example:

```
11/11/2019 11:50:30 AM
```

The operating system's control panel, by default, determines such things as whether the day, month, or year comes first, the use of leading zeros, and whether 12- or 24-hour time is used.

Calling `ToString` on a `DateTimeOffset` is the same, except that the offset is also returned:

```
11/11/2019 11:50:30 AM -06:00
```

The `ToShortDateString` and `ToLongDateString` methods return just the date portion. The long date format is also determined by the control panel; an example is "Wednesday, 11 November 2015." `ToShortTimeString` and `ToLongTimeString` return just the time portion, such as 17:10:10 (the former excludes seconds).

These four just-described methods are actually shortcuts to four different *format strings*. `ToString` is overloaded to accept a format string and provider, allowing you to specify a wide range of options and control how regional settings are applied. We describe this in "Formatting and Parsing".

WARNING

`DateTime`s and `DateTimeOffset`s can be misparsed if the culture settings differ from those in force when formatting takes place. You can avoid this problem by using `ToString` in conjunction with a format string that ignores culture settings (such as “o”):

```
DateTime dt1 = DateTime.Now;
string cannotBeMisparsed = dt1.ToString ("o");
DateTime dt2 = DateTime.Parse (cannotBeMisparsed);
```

The static `Parse`/`TryParse` and `ParseExact`/`TryParseExact` methods do the reverse of `ToString`, converting a string to a `DateTime` or `DateTimeOffset`. These methods are also overloaded to accept a format provider. The `Try*` methods return `false` instead of throwing a `FormatException`.

Null DateTime and DateTimeOffset values

Because `DateTime` and `DateTimeOffset` are structs, they are not intrinsically nullable. When you need nullability, there are two ways around this:

- Use a `Nullable` type (i.e., `DateTime?` or `DateTimeOffset?`).
- Use the static field `DateTime.MinValue` or `DateTimeOffset.MinValue` (the *default values* for these types).

A nullable type is usually the best approach because the compiler helps to prevent mistakes. `DateTime.MinValue` is useful for backward compatibility with code written prior to C# 2.0 (when nullable value types were introduced).

WARNING

Calling `ToUniversalTime` or `ToLocalTime` on a `DateTime.MinValue` can result in it no longer being `DateTime.MinValue` (depending on which side of GMT you are on). If you're right on GMT (England, outside daylight saving), the problem won't arise at all because local and UTC times are the same. This is your compensation for the English winter!

DateOnly and TimeOnly

The `DateOnly` and `TimeOnly` structs (from .NET 6) exist for when you *only* want to represent a date or time.

`DateOnly` is similar to `DateTime`, but without a time component. `DateOnly` also lacks `DateTimeKind`; in effect, it's always `Unspecified` and has no concept of `Local` or `Utc`. The historical alternative to `DateOnly` was to use `DateTime` with a zero time (midnight). The difficulty with this approach is that equality comparisons fail when a non-zero time find its way into your code.

`TimeOnly` is similar to `DateTime`, but without a date component. `TimeOnly` is intended for capturing the time of day and is suitable for applications such as recording alarm times or opening hours.

Dates and Time Zones

In this section, we examine in more detail how time zones influence `DateTime` and `DateTimeOffset`. We also look at the `TimeZoneInfo` type, which provides information on time zone offsets and Daylight Saving Time.

DateTime and Time Zones

`DateTime` is simplistic in its handling of time zones. Internally, it stores a `DateTime` using two pieces of information:

- A 62-bit number, indicating the number of ticks since 1/1/0001

- A 2-bit enum, indicating the `DateTimeKind` (`Unspecified`, `Local`, or `Utc`)

When you compare two `DateTime` instances, only their *ticks* values are compared; their `DateTimeKinds` are ignored:

```
DateTime dt1 = new DateTime (2000, 1, 1, 10, 20, 30, DateTimeKind.Local);
DateTime dt2 = new DateTime (2000, 1, 1, 10, 20, 30, DateTimeKind.Utc);
Console.WriteLine (dt1 == dt2);          // True
DateTime local = DateTime.Now;
DateTime utc = local.ToUniversalTime();
Console.WriteLine (local == utc);        // False
```

The instance methods `ToUniversalTime`/`ToLocalTime` convert to universal/local time. These apply the computer's current time zone settings and return a new `DateTime` with a `DateTimeKind` of `Utc` or `Local`. No conversion happens if you call `ToUniversalTime` on a `DateTime` that's already `Utc`, or `ToLocalTime` on a `DateTime` that's already `Local`. You will get a conversion, however, if you call `ToUniversalTime` or `ToLocalTime` on a `DateTime` that's `Unspecified`.

You can construct a `DateTime` that differs from another only in `Kind` with the static `DateTime.SpecifyKind` method:

```
DateTime d = new DateTime (2015, 12, 12);  // Unspecified
DateTime utc = DateTime.SpecifyKind (d, DateTimeKind.Utc);
Console.WriteLine (utc);           // 12/12/2015 12:00:00 AM
```

DateTimeOffset and Time Zones

Internally, `DateTimeOffset` comprises a `DateTime` field whose value is always in UTC, and a 16-bit integer field for the UTC offset in minutes. Comparisons look only at the (UTC) `DateTime`; the `Offset` is used primarily for formatting.

The `ToUniversalTime`/`ToLocalTime` methods return a `DateTimeOffset` representing the same point in time but with a UTC or local offset. Unlike with DateTime, these methods don't affect the underlying date/time value, only the offset:

```
DateTimeOffset local = DateTimeOffset.Now;
DateTimeOffset utc   = local.ToUniversalTime();

Console.WriteLine (local.Offset);   // -06:00:00 (in Central America)
Console.WriteLine (utc.Offset);     // 00:00:00

Console.WriteLine (local == utc);                // True
```

To include the Offset in the comparison, you must use the EqualsExact method:

```
Console.WriteLine (local.EqualsExact (utc));      // False
```

TimeZoneInfo

The TimeZoneInfo class provides information on time zone names, UTC offsets, and Daylight Saving Time rules.

TimeZone

The static TimeZone.CurrentTimeZone method returns a TimeZone

```
TimeZone zone = TimeZone.CurrentTimeZone;
Console.WriteLine (zone.StandardName);      // Pacific Standard Time
Console.WriteLine (zone.DaylightName);      // Pacific Daylight Time
```

The GetDaylightChanges method returns specific Daylight Saving Time information for a given year:

```
DaylightTime day = zone.GetDaylightChanges (2019);
Console.WriteLine (day.Start.ToString ("M"));       // 10 March
Console.WriteLine (day.End.ToString ("M"));         // 03 November
Console.WriteLine (day.Delta);                      // 01:00:00
```

TimeZoneInfo

The static `TimeZoneInfo.Local` method returns a `TimeZoneInfo` object based on the current local settings. The following demonstrates the result if run in California:

```
TimeZoneInfo zone = TimeZoneInfo.Local;
Console.WriteLine (zone.StandardName);      // Pacific Standard Time
Console.WriteLine (zone.DaylightName);      // Pacific Daylight Time
```

The `IsDaylightSavingTime` and `GetUtcOffset` methods work as follows:

```
DateTime dt1 = new DateTime (2019, 1, 1);   // DateTimeOffset works, too
DateTime dt2 = new DateTime (2019, 6, 1);
Console.WriteLine (zone.IsDaylightSavingTime (dt1));     // True
Console.WriteLine (zone.IsDaylightSavingTime (dt2));     // False
Console.WriteLine (zone.GetUtcOffset (dt1));             // -08:00:00
Console.WriteLine (zone.GetUtcOffset (dt2));             // -07:00:00
```

You can obtain a `TimeZoneInfo` for any of the world's time zones by calling `FindSystemTimeZoneById` with the zone ID. We'll switch to Western Australia for reasons that will soon become clear:

```
TimeZoneInfo wa = TimeZoneInfo.FindSystemTimeZoneById
                  ("W. Australia Standard Time");

Console.WriteLine (wa.Id);                   // W. Australia Standard Time
Console.WriteLine (wa.DisplayName);          // (GMT+08:00) Perth
Console.WriteLine (wa.BaseUtcOffset);        // 08:00:00
Console.WriteLine (wa.SupportsDaylightSavingTime);     // True
```

The `Id` property corresponds to the value passed to `FindSystemTimeZoneById`. The static `GetSystemTimeZones` method returns all world time zones; hence, you can list all valid zone ID strings as follows:

```
foreach (TimeZoneInfo z in TimeZoneInfo.GetSystemTimeZones())
  Console.WriteLine (z.Id);
```

NOTE

You can also create a custom time zone by calling `TimeZoneInfo.CreateCustomTimeZone`. Because `TimeZoneInfo` is immutable, you must pass in all the relevant data as method arguments.

You can serialize a predefined or custom time zone to a (semi) human-readable string by calling `ToSerializedString`—and deserialize it by calling `TimeZoneInfo.FromSerializedString`.

The static `ConvertTime` method converts a `DateTime` or `DateTimeOffset` from one time zone to another. You can include either just a destination `TimeZoneInfo`, or both source and destination `TimeZoneInfo` objects. You can also convert directly from or to UTC with the methods `ConvertTimeFromUtc` and `ConvertTimeToUtc`.

For working with Daylight Saving Time, `TimeZoneInfo` provides the following additional methods:

- `IsInvalidTime` returns `true` if a `DateTime` is within the hour (or delta) that's skipped when the clocks move forward.
- `IsAmbiguousTime` returns `true` if a `DateTime` or `DateTimeOffset` is within the hour (or delta) that's repeated when the clocks move back.
- `GetAmbiguousTimeOffsets` returns an array of `TimeSpans` representing the valid offset choices for an ambiguous `DateTime` or `DateTimeOffset`.

You can't obtain simple dates from a `TimeZoneInfo` indicating the start and end of Daylight Saving Time. Instead, you must call `GetAdjustmentRules`, which returns a declarative summary of all daylight saving rules that apply to all years. Each rule has a `DateStart` and `DateEnd` indicating the date range within which the rule is valid:

```
foreach (TimeZoneInfo.AdjustmentRule rule in wa.GetAdjustmentRules())
  Console.WriteLine ("Rule: applies from " + rule.DateStart +
                                 " to " + rule.DateEnd);
```

Western Australia first introduced Daylight Saving Time in 2006, *midseason* (and then rescinded it in 2009). This required a special rule for the first year; hence, there are two rules:

```
Rule: applies from 1/01/2006 12:00:00 AM to 31/12/2006 12:00:00 AM
Rule: applies from 1/01/2007 12:00:00 AM to 31/12/2009 12:00:00 AM
```

Each `AdjustmentRule` has a `DaylightDelta` property of type `TimeSpan` (this is one hour in almost every case) and properties called `DaylightTransitionStart` and `DaylightTransitionEnd`. The latter two are of type `TimeZoneInfo.TransitionTime`, which has the following properties:

```
public bool IsFixedDateRule { get; }
public DayOfWeek DayOfWeek { get; }
public int Week { get; }
public int Day { get; }
public int Month { get; }
public DateTime TimeOfDay { get; }
```

A transition time is somewhat complicated in that it needs to represent both fixed and floating dates. An example of a floating date is "the last Sunday in March." Here are the rules for interpreting a transition time:

1. If, for an end transition, `IsFixedDateRule` is `true`, `Day` is `1`, `Month` is `1`, and `TimeOfDay` is `DateTime.MinValue`, there is no end to Daylight Saving Time in that year (this can happen only in the southern hemisphere, upon the initial introduction of daylight saving time to a region).

2. Otherwise, if `IsFixedDateRule` is `true`, the `Month`, `Day`, and `TimeOfDay` properties determine the start or end of the adjustment rule.

3. Otherwise, if `IsFixedDateRule` is `false`, the `Month`, `DayOfWeek`, `Week`, and `TimeOfDay` properties determine the start or end of the adjustment rule.

In the last case, `Week` refers to the week of the month, with “5” meaning the last week. We can demonstrate this by enumerating the adjustment rules for our `wa` time zone:

```
foreach (TimeZoneInfo.AdjustmentRule rule in wa.GetAdjustmentRules())
{
  Console.WriteLine ("Rule: applies from " + rule.DateStart +
                                    " to " + rule.DateEnd);

  Console.WriteLine ("   Delta: " + rule.DaylightDelta);

  Console.WriteLine ("   Start: " + FormatTransitionTime
                                   (rule.DaylightTransitionStart, false));

  Console.WriteLine ("   End:   " + FormatTransitionTime
                                   (rule.DaylightTransitionEnd, true));
  Console.WriteLine();
}
```

In `FormatTransitionTime`, we honor the rules just described:

```
static string FormatTransitionTime (TimeZoneInfo.TransitionTime tt,
                                    bool endTime)
{
  if (endTime && tt.IsFixedDateRule
              && tt.Day == 1 && tt.Month == 1
              && tt.TimeOfDay == DateTime.MinValue)
    return "-";

  string s;
  if (tt.IsFixedDateRule)
    s = tt.Day.ToString();
  else
    s = "The " +
        "first second third fourth last".Split() [tt.Week - 1] +
        " " + tt.DayOfWeek + " in";

  return s + " " + DateTimeFormatInfo.CurrentInfo.MonthNames [tt.Month-1]
           + " at " + tt.TimeOfDay.TimeOfDay;
}
```

Daylight Saving Time and DateTime

If you use a `DateTimeOffset` or a UTC `DateTime`, equality comparisons are unimpeded by the effects of Daylight Saving Time. But with local `DateTime`s, daylight saving can be problematic.

We can summarize the rules as follows:

- Daylight saving affects local time but not UTC time.
- When the clocks turn back, comparisons that rely on time moving forward will break if (and only if) they use local `DateTime`s.
- You can always reliably round-trip between UTC and local times (on the same computer)—even as the clocks turn back.

The `IsDaylightSavingTime` tells you whether a given local `DateTime` is subject to Daylight Saving Time. UTC times always return `false`:

```
Console.Write (DateTime.Now.IsDaylightSavingTime());     // True or False
Console.Write (DateTime.UtcNow.IsDaylightSavingTime());  // Always False
```

Assuming `dto` is a `DateTimeOffset`, the following expression does the same:

```
dto.LocalDateTime.IsDaylightSavingTime
```

The end of Daylight Saving Time presents a particular complication for algorithms that use local time, because when the clocks go back, the same hour (or more precisely, `Delta`) repeats itself.

NOTE

You can reliably compare any two `DateTime`s by first calling `ToUniversalTime` on each. This strategy fails if (and only if) exactly one of them has a `DateTimeKind` of `Unspecified`. This potential for failure is another reason for favoring `DateTimeOffset`.

Formatting and Parsing

Formatting means converting *to* a string; parsing means converting *from* a string. The need to format or parse arises frequently in programming, in a variety of situations. Hence, .NET provides a variety of mechanisms:

`ToString` *and* `Parse`

> These methods provide default functionality for many types.

Format providers

> These manifest as additional `ToString` (and `Parse`) methods that accept a *format string* and/or a *format provider*. Format providers are highly flexible and culture-aware. .NET includes format providers for the numeric types and `DateTime`/`DateTimeOffset`.

`XmlConvert`

> This is a static class with methods that format and parse while honoring XML standards. `XmlConvert` is also useful for general-purpose conversion when you need culture independence or you want to preempt misparsing. `XmlConvert` supports the numeric types, `bool`, `DateTime`, `DateTimeOffset`, `TimeSpan`, and `Guid`.

Type converters

> These target designers and XAML parsers.

In this section, we discuss the first two mechanisms, focusing particularly on format providers. We then describe `XmlConvert`, type converters, and other conversion mechanisms.

ToString and Parse

The simplest formatting mechanism is the `ToString` method. It gives meaningful output on all simple value types (`bool`, `DateTime`, `DateTimeOffset`, `TimeSpan`, `Guid`, and all the numeric types). For the reverse operation, each of these types defines a static `Parse` method:

```
string s = true.ToString();     // s = "True"
bool b = bool.Parse (s);        // b = true
```

If the parsing fails, a `FormatException` is thrown. Many types also define a `TryParse` method, which returns `false` if the conversion fails rather than throwing an exception:

```
bool failure = int.TryParse ("qwerty", out int i1);
bool success = int.TryParse ("123", out int i2);
```

If you don't care about the output and want to test only whether parsing would succeed, you can use a discard:

```
bool success = int.TryParse ("123", out int _);
```

If you anticipate an error, calling `TryParse` is faster and more elegant than calling `Parse` in an exception handling block.

The `Parse` and `TryParse` methods on `DateTime`(`Offset`) and the numeric types respect local culture settings; you can change this by specifying a `CultureInfo` object. Specifying invariant culture is often a good idea. For instance, parsing "1.234" into a `double` gives us 1234 in Germany:

```
Console.WriteLine (double.Parse ("1.234"));   // 1234  (In Germany)
```

This is because in Germany, the period indicates a thousands separator rather than a decimal point. Specifying *invariant culture* fixes this:

```
double x = double.Parse ("1.234", CultureInfo.InvariantCulture);
```

The same applies when calling `ToString()`:

```
string x = 1.234.ToString (CultureInfo.InvariantCulture);
```

NOTE

From .NET 8, the .NET numeric and date/time types (as well as other simple types) allow direct formatting and parsing of UTF-8, via new `TryFormat` and `Parse`/`TryParse` methods that operate on a byte array or `Span<byte>` (see Chapter 23). In high-performance scenarios, this can be more efficient than working with ordinary (UTF-16) strings and performing a separate UTF-8 encoding/decoding.

Format Providers

Sometimes, you need more control over how formatting and parsing take place. There are dozens of ways to format a `DateTime(Offset)`, for instance. Format providers allow extensive control over formatting and parsing, and are supported for numeric types and date/times. Format providers are also used by user interface controls for formatting and parsing.

The gateway to using a format provider is `IFormattable`. All numeric types—and `DateTime(Offset)`—implement this interface:

```
public interface IFormattable
{
  string ToString (string format, IFormatProvider formatProvider);
}
```

The first argument is the *format string*; the second is the *format provider*. The format string provides instructions; the format provider determines how the instructions are translated. For example:

```
NumberFormatInfo f = new NumberFormatInfo();
f.CurrencySymbol = "$$";
Console.WriteLine (3.ToString ("C", f));          // $$ 3.00
```

Here, `"C"` is a format string that indicates *currency*, and the `NumberFormatInfo` object is a format provider that determines how currency—and other numeric representations—are rendered. This mechanism allows for globalization.

NOTE

All format strings for numbers and dates are listed in "Standard Format Strings and Parsing Flags".

If you specify a `null` format string or provider, a default is applied. The default format provider is `CultureInfo.CurrentCulture`, which, unless reassigned, reflects the computer's runtime control panel settings. For example, on this computer:

```
Console.WriteLine (10.3.ToString ("C", null));  // $10.30
```

For convenience, most types overload `ToString` such that you can omit a `null` provider:

```
Console.WriteLine (10.3.ToString ("C"));     // $10.30
Console.WriteLine (10.3.ToString ("F4"));    // 10.3000 (Fix to 4 D.P.)
```

Calling `ToString` on a `DateTime`(`Offset`) or a numeric type with no arguments is equivalent to using a default format provider, with an empty format string.

.NET defines three format providers (all of which implement `IFormatProvider`):

```
NumberFormatInfo
DateTimeFormatInfo
CultureInfo
```

NOTE

All `enum` types are also formattable, though there's no special `IFormatProvider` class.

Format providers and CultureInfo

Within the context of format providers, `CultureInfo` acts as an indirection mechanism for the other two format providers, returning a `NumberFormatInfo` or `DateTimeFormatInfo` object applicable to the culture's regional settings.

In the following example, we request a specific culture (*en*glish language in *G*reat *B*ritain):

```
CultureInfo uk = CultureInfo.GetCultureInfo ("en-GB");
Console.WriteLine (3.ToString ("C", uk));      // £3.00
```

This executes using the default `NumberFormatInfo` object applicable to the en-GB culture.

The next example formats a `DateTime` with invariant culture. Invariant culture is always the same, regardless of the computer's settings:

```
DateTime dt = new DateTime (2000, 1, 2);
CultureInfo iv = CultureInfo.InvariantCulture;
Console.WriteLine (dt.ToString (iv));           // 01/02/2000 00:00:00
Console.WriteLine (dt.ToString ("d", iv));      // 01/02/2000
```

NOTE

Invariant culture is based on American culture, with the following differences:

- The currency symbol is ☼ instead of $.
- Dates and times are formatted with leading zeros (though still with the month first).
- Time uses the 24-hour format rather than an AM/PM designator.

Using NumberFormatInfo or DateTimeFormatInfo

In the next example, we instantiate a `NumberFormatInfo` and change the group separator from a comma to a space. We then use it to format a number to three decimal places:

```
NumberFormatInfo f = new NumberFormatInfo ();
f.NumberGroupSeparator = " ";
Console.WriteLine (12345.6789.ToString ("N3", f));   // 12 345.679
```

The initial settings for a `NumberFormatInfo` or `DateTimeFormatInfo` are based on the invariant culture. Sometimes, however, it's more useful to choose a different starting point. To do this, you can `Clone` an existing format provider:

```
NumberFormatInfo f = (NumberFormatInfo)
                      CultureInfo.CurrentCulture.NumberFormat.Clone();
```

A cloned format provider is always writable—even if the original was read-only.

Composite formatting

Composite format strings allow you to combine variable substitution with format strings. The static `string.Format` method accepts a composite format string (we illustrated this in "String.Format and composite format strings"):

```
string composite = "Credit={0:C}";
Console.WriteLine (string.Format (composite, 500));   // Credit=$500.00
```

The `Console` class itself overloads its `Write` and `WriteLine` methods to accept composite format strings, allowing us to shorten this example slightly:

```
Console.WriteLine ("Credit={0:C}", 500);   // Credit=$500.00
```

You can also append a composite format string to a `StringBuilder` (via `AppendFormat`), and to a `TextWriter` for I/O (see Chapter 15).

`string.Format` accepts an optional format provider. A simple application for this is to call `ToString` on an arbitrary object while passing in a format provider:

```
string s = string.Format (CultureInfo.InvariantCulture, "{0}", someObject);
```

This is equivalent to the following:

```
string s;
if (someObject is IFormattable)
  s = ((IFormattable)someObject).ToString (null,
                                          CultureInfo.InvariantCulture);
else if (someObject == null)
  s = "";
else
  s = someObject.ToString();
```

Parsing with format providers

There's no standard interface for parsing through a format provider. Instead, each participating type overloads its static `Parse` (and `TryParse`) method to accept a format provider, and optionally, a `NumberStyles` or `DateTimeStyles` enum.

`NumberStyles` and `DateTimeStyles` control how parsing works: they let you specify such things as whether parentheses or a currency symbol can appear in the input string. (By default, the answer to both questions is *no*.) For example:

```
int error = int.Parse ("(2)");   // Exception thrown

int minusTwo = int.Parse ("(2)", NumberStyles.Integer |
                                 NumberStyles.AllowParentheses);   // OK

decimal fivePointTwo = decimal.Parse ("£5.20", NumberStyles.Currency,
                       CultureInfo.GetCultureInfo ("en-GB"));
```

The next section lists all NumberStyles and DateTimeStyles members as well as the default parsing rules for each type.

IFormatProvider and ICustomFormatter

All format providers implement IFormatProvider:

```
public interface IFormatProvider { object GetFormat (Type formatType); }
```

The purpose of this method is to provide indirection—this is what allows CultureInfo to defer to an appropriate NumberFormatInfo or DateTimeFormatInfo object to do the work.

By implementing IFormatProvider—along with ICustomFormatter—you can also write your own format provider that works in conjunction with existing types. ICustomFormatter defines a single method, as follows:

```
string Format (string format, object arg, IFormatProvider formatProvider);
```

The following custom format provider writes numbers as words:

```
public class WordyFormatProvider : IFormatProvider, ICustomFormatter
{
  static readonly string[] _numberWords =
   "zero one two three four five six seven eight nine minus point".Split();

  IFormatProvider _parent;   // Allows consumers to chain format providers

  public WordyFormatProvider () : this (CultureInfo.CurrentCulture) { }
  public WordyFormatProvider (IFormatProvider parent) => _parent = parent;

  public object GetFormat (Type formatType)
  {
    if (formatType == typeof (ICustomFormatter)) return this;
    return null;
  }

  public string Format (string format, object arg, IFormatProvider prov)
  {
    // If it's not our format string, defer to the parent provider:
    if (arg == null || format != "W")
```

```
      return string.Format (_parent, "{0:" + format + "}", arg);

    StringBuilder result = new StringBuilder();
    string digitList = string.Format (CultureInfo.InvariantCulture,
                                      "{0}", arg);
    foreach (char digit in digitList)
    {
      int i = "0123456789-.".IndexOf (digit,
                                      StringComparison.InvariantCulture);
      if (i == -1) continue;
      if (result.Length > 0) result.Append (' ');
      result.Append (_numberWords[i]);
    }
    return result.ToString();
  }
}
```

Notice that in the `Format` method, we used `string.Format`—with `InvariantCulture`—to convert the input number to a string. It would have been simpler just to call `ToString()` on `arg`, but then `CurrentCulture` would have been used, instead. The reason for needing the invariant culture is evident a few lines later:

```
int i = "0123456789-.".IndexOf (digit, StringComparison.InvariantCulture);
```

It's critical here that the number string comprises only the characters `0123456789-.` and not any internationalized versions of these.

Here's an example of using `WordyFormatProvider`:

```
double n = -123.45;
IFormatProvider fp = new WordyFormatProvider();
Console.WriteLine (string.Format (fp, "{0:C} in words is {0:W}", n));

// -$123.45 in words is minus one two three point four five
```

You can use custom format providers only in composite format strings.

Standard Format Strings and Parsing Flags

The standard format strings control how a numeric type or `DateTime`/`DateTimeOffset` is converted to a string. There are two kinds of format strings:

Standard format strings

With these, you provide general guidance. A standard format string consists of a single letter, followed, optionally, by a digit (whose meaning depends on the letter). An example is `"C"` or `"F2"`.

Custom format strings

With these, you micromanage every character with a template. An example is `"0:#.000E+00"`.

Custom format strings are unrelated to custom format providers.

Numeric Format Strings

Table 6-2 lists all standard numeric format strings.

Table 6-2. Standard numeric format strings

Letter	Meaning	Sample input	Result	Note

`G` or `g`	"General"	`1.2345, "G"`
		`0.00001, "G"`
		`0.00001, "g"`
		`1.2345, "G3"`
		`12345, "G3"`
			`1.2345`
			`1E-05`
			`1e-05`
			`1.23`
			`1.23E04`
				Switc expo notat smal numb `G3` lin preci three *total* after

`F`	Fixed point	`2345.678, "F2"`
		`2345.6, "F2"`
			`2345.68`
			`2345.60`
				`F2` rou two c place

`N`	Fixed point with *group separator* ("Numeric")	`2345.678, "N2"`
		`2345.6, "N2"`
			`2,345.68`
			`2,345.60`
				As ab with (1,00 separ (deta form provi

`D`	Pad with leading zeros	`123, "D5"`
		`123, "D1"`
			`00123`
			`123`
				For i types `D5` pa five does trunc

Letter	Meaning	Sample input	Result	Note

`E` or `e`	Force exponential notation	`56789, "E"`
		`56789, "e"`
		`56789, "E2"`
			`5.678900E+004`
			`5.678900e+004`
			`5.68E+004`
				Six-d defau preci

`C`	Currency	`1.2, "C"`
		`1.2, "C4"`
			`$1.20`
			`$1.2000`
				`C` witl uses numb D.P. forma provi

`P`	Percent	`.503, "P"`
		`.503, "P0"`
			`50.30%`
			`50%`
				Uses and l from provi Deci place optio overr

`X` or `x`	Hexadecimal	`47, "X"`
		`47, "x"`
		`47, "X4"`
			`2F`
			`2f`
			`002F`
				`X` for uppe digits lowe digits Integ

`R` or `G9`/`G17`	Round-trip	`1f / 3f, "R"`	`0.333333`**`43`**	Use `R` `teger,` `uble,` `loat.`

Supplying no numeric format string (or a null or blank string) is equivalent to using the "G" standard format string followed by no digit. This exhibits the following behavior:

- Numbers smaller than 10^{-4} or larger than the type's precision are expressed in exponential (scientific) notation.
- The two decimal places at the limit of `float` or `double`'s precision are rounded away to mask the inaccuracies inherent in conversion to decimal from their underlying binary form.

NOTE

The automatic rounding just described is usually beneficial and goes unnoticed. However, it can cause trouble if you need to round-trip a number—in other words, convert it to a string and back again (maybe repeatedly) while preserving value equality. For this reason, the `R`, `G17`, and `G9` format strings exist to circumvent this implicit rounding.

Table 6-3 lists custom numeric format strings.

Table 6-3. Custom numeric format strings

Specifier	Meaning	Sample input	Result	Note

`#`	Digit placeholder	`12.345, ".##"`
		`12.345, ".####"`
			`12.35`
			`12.345`
				Limi
				after

`0`	Zero placeholder	`12.345, ".00"`
		`12.345, ".0000"`
		`99, "000.00"`
			`12.35`
			`12.3450`
			`099.00`
				As ab
				also p
				zeros
				and a

`.`	Decimal point			Indic Actu come `mberFo`

`,`	Group separator	`1234, "#,###,###"`
		`1234, "0,000,000"`
			`1,234`
			`0,001,234`
				Symb
				from
				`atInfo`

`,`
(as above)
	Multiplier	`1000000, "#,"`
		`1000000, "#,,`
			`1000`
			`1`
				If co
				end c
				D.P.,
				a mu
				divid
				by 1,
				1,000

Specifier	Meaning	Sample input	Result	Note
`%`	Percent notation	`0.6, "00%"`	`60%`	First multi 100 a subst perce symb obtai `Numbe` .
`E0, e0, E+0, e+0` `E-0, e-0`	Exponent notation	`1234, "0E0"` `1234, "0E+0"` `1234, "0.00E00"` `1234, "0.00e00"`	`1E3` `1E+3` `1.23E03` `1.23e03`	
`\`	Literal character quote	`50, @"\#0"`	`#50`	Use i conju with prefi string \\
`'xx''xx'`	Literal string quote	`50, "0 '...'"`	`50 ...`	
`;`	Section separator	`15, "#;(#);zero"`	`15`	(If po
		`-5, "#;(#);zero"`	`(5)`	(If ne
		`0, "#;(#);zero"`	`zero`	(If ze
Any other char	Literal	`35.2, "$0 . 00c"`	`$35 . 20c`	

NumberStyles

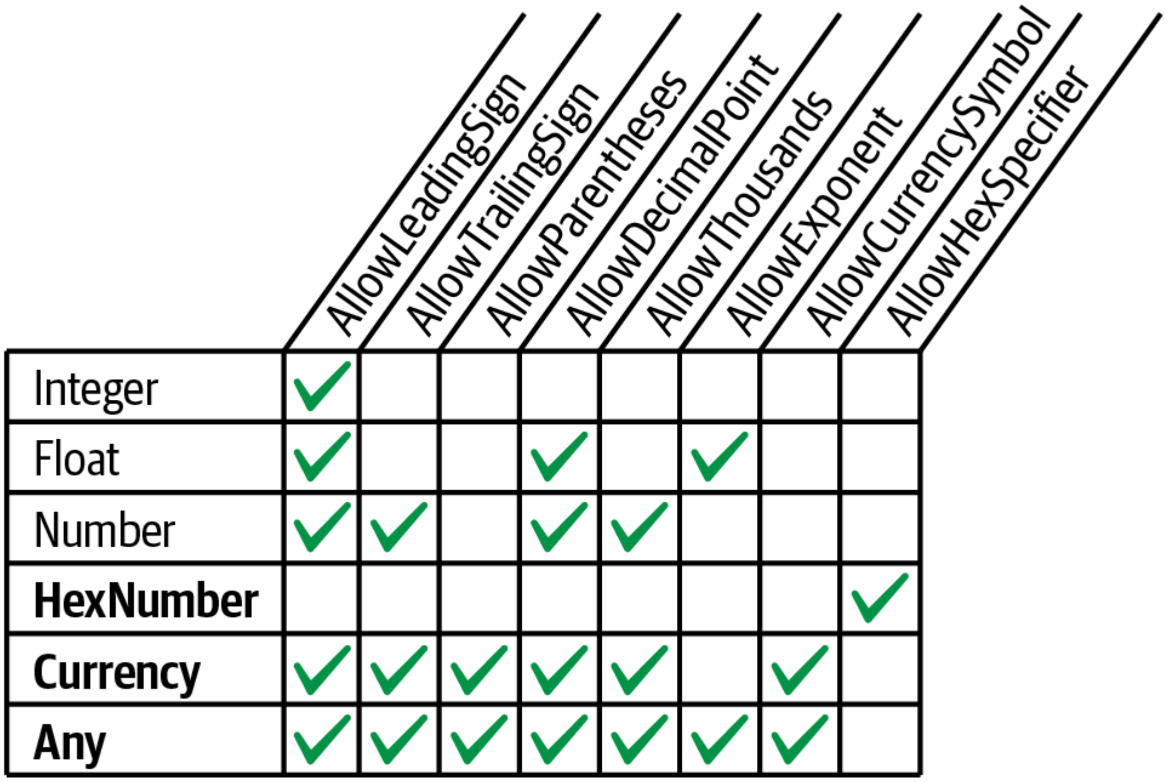
Each numeric type defines a static `Parse` method that accepts a `NumberStyles` argument. `NumberStyles` is a flags enum that lets you determine how the string is read as it’s converted to a numeric type. It has the following combinable members:

```
AllowLeadingWhite    AllowTrailingWhite
AllowLeadingSign     AllowTrailingSign
AllowParentheses     AllowDecimalPoint
AllowThousands       AllowExponent
AllowCurrencySymbol  AllowHexSpecifier
```

`NumberStyles` also defines these composite members:

```
None  Integer  Float  Number  HexNumber  Currency  Any
```

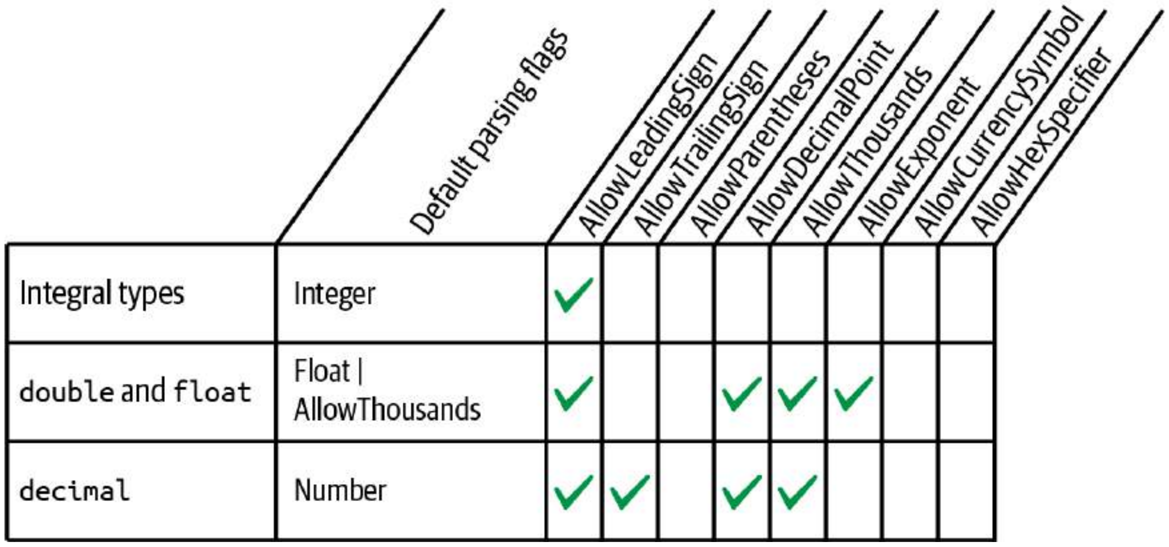
Except for `None`, all composite values include `AllowLeadingWhite` and `AllowTrailingWhite`. Figure 6-1 shows their remaining makeup, with the most useful three emphasized.



	AllowLeadingSign	AllowTrailingSign	AllowParentheses	AllowDecimalPoint	AllowThousands	AllowExponent	AllowCurrencySymbol	AllowHexSpecifier
Integer	✓							
Float	✓			✓		✓		
Number	✓	✓		✓	✓			
HexNumber								✓
Currency	✓	✓	✓	✓	✓		✓	
Any	✓	✓	✓	✓	✓	✓	✓	

Figure 6-1. Composite NumberStyles

When you call `Parse` without specifying any flags, the defaults illustrated in Figure 6-2 are applied.



	Default parsing flags	AllowLeadingSign	AllowTrailingSign	AllowParentheses	AllowDecimalPoint	AllowThousands	AllowExponent	AllowCurrencySymbol	AllowHexSpecifier
Integral types	Integer	✓							
`double` and `float`	Float | AllowThousands	✓			✓	✓	✓		
`decimal`	Number	✓	✓		✓	✓			

Figure 6-2. Default parsing flags for numeric types

If you don't want the defaults shown in Figure 6-2, you must explicitly specify `NumberStyles`:

```
int thousand = int.Parse ("3E8", NumberStyles.HexNumber);
int minusTwo = int.Parse ("(2)", NumberStyles.Integer |
                                 NumberStyles.AllowParentheses);
double aMillion = double.Parse ("1,000,000", NumberStyles.Any);
decimal threeMillion = decimal.Parse ("3e6", NumberStyles.Any);
decimal fivePointTwo = decimal.Parse ("$5.20", NumberStyles.Currency);
```

Because we didn't specify a format provider, this example works with your local currency symbol, group separator, decimal point, and so on. The next example is hardcoded to work with the euro sign and a blank group separator for currencies:

```
NumberFormatInfo ni = new NumberFormatInfo();
ni.CurrencySymbol = "€";
ni.CurrencyGroupSeparator = " ";
double million = double.Parse ("€1 000 000", NumberStyles.Currency, ni);
```

Date/Time Format Strings

Format strings for `DateTime`/`DateTimeOffset` can be divided into two groups based on whether they honor culture and format provider settings. Table 6-4 lists those that do; Table 6-5 lists those that don't. The sample output comes from formatting the following `DateTime` (with *invariant culture*, in the case of Table 6-4):

```
new DateTime (2000, 1, 2,  17, 18, 19);
```

Table 6-4. Culture-sensitive date/time format strings

Format string	Meaning	Sample output
`d`	Short date	`01/02/2000`
`D`	Long date	`Sunday, 02 January 2000`
`t`	Short time	`17:18`
`T`	Long time	`17:18:19`
`f`	Long date + short time	`Sunday, 02 January 2000 17:18`
`F`	Long date + long time	`Sunday, 02 January 2000 17:18:19`
`g`	Short date + short time	`01/02/2000 17:18`
`G` (default)	Short date + long time	`01/02/2000 17:18:19`
`m, M`	Month and day	`02 January`
`y, Y`	Year and month	`January 2000`

Table 6-5. Culture-insensitive date/time format strings

Format string	Meaning	Sample output	Notes
`o`	Round-trippable	`2000-01-02T17:18:19.0000000`	Will append time zone information unless `DateTimeKind` is `Unspecified`
`r`, `R`	RFC 1123 standard	`Sun, 02 Jan 2000 17:18:19 GMT`	You must explicitly convert to UTC with `DateTime.ToUniversalTime`
`s`	Sortable; ISO 8601	`2000-01-02T17:18:19`	Compatible with text-based sorting
`u`	"Universal" sortable	`2000-01-02 17:18:19Z`	Similar to above; must explicitly convert to UTC
`U`	UTC	`Sunday, 02 January 2000 17:18:19`	Long date + short time, converted to UTC

The format strings `"r"`, `"R"`, and `"u"` emit a suffix that implies UTC; yet they don't automatically convert a local to a UTC `DateTime` (so you must do the conversion yourself). Ironically, `"U"` automatically converts to UTC, but doesn't write a time zone suffix! In fact, `"o"` is the only format specifier in the group that can write an unambiguous `DateTime` without intervention.

`DateTimeFormatInfo` also supports custom format strings: these are analogous to numeric custom format strings. The list is extensive and is available online in Microsoft’s documentation. Here’s an example of a custom format string:

```
yyyy-MM-dd HH:mm:ss
```

Parsing and misparsing DateTimes

Strings that put the month or day first are ambiguous and can easily be misparsed—particularly if you have global customers. This is not a problem in user interface controls, because the same settings are in force when parsing as when formatting. But when writing to a file, for instance, day/month misparsing can be a real problem. There are two solutions:

- Always state the same explicit culture when formatting and parsing (e.g., invariant culture).
- Format `DateTime` and `DateTimeOffset`s in a manner *independent* of culture.

The second approach is more robust—particularly if you choose a format that puts the four-digit year first: such strings are much more difficult to misparse by another party. Further, strings formatted with a *standards-compliant* year-first format (such as `"o"`) can parse correctly alongside locally formatted strings—rather like a “universal donor.” (Dates formatted with `"s"` or `"u"` have the further benefit of being sortable.)

To illustrate, suppose that we generate a culture-insensitive `DateTime` string `s` as follows:

```
string s = DateTime.Now.ToString ("o");
```

NOTE

The "o" format string includes milliseconds in the output. The following custom format string gives the same result as "o", but without milliseconds:

```
yyyy-MM-ddTHH:mm:ss K
```

We can reparse this in two ways. ParseExact demands strict compliance with the specified format string:

```
DateTime dt1 = DateTime.ParseExact (s, "o", null);
```

(You can achieve a similar result with XmlConvert's ToString and ToDateTime methods.)

Parse, however, implicitly accepts both the "o" format and the CurrentCulture format:

```
DateTime dt2 = DateTime.Parse (s);
```

This works with both DateTime and DateTimeOffset.

NOTE

ParseExact is usually preferable if you know the format of the string that you're parsing. It means that if the string is incorrectly formatted, an exception will be thrown—which is usually better than risking a misparsed date.

DateTimeStyles

DateTimeStyles is a flags enum that provides additional instructions when calling Parse on a DateTime(Offset). Here are its members:

```
None,
AllowLeadingWhite, AllowTrailingWhite, AllowInnerWhite,
AssumeLocal, AssumeUniversal, AdjustToUniversal,
NoCurrentDateDefault, RoundTripKind
```

There is also a composite member, `AllowWhiteSpaces`:

```
AllowWhiteSpaces = AllowLeadingWhite | AllowTrailingWhite | AllowInnerWhite
```

The default is `None`. This means that extra whitespace is normally prohibited (whitespace that's part of a standard `DateTime` pattern is exempt).

`AssumeLocal` and `AssumeUniversal` apply if the string doesn't have a time zone suffix (such as `Z` or `+9:00`). `AdjustToUniversal` still honors time zone suffixes, but then converts to UTC using the current regional settings.

If you parse a string comprising a time but no date, today's date is applied by default. If you apply the `NoCurrentDateDefault` flag, however, it instead uses 1st January 0001.

Enum Format Strings

In "Enums", we described formatting and parsing enum values. Table 6-6 lists each format string and the result of applying it to the following expression:

```
Console.WriteLine (System.ConsoleColor.Red.ToString (formatString));
```

Table 6-6. Enum format strings

Format string	Meaning	Sample output	Notes
`G` or `g`	"General"	`Red`	Default
`F` or `f`	Treat as though `Flags` attribute were present	`Red`	Works on combined members even if `enum` has no `Flags` attribute
`D` or `d`	Decimal value	`12`	Retrieves underlying integral value
`X` or `x`	Hexadecimal value	`0000000C`	Retrieves underlying integral value

Other Conversion Mechanisms

In the previous two sections, we covered format providers—.NET's primary mechanism for formatting and parsing. Other important conversion mechanisms are scattered through various types and namespaces. Some convert to and from `string`, and some do other kinds of conversions. In this section, we discuss the following topics:

- The `Convert` class and its functions:
    - Real to integral conversions that round rather than truncate
    - Parsing numbers in base 2, 8, and 16
    - Dynamic conversions

    - Base-64 translations

- `XmlConvert` and its role in formatting and parsing for XML
- Type converters and their role in formatting and parsing for designers and XAML
- `BitConverter`, for binary conversions

Convert

.NET calls the following types *base types*:

- `bool`, `char`, `string`, `System.DateTime`, and `System.DateTimeOffset`
- All the C# numeric types

The static `Convert` class defines methods for converting every base type to every other base type. Unfortunately, most of these methods are useless: either they throw exceptions or they are redundant alongside implicit casts. Among the clutter, however, are some useful methods, listed in the following sections.

> **NOTE**
>
> All base types (explicitly) implement `IConvertible`, which defines methods for converting to every other base type. In most cases, the implementation of each of these methods simply calls a method in `Convert`. On rare occasions, it can be useful to write a method that accepts an argument of type `IConvertible`.

Rounding real to integral conversions

In Chapter 2, we saw how implicit and explicit casts allow you to convert between numeric types. In summary:

- Implicit casts work for nonlossy conversions (e.g., `int` to `double`).

- Explicit casts are required for lossy conversions (e.g., `double` to `int`).

Casts are optimized for efficiency; hence, they *truncate* data that won't fit. This can be a problem when converting from a real number to an integer, because often you want to *round* rather than truncate. `Convert`'s numerical conversion methods address just this issue—they always *round*:

```
double d = 3.9;
int i = Convert.ToInt32 (d);    // i == 4
```

`Convert` uses *banker's rounding*, which snaps midpoint values to even integers (this avoids positive or negative bias). If banker's rounding is a problem, first call `Math.Round` on the real number: this accepts an additional argument that allows you to control midpoint rounding.

Parsing numbers in base 2, 8, and 16

Hidden among the `To(`*`integral-type`*`)` methods are overloads that parse numbers in another base:

```
int thirty = Convert.ToInt32  ("1E", 16);    // Parse in hexadecimal
uint five  = Convert.ToUInt32 ("101", 2);    // Parse in binary
```

The second argument specifies the base. It can be any base you like—as long as it's 2, 8, 10, or 16!

Dynamic conversions

Occasionally, you need to convert from one type to another, but you don't know what the types are until runtime. For this, the `Convert` class provides a `ChangeType` method:

```
public static object ChangeType (object value, Type conversionType);
```

The source and target types must be one of the "base" types. `ChangeType` also accepts an optional `IFormatProvider` argument. Here's an example:

```
Type targetType = typeof (int);
object source = "42";

object result = Convert.ChangeType (source, targetType);

Console.WriteLine (result);             // 42
Console.WriteLine (result.GetType());   // System.Int32
```

An example of when this might be useful is in writing a deserializer that can work with multiple types. It can also convert any enum to its integral type (see “Enums”).

A limitation of `ChangeType` is that you cannot specify a format string or parsing flag.

Base-64 conversions

Sometimes, you need to include binary data such as a bitmap within a text document such as an XML file or email message. Base 64 is a ubiquitous means of encoding binary data as readable characters, using 64 characters from the ASCII set.

`Convert`’s `ToBase64String` method converts from a byte array to base 64; `FromBase64String` does the reverse.

XmlConvert

If you’re dealing with data that’s originated from or destined for an XML file, `XmlConvert` (in the `System.Xml` namespace) provides the most suitable methods for formatting and parsing. The methods in `XmlConvert` handle the nuances of XML formatting without needing special format strings. For instance, `true` in XML is “true” and not “True.” The .NET BCL internally uses `XmlConvert` extensively. `XmlConvert` is also good for general-purpose, culture-independent serialization.

The formatting methods in `XmlConvert` are all provided as overloaded `ToString` methods; the parsing methods are called `ToBoolean`, `ToDateTime`, and so on:

```
string s = XmlConvert.ToString (true);         // s = "true"
bool isTrue = XmlConvert.ToBoolean (s);
```

The methods that convert to and from `DateTime` accept an `XmlDateTimeSerializationMode` argument. This is an `enum` with the following values:

```
Unspecified, Local, Utc, RoundtripKind
```

`Local` and `Utc` cause a conversion to take place when formatting (if the `DateTime` is not already in that time zone). The time zone is then appended to the string:

```
2010-02-22T14:08:30.9375          // Unspecified
2010-02-22T14:07:30.9375+09:00    // Local
2010-02-22T05:08:30.9375Z         // Utc
```

`Unspecified` strips away any time-zone information embedded in the `DateTime` (i.e., `DateTimeKind`) before formatting. `RoundtripKind` honors the `DateTime`'s `DateTimeKind`—so when it's reparsed, the resultant `DateTime` struct will be exactly as it was originally.

Type Converters

Type converters are designed to format and parse in design-time environments. They also parse values in Extensible Application Markup Language (XAML) documents—as used in Windows Presentation Foundation (WPF).

In .NET, there are more than 100 type converters—covering such things as colors, images, and URIs. In contrast, format providers are implemented for only a handful of simple value types.

Type converters typically parse strings in a variety of ways—without needing hints. For instance, in a WPF application in Visual Studio, if you assign a control a background color by typing `"Beige"` into the appropriate property window, `Color`’s type converter figures out that you’re referring to a color name and not an RGB string or system color. This flexibility can sometimes make type converters useful in contexts outside of designers and XAML documents.

All type converters subclass `TypeConverter` in `System.ComponentModel`. To obtain a `TypeConverter`, call `TypeDescriptor.GetConverter`. The following obtains a `TypeConverter` for the `Color` type (in the `System.Drawing` namespace):

```
TypeConverter cc = TypeDescriptor.GetConverter (typeof (Color));
```

Among many other methods, `TypeConverter` defines methods to `ConvertToString` and `ConvertFromString`. We can call these as follows:

```
Color beige  = (Color) cc.ConvertFromString ("Beige");
Color purple = (Color) cc.ConvertFromString ("#800080");
Color window = (Color) cc.ConvertFromString ("Window");
```

By convention, type converters have names ending in *Converter* and are usually in the same namespace as the type they’re converting. A type links to its converter via a `TypeConverterAttribute`, allowing designers to pick up converters automatically.

Type converters can also provide design-time services such as generating standard value lists for populating a drop-down list in a designer or assisting with code serialization.

BitConverter

Most base types can be converted to a byte array, by calling `BitConverter.GetBytes`:

```
foreach (byte b in BitConverter.GetBytes (3.5))
  Console.Write (b + " ");                         // 0 0 0 0 0 0 12 64
```

`BitConverter` also provides methods, such as `ToDouble`, for converting in the other direction.

The `decimal` and `DateTime(Offset)` types are not supported by `BitConverter`. You can, however, convert a `decimal` to an `int` array by calling `decimal.GetBits`. To go the other way around, `decimal` provides a constructor that accepts an `int` array.

In the case of `DateTime`, you can call `ToBinary` on an instance—this returns a `long` (upon which you can then use `BitConverter`). The static `DateTime.FromBinary` method does the reverse.

Globalization

There are two aspects to *internationalizing* an application: *globalization* and *localization*.

Globalization is concerned with three tasks (in decreasing order of importance):

1. Making sure that your program doesn't *break* when run in another culture
2. Respecting a local culture's formatting rules; for instance, when displaying dates
3. Designing your program so that it picks up culture-specific data and strings from satellite assemblies that you can later write and deploy

Localization means concluding that last task by writing satellite assemblies for specific cultures. You can do this *after* writing your program (we cover the details in "Resources and Satellite Assemblies").

.NET helps you with the second task by applying culture-specific rules by default. We've already seen how calling `ToString` on a `DateTime` or number respects local formatting rules. Unfortunately, this makes it easy to fail the first task and have your program break because you're expecting dates or numbers to be formatted according to an assumed culture. The solution, as we’ve seen, is either to specify a culture (such as the invariant culture) when formatting and parsing or to use culture-independent methods such as those in `XmlConvert`.

Globalization Checklist

We’ve already covered the important points in this chapter. Here’s a summary of the essential work required:

- Understand Unicode and text encodings (see “Text Encodings and Unicode”).
- Be mindful that methods such as `ToUpper` and `ToLower` on `char` and `string` are culture sensitive: use `ToUpperInvariant`/`ToLowerInvariant` unless you want culture sensitivity.
- Favor culture-independent formatting and parsing mechanisms for `DateTime` and `DateTimeOffsets` such as `ToString("o")` and `XmlConvert`.
- Otherwise, specify a culture when formatting/parsing numbers or date/times (unless you *want* local-culture behavior).

Testing

You can test against different cultures by reassigning `Thread`’s `CurrentCulture` property (in `System.Threading`). The following changes the current culture to Turkey:

```
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo ("tr-TR");
```

Turkey is a particularly good test case because:

- `"i".ToUpper() != "I"` and `"I".ToLower() != "i"`.

- Dates are formatted as day.month.year (note the period separator).
- The decimal point indicator is a comma instead of a period.

You can also experiment by changing the number and date formatting settings in the Windows Control Panel: these are reflected in the default culture (`CultureInfo.CurrentCulture`).

`CultureInfo.GetCultures()` returns an array of all available cultures.

NOTE

`Thread` and `CultureInfo` also support a `CurrentUICulture` property. This is concerned more with localization, which we cover in Chapter 17.

Working with Numbers

Conversions

We covered numeric conversions in previous chapters and sections; Table 6-7 summarizes all of the options.

Table 6-7. Summary of numeric conversions

Task	Functions	Examples
Parsing base 10 numbers	`Parse` `TryParse`	`double d = double.Parse ("3.5");` `int i;` `bool ok = int.TryParse ("3", out i);`
Parsing from base 2, 8, or 16	`Convert.ToIntegral`	`int i = Convert.ToInt32 ("1E", 16);`
Formatting to hexadecimal	`ToString ("X")`	`string hex = 45.ToString ("X");`
Lossless numeric conversion	Implicit cast	`int i = 23;` `double d = i;`
Truncating numeric conversion	Explicit cast	`double d = 23.5;` `int i = (int) d;`
Rounding numeric conversion (real to integral)	`Convert.ToIntegral`	`double d = 23.5;` `int i = Convert.ToInt32 (d);`

Math

Table 6-8 lists the key members of the static `Math` class. The trigonometric functions accept arguments of type `double`; other methods such as `Max` are overloaded to operate on all numeric types. The `Math` class also defines the mathematical constants `E` (*e*) and `PI`.

Table 6-8. Methods in the static Math class

Category	Methods
Rounding	`Round`, `Truncate`, `Floor`, `Ceiling`
Maximum/minimum	`Max`, `Min`
Absolute value and sign	`Abs`, `Sign`
Square root	`Sqrt`
Raising to a power	`Pow`, `Exp`
Logarithm	`Log`, `Log10`
Trigonometric	`Sin`, `Cos`, `Tan`, `Sinh`, `Cosh`, `Tanh`, `Asin`, `Acos`, `Atan`

The `Round` method lets you specify the number of decimal places with which to round as well as how to handle midpoints (away from zero, or with banker's rounding). `Floor` and `Ceiling` round to the nearest integer: `Floor` always rounds down, and `Ceiling` always rounds up—even with negative numbers.

`Max` and `Min` accept only two arguments. If you have an array or sequence of numbers, use the `Max` and `Min` extension methods in `System.Linq.Enumerable`.

BigInteger

The `BigInteger` struct is a specialized numeric type. It resides in the `System.Numerics` namespace and allows you to represent an arbitrarily large integer without any loss of precision.

C# doesn't provide native support for `BigInteger`, so there's no way to represent `BigInteger` literals. You can, however, implicitly convert from any other integral type to a `BigInteger`:

```
BigInteger twentyFive = 25;      // implicit conversion from integer
```

To represent a bigger number, such as one googol (10^{100}), you can use one of `BigInteger`'s static methods, such as `Pow` (raise to the power):

```
BigInteger googol = BigInteger.Pow (10, 100);
```

Alternatively, you can `Parse` a string:

```
BigInteger googol = BigInteger.Parse ("1".PadRight (101, '0'));
```

Calling `ToString()` on this prints every digit:

```
Console.WriteLine (googol.ToString()); // 10000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
```

You can perform potentially lossy conversions between `BigInteger` and the standard numeric types by using the explicit cast operator:

```
double g2 = (double) googol;        // Explicit cast
BigInteger g3 = (BigInteger) g2;    // Explicit cast
Console.WriteLine (g3);
```

The output from this demonstrates the loss of precision:

```
9999999999999999673361688041166912...
```

`BigInteger` overloads all the arithmetic operators including remainder (`%`) as well as the comparison and equality operators.

You can also construct a `BigInteger` from a byte array. The following code generates a 32-byte random number suitable for cryptography and then assigns it to a `BigInteger`:

```
// This uses the System.Security.Cryptography namespace:
RandomNumberGenerator rand = RandomNumberGenerator.Create();
byte[] bytes = new byte [32];
rand.GetBytes (bytes);
var bigRandomNumber = new BigInteger (bytes);   // Convert to BigInteger
```

The advantage of storing such a number in a `BigInteger` over a byte array is that you get value-type semantics. Calling `ToByteArray` converts a `BigInteger` back to a byte array.

Half

The `Half` struct is a 16-bit floating point type, and was introduced with .NET 5. `Half` is intended mainly for interoperating with graphics card processors and does not have native support in most CPUs.

You can convert between `Half` and `float` or `double` via an explicit cast:

```
Half h = (Half) 123.456;
Console.WriteLine (h);     // 123.44  (note loss of precision)
```

There are no arithmetic operations defined for this type, so you must convert to another type such as `float` or `double` in order to perform calculations.

`Half` has a range of -65500 to 65500:

```
Console.WriteLine (Half.MinValue);   // -65500
Console.WriteLine (Half.MaxValue);   // 65500
```

Note the loss of precision at the maximum range:

```
Console.WriteLine ((Half)65500);     // 65500
Console.WriteLine ((Half)65490);     // 65500
Console.WriteLine ((Half)65480);     // 65470
```

Complex

The `Complex` struct is another specialized numeric type that represents complex numbers with real and imaginary components of type `double`. `Complex` resides in the namespace (along with `BigInteger`).

To use `Complex`, instantiate the struct, specifying the real and imaginary values:

```
var c1 = new Complex (2, 3.5);
var c2 = new Complex (3, 0);
```

There are also implicit conversions from the standard numeric types.

The `Complex` struct exposes properties for the real and imaginary values as well as the phase and magnitude:

```
Console.WriteLine (c1.Real);       // 2
Console.WriteLine (c1.Imaginary);  // 3.5
Console.WriteLine (c1.Phase);      // 1.05165021254837
Console.WriteLine (c1.Magnitude);  // 4.03112887414927
```

You can also construct a `Complex` number by specifying magnitude and phase:

```
Complex c3 = Complex.FromPolarCoordinates (1.3, 5);
```

The standard arithmetic operators are overloaded to work on `Complex` numbers:

```
Console.WriteLine (c1 + c2);    // (5, 3.5)
Console.WriteLine (c1 * c2);    // (6, 10.5)
```

The `Complex` struct exposes static methods for more advanced functions, including the following:

- Trigonometric (`Sin`, `Asin`, `Sinh`, `Tan`, etc.)

- Logarithms and exponentiations
- `Conjugate`

Random

The `Random` class generates a pseudorandom sequence of random `byte`s, `integer`s, or `double`s.

To use `Random`, you first instantiate it, optionally providing a seed to initiate the random number series. Using the same seed guarantees the same series of numbers (if run under the same CLR version), which is sometimes useful when you want reproducibility:

```
Random r1 = new Random (1);
Random r2 = new Random (1);
Console.WriteLine (r1.Next (100) + ", " + r1.Next (100));      // 24, 11
Console.WriteLine (r2.Next (100) + ", " + r2.Next (100));      // 24, 11
```

If you don't want reproducibility, you can construct `Random` with no seed; in that case, it uses the current system time to make one up.

WARNING

Because the system clock has limited granularity, two `Random` instances created close together (typically within 10 ms) will yield the same sequence of values. A common trap is to instantiate a new `Random` object every time you need a random number rather than reusing the *same* object.

A good pattern is to declare a single static `Random` instance. In multithreaded scenarios, however, this can cause trouble because `Random` objects are not thread-safe. We describe a workaround in "Thread-Local Storage".

Calling `Next(`*n*`)` generates a random integer between 0 and *n*`−1`. `NextDouble` generates a random `double` between 0 and 1. `NextBytes` fills a byte array with random values.

From .NET 8, the `Random` class includes a `GetItems` method, which picks *n* random items from a collection. The following code picks two random numbers from a collection of five:

```
int[] numbers = { 10, 20, 30, 40, 50 };
int[] randomTwo = new Random().GetItems (numbers, 2);
```

From .NET 8, there's also a `Shuffle` method to randomize the order of items within an array or span.

`Random` is not considered random enough for high-security applications such as cryptography. For this, .NET provides a *cryptographically strong* random number generator, in the `System.Security.Cryptography` namespace. Here's how to use it:

```
var rand = System.Security.Cryptography.RandomNumberGenerator.Create();
byte[] bytes = new byte [32];
rand.GetBytes (bytes);       // Fill the byte array with random numbers.
```

The downside is that it's less flexible: filling a byte array is the only means of obtaining random numbers. To obtain an integer, you must use `BitConverter`:

```
byte[] bytes = new byte [4];
rand.GetBytes (bytes);
int i = BitConverter.ToInt32 (bytes, 0);
```

BitOperations

The `System.Numerics.BitOperations` class (from .NET 6) exposes the following methods to help with base-2 operations:

`IsPow2`

Returns true if a number is a power of 2

`LeadingZeroCount`/`TrailingZeroCount`

Returns the number of leading zeros, when formatted as a base-2 32-bit or 64-bit unsigned integer

`Log2`

Returns the integer base-2 log of an unsigned integer

`PopCount`

Returns the number of bits set to 1 in an unsigned integer

`RotateLeft/RotateRight`

Performs a bitwise left/right rotation

`RoundUpToPowerOf2`

Rounds an unsigned integer up to the closest power of 2

Enums

In Chapter 3, we described C#'s enum type and showed how to combine members, test equality, use logical operators, and perform conversions. .NET extends C#'s support for enums through the `System.Enum` type. This type has two roles:

- Providing type unification for all `enum` types
- Defining static utility methods

Type unification means that you can implicitly cast any enum member to a `System.Enum` instance:

```
Display (Nut.Macadamia);     // Nut.Macadamia
Display (Size.Large);        // Size.Large

void Display (Enum value)
{
  Console.WriteLine (value.GetType().Name + "." + value.ToString());
}

enum Nut  { Walnut, Hazelnut, Macadamia }
enum Size { Small, Medium, Large }
```

The static utility methods on `System.Enum` are primarily related to performing conversions and obtaining lists of members.

Enum Conversions

There are three ways to represent an enum value:

- As an `enum` member
- As its underlying integral value
- As a string

In this section, we describe how to convert between each.

Enum to integral conversions

Recall that an explicit cast converts between an `enum` member and its integral value. An explicit cast is the correct approach if you know the `enum` type at compile time:

```
[Flags]
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
int i = (int) BorderSides.Top;            // i == 4
BorderSides side = (BorderSides) i;       // side == BorderSides.Top
```

You can cast a `System.Enum` instance to its integral type in the same way. The trick is to first cast to an `object` and then the integral type:

```
static int GetIntegralValue (Enum anyEnum)
{
  return (int) (object) anyEnum;
}
```

This relies on you knowing the integral type: the method we just wrote would crash if passed an `enum` whose integral type was `long`. To write a method that works with an `enum` of any integral type, you can take one of three approaches. The first is to call `Convert.ToDecimal`:

```
static decimal GetAnyIntegralValue (Enum anyEnum)
{
  return Convert.ToDecimal (anyEnum);
}
```

This works because every integral type (including `ulong`) can be converted to decimal without loss of information. The second approach is to call `Enum.GetUnderlyingType` in order to obtain the `enum`'s integral type, and then call `Convert.ChangeType`:

```
static object GetBoxedIntegralValue (Enum anyEnum)
{
  Type integralType = Enum.GetUnderlyingType (anyEnum.GetType());
  return Convert.ChangeType (anyEnum, integralType);
}
```

This preserves the original integral type, as the following example shows:

```
object result = GetBoxedIntegralValue (BorderSides.Top);
Console.WriteLine (result);                              // 4
Console.WriteLine (result.GetType());                    // System.Int32
```

NOTE

Our `GetBoxedIntegralType` method in fact performs no value conversion; rather, it *reboxes* the same value in another type. It translates an integral value in *enum-type* clothing to an integral value in *integral-type* clothing. We describe this further in "How Enums Work".

The third approach is to call `Format` or `ToString` specifying the `"d"` or `"D"` format string. This gives you the `enum`'s integral value as a string, and it is useful when writing custom serialization formatters:

```
static string GetIntegralValueAsString (Enum anyEnum)
{
  return anyEnum.ToString ("D");      // returns something like "4"
}
```

Integral to enum conversions

`Enum.ToObject` converts an integral value to an `enum` instance of the given type:

```
object bs = Enum.ToObject (typeof (BorderSides), 3);
Console.WriteLine (bs);                              // Left, Right
```

This is the dynamic equivalent of the following:

```
BorderSides bs = (BorderSides) 3;
```

`ToObject` is overloaded to accept all integral types as well as `object`. (The latter works with any boxed integral type.)

String conversions

To convert an `enum` to a string, you can either call the static `Enum.Format` method or call `ToString` on the instance. Each method accepts a format string, which can be `"G"` for default formatting behavior, `"D"` to emit the underlying integral value as a string, `"X"` for the same in hexadecimal, or `"F"` to format combined members of an enum without the `Flags` attribute. We listed examples of these in “Standard Format Strings and Parsing Flags”.

`Enum.Parse` converts a string to an `enum`. It accepts the `enum` type and a string that can include multiple members:

```
BorderSides leftRight = (BorderSides) Enum.Parse (typeof (BorderSides),
                                                  "Left, Right");
```

An optional third argument lets you perform case-insensitive parsing. An `ArgumentException` is thrown if the member is not found.

Enumerating Enum Values

`Enum.GetValues` returns an array comprising all members of a particular `enum` type:

```
foreach (Enum value in Enum.GetValues (typeof (BorderSides)))
  Console.WriteLine (value);
```

Composite members such as `LeftRight = Left | Right` are included, too.

`Enum.GetNames` performs the same function, but returns an array of *strings*.

> **NOTE**
>
> Internally, the CLR implements `GetValues` and `GetNames` by reflecting over the fields in the `enum`'s type. The results are cached for efficiency.

How Enums Work

The semantics of `enum`s are enforced largely by the compiler. In the CLR, there's no runtime difference between an `enum` instance (when unboxed) and its underlying integral value. Further, an `enum` definition in the CLR is merely a subtype of `System.Enum` with static integral-type fields for each member. This makes the ordinary use of an `enum` highly efficient, with a runtime cost matching that of integral constants.

The downside of this strategy is that `enum`s can provide *static* but not *strong* type safety. We saw an example of this in Chapter 3:

```
[Flags] public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
BorderSides b = BorderSides.Left;
b += 1234;                          // No error!
```

When the compiler is unable to perform validation (as in this example), there's no backup from the runtime to throw an exception.

What we said about there being no runtime difference between an `enum` instance and its integral value might seem at odds with the following:

```
[Flags] public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
Console.WriteLine (BorderSides.Right.ToString());        // Right
Console.WriteLine (BorderSides.Right.GetType().Name);    // BorderSides
```

Given the nature of an `enum` instance at runtime, you'd expect this to print `2` and `Int32`! The reason for its behavior is down to some more compile-time trickery. C# explicitly *boxes* an `enum` instance before calling its virtual methods—such as `ToString` or `GetType`. And when an `enum` instance is boxed, it gains a runtime wrapping that references its `enum` type.

The Guid Struct

The `Guid` struct represents a globally unique identifier: a 16-byte value that, when generated, is almost certainly unique in the world. `Guid`s are often used for keys of various sorts, in applications and databases. There are 2^{128}, or 3.4×10^{38}, unique `Guid`s.

The static `Guid.NewGuid` method generates a unique `Guid`:

```
Guid g = Guid.NewGuid ();
Console.WriteLine (g.ToString());  // 0d57629c-7d6e-4847-97cb-9e2fc25083fe
```

To instantiate an existing value, you use one of the constructors. The two most useful constructors are:

```
public Guid (byte[] b);    // Accepts a 16-byte array
public Guid (string g);    // Accepts a formatted string
```

When represented as a string, a `Guid` is formatted as a 32-digit hexadecimal number, with optional hyphens after the 8th, 12th, 16th, and 20th digits. The whole string can also be optionally wrapped in brackets or braces:

```
Guid g1 = new Guid ("{0d57629c-7d6e-4847-97cb-9e2fc25083fe}");
Guid g2 = new Guid ("0d57629c7d6e484797cb9e2fc25083fe");
Console.WriteLine (g1 == g2);  // True
```

Being a struct, a `Guid` honors value-type semantics; hence, the equality operator works in the preceding example.

The `ToByteArray` method converts a `Guid` to a byte array.

The static `Guid.Empty` property returns an empty `Guid` (all zeros). This is often used in place of `null`.

Equality Comparison

Until now, we've assumed that the == and `!=` operators are all there is to equality comparison. The issue of equality, however, is more complex and subtler, sometimes requiring the use of additional methods and interfaces. This section explores the standard C# and .NET protocols for equality, focusing particularly on two questions:

- When are == and `!=` adequate—and inadequate—for equality comparison, and what are the alternatives?
- How and when should you customize a type's equality logic?

But before exploring the details of equality protocols and how to customize them, we first must look at the preliminary concept of value versus referential equality.

Value Versus Referential Equality

There are two kinds of equality:

Value equality

> Two values are *equivalent* in some sense.

Referential equality

> Two references refer to *exactly the same object*.

Unless overridden:

- Value types use *value equality*.
- Reference types use *referential equality*. (This is overridden with anonymous types and records.)

Value types, in fact, can use *only* value equality (unless boxed). A simple demonstration of value equality is to compare two numbers:

```
int x = 5, y = 5;
Console.WriteLine (x == y);   // True (by virtue of value equality)
```

A more elaborate demonstration is to compare two `DateTimeOffset` structs. The following prints `True` because the two `DateTimeOffset`s refer to the *same point in time* and so are considered equivalent:

```
var dt1 = new DateTimeOffset (2010, 1, 1, 1, 1, 1, TimeSpan.FromHours(8));
var dt2 = new DateTimeOffset (2010, 1, 1, 2, 1, 1, TimeSpan.FromHours(9));
Console.WriteLine (dt1 == dt2);   // True
```

NOTE

`DateTimeOffset` is a struct whose equality semantics have been tweaked. By default, structs exhibit a special kind of value equality called *structural equality* in which two values are considered equal if all of their members are equal. (You can see this by creating a struct and calling its `Equals` method; more on this later.)

Reference types exhibit referential equality by default. In the following example, `f1` and `f2` are not equal, despite their objects having identical content:

```
class Foo { public int X; }
...
Foo f1 = new Foo { X = 5 };
Foo f2 = new Foo { X = 5 };
Console.WriteLine (f1 == f2);   // False
```

In contrast, `f3` and `f1` are equal because they reference the same object:

```
Foo f3 = f1;
Console.WriteLine (f1 == f3);   // True
```

Later in this section, we explain how you can *customize* reference types to exhibit value equality. An example of this is the `Uri` class in the `System` namespace:

```
Uri uri1 = new Uri ("http://www.linqpad.net");
Uri uri2 = new Uri ("http://www.linqpad.net");
Console.WriteLine (uri1 == uri2);              // True
```

The `string` class exhibits similar behavior:

```
var s1 = "http://www.linqpad.net";
var s2 = "http://" + "www.linqpad.net";
Console.WriteLine (s1 == s2);       // True
```

Standard Equality Protocols

There are three standard protocols that types can implement for equality comparison:

- The == and `!=` operators
- The virtual `Equals` method in `object`
- The `IEquatable<T>` interface

In addition, there are the *pluggable* protocols and the `IStructuralEquatable` interface, which we describe in Chapter 7.

== and !=

We’ve already seen in many examples how the standard == and `!=` operators perform equality/inequality comparisons. The subtleties with == and `!=` arise because they are *operators*; thus, they are statically resolved (in fact, they are implemented as `static` functions). So, when you use == or `!=`, C# makes a *compile-time* decision as to which type will perform the comparison, and no `virtual` behavior comes into play. This is normally desirable. In the following example, the compiler hardwires == to the `int` type because `x` and `y` are both `int`:

```
int x = 5;
int y = 5;
Console.WriteLine (x == y);      // True
```

But in the next example, the compiler wires the == operator to the `object` type:

```
object x = 5;
object y = 5;
Console.WriteLine (x == y);      // False
```

Because `object` is a class (and so a reference type), `object`’s == operator uses *referential equality* to compare `x` and `y`. The result is `false` because `x` and `y` each refer to different boxed objects on the heap.

The virtual Object.Equals method

To correctly equate `x` and `y` in the preceding example, we can use the virtual `Equals` method. `Equals` is defined in `System.Object` and so is available to all types:

```
object x = 5;
object y = 5;
```

```
Console.WriteLine (x.Equals (y));      // True
```

Equals is resolved at runtime—according to the object's actual type. In this case, it calls Int32's Equals method, which applies *value equality* to the operands, returning true. With reference types, Equals performs referential equality comparison by default; with structs, Equals performs structural comparison by calling Equals on each of its fields.

WHY THE COMPLEXITY?

You might wonder why the designers of C# didn't avoid the problem by making == virtual and thus functionally identical to Equals. There are three reasons for this:

- If the first operand is null, Equals fails with a NullReferenceException; a static operator does not.
- Because the == operator is statically resolved, it executes extremely quickly. This means that you can write computationally intensive code without penalty—and without needing to learn another language such as C++.
- Sometimes it can be useful to have == and Equals apply different definitions of equality. We describe this scenario later in this section.

Essentially, the complexity of the design reflects the complexity of the situation: the concept of equality covers a multitude of scenarios.

Hence, Equals is suitable for equating two objects in a type-agnostic fashion. The following method equates two objects of any type:

```
public static bool AreEqual (object obj1, object obj2)
  => obj1.Equals (obj2);
```

There is one case, however, in which this fails. If the first argument is `null`, you get a `NullReferenceException`. Here's the fix:

```
public static bool AreEqual (object obj1, object obj2)
{
  if (obj1 == null) return obj2 == null;
  return obj1.Equals (obj2);
}
```

Or, more succinctly:

```
public static bool AreEqual (object obj1, object obj2)
  => obj1 == null ? obj2 == null : obj1.Equals (obj2);
```

The static object.Equals method

The `object` class provides a static helper method that does the work of `AreEqual` in the preceding example. Its name is `Equals`—just like the virtual method—but there's no conflict because it accepts *two* arguments:

```
public static bool Equals (object objA, object objB)
```

This provides a null-safe equality comparison algorithm for when the types are unknown at compile time:

```
object x = 3, y = 3;
Console.WriteLine (object.Equals (x, y));   // True
x = null;
Console.WriteLine (object.Equals (x, y));   // False
y = null;
Console.WriteLine (object.Equals (x, y));   // True
```

A useful application is when writing generic types. The following code will not compile if `object.Equals` is replaced with the `==` or `!=` operator:

```
class Test <T>
{
  T _value;
  public void SetValue (T newValue)
  {
    if (!object.Equals (newValue, _value))
    {
      _value = newValue;
      OnValueChanged();
    }
  }
  protected virtual void OnValueChanged() { ... }
}
```

Operators are prohibited here because the compiler cannot bind to the static method of an unknown type.

NOTE

A more elaborate way to implement this comparison is with the `EqualityComparer<T>` class. This has the advantage of avoiding boxing:

```
if (!EqualityComparer<T>.Default.Equals (newValue, _value))
```

We discuss `EqualityComparer<T>` in more detail in Chapter 7 (see "Plugging in Equality and Order").

The static object.ReferenceEquals method

Occasionally, you need to force referential equality comparison. The static `object.ReferenceEquals` method does just that:

```
Widget w1 = new Widget();
Widget w2 = new Widget();
Console.WriteLine (object.ReferenceEquals (w1, w2));     // False

class Widget { ... }
```

You might want to do this because it's possible for `Widget` to override the virtual `Equals` method such that `w1.Equals(w2)` would return `true`. Further, it's possible for `Widget` to overload the == operator so that `w1==w2` would also return `true`. In such cases, calling `object.ReferenceEquals` guarantees normal referential equality semantics.

NOTE

Another way to force referential equality comparison is to cast the values to `object` and then apply the == operator.

The IEquatable<T> interface

A consequence of calling `object.Equals` is that it forces boxing on value types. This is undesirable in highly performance-sensitive scenarios because boxing is relatively expensive compared to the actual comparison. A solution was introduced in C# 2.0, with the `IEquatable<T>` interface:

```
public interface IEquatable<T>
{
  bool Equals (T other);
}
```

The idea is that `IEquatable<T>`, when implemented, gives the same result as calling `object`'s virtual `Equals` method—but more quickly. Most basic .NET types implement `IEquatable<T>`. You can use `IEquatable<T>` as a constraint in a generic type:

```
class Test<T> where T : IEquatable<T>
{
  public bool IsEqual (T a, T b)
  {
    return a.Equals (b);     // No boxing with generic T
  }
}
```

If we remove the generic constraint, the class would still compile, but `a.Equals(b)` would instead bind to the slower `object.Equals` (slower assuming `T` was a value type).

When Equals and == are not equal

We said earlier that it's sometimes useful for == and `Equals` to apply different definitions of equality. For example:

```
double x = double.NaN;
Console.WriteLine (x == x);            // False
Console.WriteLine (x.Equals (x));      // True
```

The `double` type's == operator enforces that one `NaN` can never equal anything else—even another `NaN`. This is most natural from a mathematical perspective, and it reflects the underlying CPU behavior. The `Equals` method, however, is obliged to apply *reflexive* equality; in other words:

> `x.Equals (x)` must *always* return true.

Collections and dictionaries rely on `Equals` behaving this way; otherwise, they could not find an item they previously stored.

Having `Equals` and == apply different definitions of equality is actually quite rare with value types. A more common scenario is with reference types; this happens when the author customizes `Equals` so that it performs value equality while leaving == to perform (default) referential equality. The `StringBuilder` class does exactly that:

```
var sb1 = new StringBuilder ("foo");
var sb2 = new StringBuilder ("foo");
Console.WriteLine (sb1 == sb2);          // False (referential equality)
Console.WriteLine (sb1.Equals (sb2));    // True  (value equality)
```

Let's now look at how to customize equality.

Equality and Custom Types

Recall default equality comparison behavior:

- Value types use *value equality*.

- Reference types use *referential equality* unless overridden (as is the case with anonymous types and records).

Further:

- A struct's `Equals` method applies *structural value equality* by default (i.e., it compares each field in the struct).

Sometimes, it makes sense to override this behavior when writing a type. There are two cases for doing so:

- To change the meaning of equality
- To speed up equality comparisons for structs

Changing the meaning of equality

Changing the meaning of equality makes sense when the default behavior of `==` and `Equals` is unnatural for your type and is *not what a consumer would expect*. An example is `DateTimeOffset`, a struct with two private fields: a UTC `DateTime` and a numeric integer offset. If you were writing this type, you'd probably want to ensure that equality comparisons considered only the UTC `DateTime` field and not the offset field. Another example is numeric types that support `NaN` values such as `float` and `double`. If you were implementing such types yourself, you'd want to ensure that `NaN`-comparison logic was supported in equality comparisons.

With classes, it's sometimes more natural to offer *value equality* as the default instead of *referential equality*. This is often the case with small classes that hold a simple piece of data, such as `System.Uri` (or `System.String`).

With records, the compiler automatically implements structural equality (by comparing each field). Sometimes, however, this will include fields that you don't want to compare, or objects that require special comparison logic, such as collections. The process of overriding equality with records is slightly different because records follow a special pattern that's designed to play well with its rules for inheritance.

Speeding up equality comparisons with structs

The default *structural equality* comparison algorithm for structs is relatively slow. Taking over this process by overriding `Equals` can improve performance by a factor of five. Overloading the `==` operator and implementing `IEquatable<T>` allows unboxed equality comparisons, and this can speed things up by a factor of five again.

> **NOTE**
>
> Overriding equality semantics for reference types doesn't benefit performance. The default algorithm for referential equality comparison is already very fast because it simply compares two 32- or 64-bit references.

There's another, rather peculiar case for customizing equality, and that's to improve a struct's hashing algorithm for better performance in a hashtable. This comes as a result of the fact that equality comparison and hashing are joined at the hip. We examine hashing in a moment.

How to override equality semantics

To override equality with classes or structs, here are the steps:

1. Override `GetHashCode()` and `Equals()`.
2. (Optionally) overload `!=` and `==`.
3. (Optionally) implement `IEquatable<T>`.

The process is different (and simpler) with records because the compiler already overrides the equality methods and operators in line with its own special pattern. If you want to intervene, you must conform to this pattern, which means writing an `Equals` method with a signature like this:

```
record Test (int X, int Y)
{
  public virtual bool Equals (Test t) => t != null && t.X == X && t.Y == Y;
}
```

Notice that `Equals` is `virtual` (not `override`) and accepts the actual record type (`Test` in this case, and not `object`). The compiler will recognize that your method has the "correct" signature and will patch it in.

You must also override `GetHashCode()`, just as you would with classes or structs. You don't need to (and shouldn't) overload `!=` and `==`, or implement `IEquatable<T>`, because this is already done for you.

Overriding GetHashCode

It might seem odd that `System.Object`—with its small footprint of members—defines a method with a specialized and narrow purpose. `GetHashCode` is a virtual method in `Object` that fits this description; it exists primarily for the benefit of just the following two types:

```
System.Collections.Hashtable
System.Collections.Generic.Dictionary<TKey,TValue>
```

These are *hashtables*—collections for which each element has a key used for storage and retrieval. A hashtable applies a very specific strategy for efficiently allocating elements based on their key. This requires that each key have an `Int32` number, or *hash code*. The hash code need not be unique for each key, but should be as varied as possible for good hashtable performance. Hashtables are considered important enough that `GetHashCode` is defined in `System.Object`—so that every type can emit a hash code.

NOTE

We describe hashtables in detail in Chapter 7.

Both reference and value types have default implementations of `GetHashCode`, meaning that you don't need to override this method—*unless you override* `Equals`. (And if you override `GetHashCode`, you will almost certainly want to also override `Equals`.)

Here are the other rules for overriding `object.GetHashCode`:

- It must return the same value on two objects for which `Equals` returns `true` (hence, `GetHashCode` and `Equals` are overridden together).
- It must not throw exceptions.
- It must return the same value if called repeatedly on the same object (unless the object has *changed*).

For maximum performance in hashtables, you should write `GetHashCode` so as to minimize the likelihood of two different values returning the same hashcode. This gives rise to the third reason for overriding `Equals` and `GetHashCode` on structs, which is to provide a more efficient hashing algorithm than the default. The default implementation for structs is at the discretion of the runtime and can be based on every field in the struct.

In contrast, the default `GetHashCode` implementation for *classes* is based on an internal object token, which is unique for each instance in the CLR's current implementation.

WARNING

If an object's hashcode changes after it's been added as a key to a dictionary, the object will no longer be accessible in the dictionary. You can preempt this by basing hashcode calculations on immutable fields.

We provide a complete example illustrating how to override `GetHashCode` shortly.

Overriding Equals

The axioms for `object.Equals` are as follows:

- An object cannot equal `null` (unless it's a nullable type).
- Equality is *reflexive* (an object equals itself).
- Equality is *commutative* (if `a.Equals(b)`, then `b.Equals(a)`).
- Equality is *transitive* (if `a.Equals(b)` and `b.Equals(c)`, then `a.Equals(c)`).
- Equality operations are repeatable and reliable (they don't throw exceptions).

Overloading == and !=

In addition to overriding `Equals`, you can optionally overload the equality and inequality operators. This is nearly always done with structs because the consequence of not doing so is that the == and `!=` operators will simply not work on your type.

With classes, there are two ways to proceed:

- Leave == and `!=` alone—so that they apply referential equality.
- Overload == and `!=` in line with `Equals`.

The first approach is most common with custom types—especially *mutable* types. It ensures that your type follows the expectation that == and `!=` should exhibit referential equality with reference types, and this avoids confusing consumers. We saw an example earlier:

```
var sb1 = new StringBuilder ("foo");
var sb2 = new StringBuilder ("foo");
Console.WriteLine (sb1 == sb2);          // False (referential equality)
Console.WriteLine (sb1.Equals (sb2));    // True  (value equality)
```

The second approach makes sense with types for which a consumer would never want referential equality. These are typically immutable—such as the `string` and `System.Uri` classes—and are sometimes good candidates for `struct`s.

NOTE

Although it’s possible to overload `!=` such that it means something other than `!(==)`, this is rarely done in practice. An example is with the types defined in the `System.Data.SqlTypes` namespace that represent native column types in SQL Server. These follow the null comparison logic of databases, whereby the = and <> operators (== and `!=` in C#) both return null if either operand is null.

Implementing IEquatable<T>

For completeness, it’s also good to implement `IEquatable<T>` when overriding `Equals`. Its results should always match those of the overridden object’s `Equals` method. Implementing `IEquatable<T>` comes at no programming cost if you structure your `Equals` method implementation as in the example that follows in a moment.

An example: the Area struct

Imagine that we need a struct to represent an area whose width and height are interchangeable. In other words, 5 × 10 is equal to 10 × 5. (Such a type would be suitable in an algorithm that arranges rectangular shapes.)

Here’s the complete code:

```
public struct Area : IEquatable <Area>
{
  public readonly int Measure1;
  public readonly int Measure2;

  public Area (int m1, int m2)
  {
    Measure1 = Math.Min (m1, m2);
    Measure2 = Math.Max (m1, m2);
  }

  public override bool Equals (object other)
```

```
    => other is Area a && Equals (a);    // Calls method below

  public bool Equals (Area other)        // Implements IEquatable<Area>
    => Measure1 == other.Measure1 && Measure2 == other.Measure2;

  public override int GetHashCode()
    => HashCode.Combine (Measure1, Measure2);

  // Note that we call the static Equals method in the object class: this
  // does null checking before calling our own (instance) Equals method.
  public static bool operator == (Area a1, Area a2) => Equals (a1, a2);

  public static bool operator != (Area a1, Area a2) => !(a1 == a2);
}
```

NOTE

From C# 10, you can shortcut the process with records. By declaring this as a `record struct`, you can remove all the code following the constructor.

In implementing `GetHashCode`, we used .NET's `HashCode.Combine` function to produce a composite hashcode. (Before that function existed, a popular approach was to multiply each value by some prime number and then add them together.)

Here's a demonstration of the `Area` struct:

```
Area a1 = new Area (5, 10);
Area a2 = new Area (10, 5);
Console.WriteLine (a1.Equals (a2));    // True
Console.WriteLine (a1 == a2);          // True
```

Pluggable equality comparers

If you want a type to take on different equality semantics just for a specific scenario, you can use a pluggable `IEqualityComparer`. This is particularly useful in conjunction with the standard collection classes, and we describe it in the following chapter, in "Plugging in Equality and Order".

Order Comparison

As well as defining standard protocols for equality, C# and .NET define two standard protocols for determining the order of one object relative to another:

- The `IComparable` interfaces (`IComparable` and `IComparable<T>`)
- The `>` and `<` operators

The `IComparable` interfaces are used by general-purpose sorting algorithms. In the following example, the static `Array.Sort` method works because `System.String` implements the `IComparable` interfaces:

```
string[] colors = { "Green", "Red", "Blue" };
Array.Sort (colors);
foreach (string c in colors) Console.Write (c + " ");   // Blue Green Red
```

The `<` and `>` operators are more specialized, and they are intended mostly for numeric types. Because they are statically resolved, they can translate to highly efficient bytecode, suitable for computationally intensive algorithms.

.NET also provides pluggable ordering protocols, via the `IComparer` interfaces. We describe these in the final section of Chapter 7.

IComparable

The `IComparable` interfaces are defined as follows:

```
public interface IComparable       { int CompareTo (object other); }
public interface IComparable<in T> { int CompareTo (T other);      }
```

The two interfaces represent the same functionality. With value types, the generic type-safe interface is faster than the nongeneric interface. In both cases, the `CompareTo` method works as follows:

- If `a` comes after `b`, `a.CompareTo(b)` returns a positive number.

- If `a` is the same as `b`, `a.CompareTo(b)` returns `0`.
- If `a` comes before `b`, `a.CompareTo(b)` returns a negative number.

For example:

```
Console.WriteLine ("Beck".CompareTo ("Anne"));       // 1
Console.WriteLine ("Beck".CompareTo ("Beck"));       // 0
Console.WriteLine ("Beck".CompareTo ("Chris"));      // -1
```

Most of the base types implement both `IComparable` interfaces. These interfaces are also sometimes implemented when writing custom types. We provide an example shortly.

IComparable versus Equals

Consider a type that both overrides `Equals` and implements the `IComparable` interfaces. You'd expect that when `Equals` returns `true`, `CompareTo` should return `0`. And you'd be right. But here's the catch:

> When `Equals` returns `false`, `CompareTo` can return what it likes (as long as it's internally consistent)!

In other words, equality can be "fussier" than comparison, but not vice versa (violate this and sorting algorithms will break). So, `CompareTo` can say, "All objects are equal," whereas `Equals` says, "But some are more equal than others!"

A great example of this is `System.String`. `String`'s `Equals` method and == operator use *ordinal* comparison, which compares the Unicode point values of each character. Its `CompareTo` method, however, uses a *culture-dependent* comparison, which sometimes puts more than one character into the same sorting position.

In Chapter 7, we discuss the pluggable ordering protocol, `IComparer`, which allows you to specify an alternative ordering algorithm when sorting or instantiating a sorted collection. A custom `IComparer` can further extend the gap between `CompareTo` and `Equals`—a case-insensitive string comparer, for instance, will return `0` when comparing `"A"` and `"a"`. The reverse rule still applies, however: `CompareTo` can never be fussier than `Equals`.

> **NOTE**
>
> When implementing the `IComparable` interfaces in a custom type, you can avoid running afoul of this rule by writing the first line of `CompareTo` as follows:
>
> ```
> if (Equals (other)) return 0;
> ```
>
> After that, it can return what it likes, as long as it's consistent!

< and >

Some types define `<` and `>` operators; for instance:

```
bool after2010 = DateTime.Now > new DateTime (2010, 1, 1);
```

You can expect the `<` and `>` operators, when implemented, to be functionally consistent with the `IComparable` interfaces. This is standard practice across .NET.

It's also standard practice to implement the `IComparable` interfaces whenever `<` and `>` are overloaded, although the reverse is not true. In fact, most .NET types that implement `IComparable` *do not* overload `<` and `>`. This differs from the situation with equality for which it's normal to overload `==` when overriding `Equals`.

Typically, `>` and `<` are overloaded only when:

- A type has a strong intrinsic concept of "greater than" and "less than" (versus `IComparable`'s broader concepts of "comes before" and "comes after").

- There is only one way *or context* in which to perform the comparison.
- The result is invariant across cultures.

System.String doesn't satisfy the last point: the results of string comparisons can vary according to language. Hence, string doesn't support the > and < operators:

```
bool error = "Beck" > "Anne";       // Compile-time error
```

Implementing the IComparable Interfaces

In the following struct representing a musical note, we implement the IComparable interfaces as well as overloading the < and > operators. For completeness, we also override Equals/GetHashCode and overload == and !=:

```
public struct Note : IComparable<Note>, IEquatable<Note>, IComparable
{
  int _semitonesFromA;
  public int SemitonesFromA { get { return _semitonesFromA; } }

  public Note (int semitonesFromA)
  {
    _semitonesFromA = semitonesFromA;
  }

  public int CompareTo (Note other)            // Generic IComparable<T>
  {
    if (Equals (other)) return 0;    // Fail-safe check
    return _semitonesFromA.CompareTo (other._semitonesFromA);
  }

  int IComparable.CompareTo (object other)     // Nongeneric IComparable
  {
    if (!(other is Note))
      throw new InvalidOperationException ("CompareTo: Not a note");
    return CompareTo ((Note) other);
  }

  public static bool operator < (Note n1, Note n2)
     => n1.CompareTo (n2) < 0;
  public static bool operator > (Note n1, Note n2)
    => n1.CompareTo (n2) > 0;

  public bool Equals (Note other)    // for IEquatable<Note>
    => _semitonesFromA == other._semitonesFromA;

  public override bool Equals (object other)
  {
    if (!(other is Note)) return false;
    return Equals ((Note) other);
  }

  public override int GetHashCode() => _semitonesFromA.GetHashCode();

  // Call the static Equals method to ensure nulls are properly handled:
  public static bool operator == (Note n1, Note n2) => Equals (n1, n2);

  public static bool operator != (Note n1, Note n2) => !(n1 == n2);
}
```

Utility Classes

Console

The static `Console` class handles standard input/output for console-based applications. In a command-line (console) application, the input comes from the keyboard via `Read`, `ReadKey`, and `ReadLine`, and the output goes to the text window via `Write` and `WriteLine`. You can control the window's position and dimensions with the properties `WindowLeft`, `WindowTop`, `WindowHeight`, and `WindowWidth`. You can also change the `BackgroundColor` and `ForegroundColor` properties and manipulate the cursor with the `CursorLeft`, `CursorTop`, and `CursorSize` properties:

```
Console.WindowWidth = Console.LargestWindowWidth;
Console.ForegroundColor = ConsoleColor.Green;
Console.Write ("test... 50%");
Console.CursorLeft -= 3;
Console.Write ("90%");     // test... 90%
```

The Write and WriteLine methods are overloaded to accept a composite format string (see String.Format in "String and Text Handling"). However, neither method accepts a format provider, so you're stuck with CultureInfo.CurrentCulture. (The workaround, of course, is to explicitly call string.Format.)

The Console.Out property returns a TextWriter. Passing Console.Out to a method that expects a TextWriter is a useful way to get that method to write to the Console for diagnostic purposes.

You can also redirect the Console's input and output streams via the SetIn and SetOut methods:

```
// First save existing output writer:
System.IO.TextWriter oldOut = Console.Out;

// Redirect the console's output to a file:
using (System.IO.TextWriter w = System.IO.File.CreateText
                                ("e:\\output.txt"))
{
  Console.SetOut (w);
  Console.WriteLine ("Hello world");
}

// Restore standard console output
Console.SetOut (oldOut);
```

In Chapter 15, we describe how streams and text writers work.

NOTE

When running WPF or Windows Forms applications under Visual Studio, the Console's output is automatically redirected to Visual Studio's output window (in debug mode). This can make Console.Write useful for diagnostic purposes; although in most cases, the Debug and Trace classes in the System.Diagnostics namespace are more appropriate (see Chapter 13).

Environment

The static `System.Environment` class provides a range of useful properties:

Files and folders

`CurrentDirectory`, `SystemDirectory`, `CommandLine`

Computer and operating system

`MachineName`, `ProcessorCount`, `OSVersion`, `NewLine`

User logon

`UserName`, `UserInteractive`, `UserDomainName`

Diagnostics

`TickCount`, `StackTrace`, `WorkingSet`, `Version`

You can obtain additional folders by calling `GetFolderPath`; we describe this in "File and Directory Operations" in Chapter 15.

You can access OS environment variables (what you see when you type "set" at the command prompt) with the following three methods: `GetEnvironmentVariable`, `GetEnvironmentVariables`, and `SetEnvironmentVariable`.

The `ExitCode` property lets you set the return code—for when your program is called from a command or batch file—and the `FailFast` method terminates a program immediately, without performing cleanup.

The `Environment` class available to Windows Store apps offers just a limited number of members (`ProcessorCount`, `NewLine`, and `FailFast`).

Process

The `Process` class in `System.Diagnostics` allows you to launch a new process. (In Chapter 13, we describe how you can also use it to interact with other processes running on the computer).

WARNING

For security reasons, the `Process` class is not available to Windows Store apps, and you cannot start arbitrary processes. Instead, you must use the `Windows.System.Launcher` class to “launch” a URI or file to which you have access; for example:

```
Launcher.LaunchUriAsync (new Uri ("http://albahari.com"));

var file = await KnownFolders.DocumentsLibrary
                             .GetFileAsync ("foo.txt");
Launcher.LaunchFileAsync (file);
```

This opens the URI or file, using whatever program is associated with the URI scheme or file extension. Your program must be in the foreground for this to work.

The static `Process.Start` method has several overloads; the simplest accepts a simple filename with optional arguments:

```
Process.Start ("notepad.exe");
Process.Start ("notepad.exe", "e:\\file.txt");
```

The most flexible overload accepts a `ProcessStartInfo` instance. With this, you can capture and redirect the launched process’s input, output, and error output (if you leave `UseShellExecute` as `false`). The following captures the output of calling `ipconfig`:

```
ProcessStartInfo psi = new ProcessStartInfo
{
  FileName = "cmd.exe",
  Arguments = "/c ipconfig /all",
  RedirectStandardOutput = true,
  UseShellExecute = false
};
Process p = Process.Start (psi);
string result = p.StandardOutput.ReadToEnd();
Console.WriteLine (result);
```

If you don’t redirect output, `Process.Start` executes the program in parallel to the caller. If you want to wait for the new process to complete, you can call `WaitForExit` on the `Process` object, with an optional timeout.

Redirecting output and error streams

With `UseShellExecute` false (the default in .NET), you can capture the standard input, output, and error streams and then write/read these streams via the `StandardInput`, `StandardOutput`, and `StandardError` properties.

A difficulty arises when you need to redirect both the standard output and standard error streams, in that you can't usually know in which order to read data from each (because you don't know in advance how the data will be interleaved). The solution is to read from both streams at once, which you can accomplish by reading from (at least) one of the streams *asynchronously*. Here's how to do this:

- Handle the `OutputDataReceived` and/or `ErrorDataReceived` events. These events fire when output/error data is received.
- Call `BeginOutputReadLine` and/or `BeginErrorReadLine`. This enables the aforementioned events.

The following method runs an executable while capturing both the output and error streams:

```
(string output, string errors) Run (string exePath, string args = "")
{
  using var p = Process.Start (new ProcessStartInfo (exePath, args)
  {
    RedirectStandardOutput = true,
    RedirectStandardError = true,
    UseShellExecute = false,
  });

  var errors = new StringBuilder ();

  // Read from the error stream asynchronously...
  p.ErrorDataReceived += (sender, errorArgs) =>
  {
    if (errorArgs.Data != null) errors.AppendLine (errorArgs.Data);
  };
  p.BeginErrorReadLine ();
```

```
    // ...while we read from the output stream synchronously:
    string output = p.StandardOutput.ReadToEnd();

    p.WaitForExit();
    return (output, errors.ToString());
  }
```

UseShellExecute

WARNING

In .NET 5+ (and .NET Core), the default for `UseShellExecute` is false, whereas in .NET Framework, it was true. Because this is a breaking change, it's worth checking all calls to `Process.Start` when porting code from .NET Framework.

The `UseShellExecute` flag changes how the CLR starts the process. With `UseShellExecute` true, you can do the following:

- Specify a path to a file or document rather than an executable (resulting in the operating system opening the file or document with its associated application)
- Specify a URL (resulting in the operating system navigating to that URL in the default web browser)
- (Windows only) Specify a Verb (such as "runas", to run the process with administrative elevation)

The drawback is that you cannot redirect the input or output streams. Should you need to do so—while launching a file or document—a workaround is to set `UseShellExecute` to false and invoke the command-line process (cmd.exe) with the "/c" switch, as we did earlier when calling *ipconfig*.

Under Windows, `UseShellExecute` instructs the CLR to use the Windows *ShellExecute* function instead of the *CreateProcess* function. Under Linux, `UseShellExecute` instructs the CLR to call *xdg-open*, *gnome-open*, or *kfmclient*.

AppContext

The static `System.AppContext` class exposes two useful properties:

- `BaseDirectory` returns the folder in which the application started. This folder is important for resolving assemblies (finding and loading dependencies) and locating configuration files (such as *appsettings.json*).
- `TargetFrameworkName` tells you the name and version of the .NET runtime that the application targets (as specified in its *.runtimeconfig.json* file). This might be older than the runtime actually in use.

In addition, the `AppContext` class manages a global string-keyed dictionary of Boolean values, intended to offer library writers a standard mechanism for allowing consumers to switch new features on or off. This untyped approach makes sense with experimental features that you want to keep undocumented to the majority of users.

The consumer of a library requests that you enable a feature as follows:

```
AppContext.SetSwitch ("MyLibrary.SomeBreakingChange", true);
```

Code within that library can then check for that switch as follows:

```
bool isDefined, switchValue;
isDefined = AppContext.TryGetSwitch ("MyLibrary.SomeBreakingChange",
                                     out switchValue);
```

`TryGetSwitch` returns false if the switch is undefined; this lets you distinguish an undefined switch from one whose value is set to false, should this be necessary.

NOTE

Ironically, the design of `TryGetSwitch` illustrates how not to write APIs. The `out` parameter is unnecessary, and the method should instead return a nullable `bool` whose value is true, false, or null for undefined. This would then enable the following use:

```
bool switchValue = AppContext.GetSwitch ("...") ?? false;
```