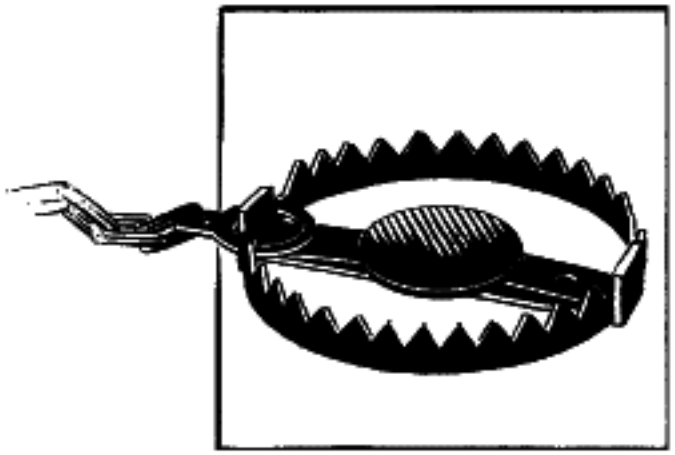


Another difference is that == does not work reliably on strings if the variables are cast to the `object` type. We explain why this is so in "Equality Comparison" on page 245.

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 before, after, 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 maps American culture).

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 up-

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.

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

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,
  CurrentCultureIgnoreCase,
  InvariantCulture,
```

```
    CurrentCultureIgnoreCase,
    InvariantCulture,
    InvariantCultureIgnoreCase,
    Ordinal,
    OrdinalIgnoreCase
}

// Case-sensitive
// Case-sensitive
// Case-sensitive
```


For example:

For example:

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

```
// True
// False
// ?
```

```
Console.WriteLine ("ü" == "ü");
Console.WriteLine (string.Equals ("ü", "ü",
                   StringComparison.CurrentCulture));
```

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

String order comparison

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);
```

The `CompareTo` instance method implements the generic `IComparable` interface, a standard comparison protocol used across the .NET Framework. This means `string`'s `CompareTo` defines the default ordering behavior strings, in such applications as sorted collections, for instance. For more information on `IComparable`, see "Order Comparison" on page 255.

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

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"));
Console.WriteLine ("Boston".CompareTo ("Boston"));
Console.WriteLine ("Boston".CompareTo ("Chicago"));
Console.WriteLine ("ü".CompareTo ("ū"));
Console.WriteLine ("foo".CompareTo ("FOO"));
```

```
// 1
// 0
```

```
// 0
// -1
 // 0
// -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

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 above 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 + ",");
```

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

```
Console.WriteLine (sb.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,
```

In our example, the expression `i + ","` means that we're still repeatedly concatenating strings. However, this incurs only a small performance cost in that the strings in question are small and don't grow with each loop iteration. For maximum performance, however, we could change the loop body to this:

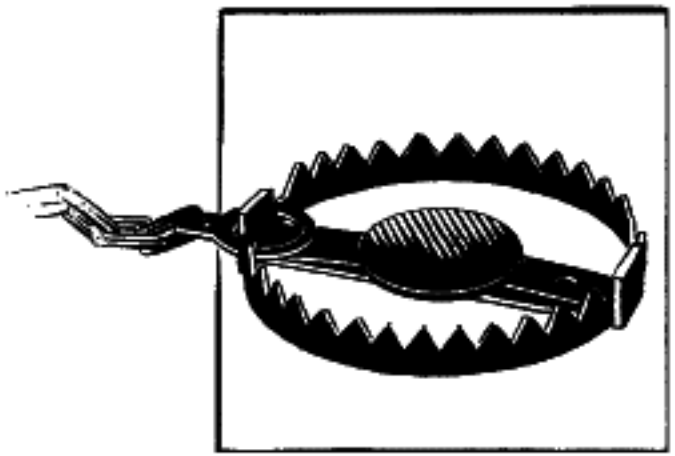
```
{ sb.Append (i.ToString()); sb.Append (","); }
```

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

As well as the `Insert`, `Remove`, and `Replace` methods (`Replace` functions such as string's `Replace`), `StringBuilder` defines a `Length` property and a writable indexer for getting/setting individual characters.

getting/setting individual characters.

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



Setting a `StringBuilder`'s `Length` to zero doesn't shrink its *internal* capacity. So, if the `StringBuilder` previously contained 1 million characters, it will continue to occupy around 2 MB of memory after zeroing its `Length`. If you want to release the memory, you must create a new `StringBuilder` and allow the old one

ory, 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 1 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 127 characters of the Unicode set, which covers most of what you see on a U.S.-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

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 impacting 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 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, and is what .NET uses internally to represent characters and strings. Some programs also write files in

UTF-16 uses one or two 16-bit words to represent each character, and 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 easiest way to instantiate a correctly configured class is to call `Encoding.GetEncoding` with a standard IANA name:

```
Encoding utf8 = Encoding.GetEncoding ("utf-8");
Encoding chinese = Encoding.GetEncoding ("GB18030");
```

The most common encodings can also be obtained through dedicated static prop-

The most common encodings can also 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`

The static `GetEncodings` method returns a list of all supported encodings, 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.

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:

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.

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

We resume this subject in Chapter 14, in "Stream Adapters" on page 552.

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`:

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);
Console.WriteLine (utf16Bytes.Length);
Console.WriteLine (utf32Bytes.Length);
```

```
// 10
// 20
// 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);
Console.WriteLine (original2);
Console.WriteLine (original3);

// 0123456789
// 0123456789
// 0123456789
```

UTF-16 and surrogate pairs

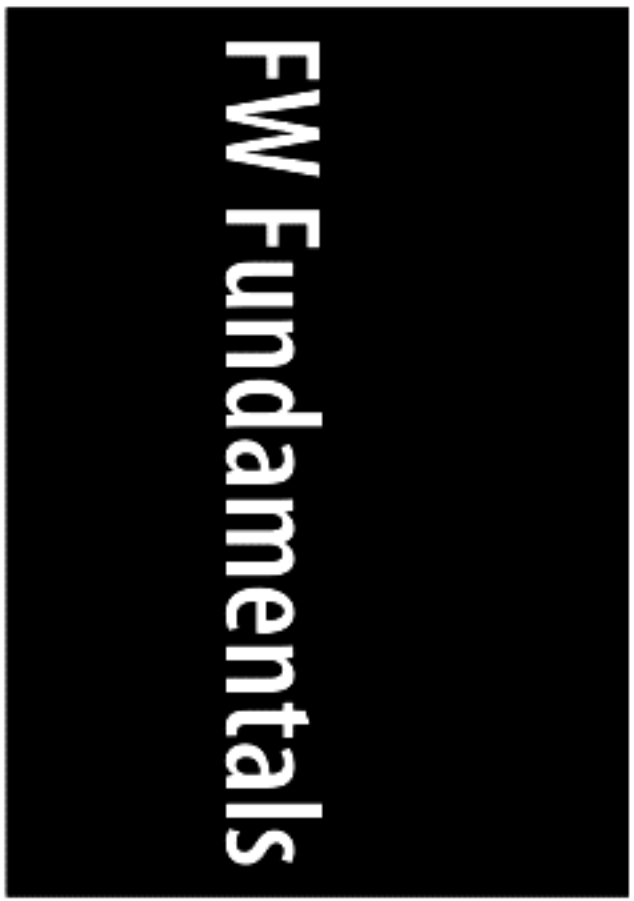
Recall that .NET stores characters and strings in UTF-16. Because UTF-16 requires one or two 16-bit words per character, and a `char` is only 16 bits in length, some Unicode characters require two `char`s to represent. This has a couple of consequences:

consequences:

-
-

A string's `Length` property may 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, and some less common Chinese characters.

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 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

Three immutable structs in the `System` namespace do the job of representing dates and times: `DateTime`, `DateTimeOffset`, and `TimeSpan`. 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:

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 (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);
```


For example:

```
Console.WriteLine (new TimeSpan (2, 30, 0));
Console.WriteLine (TimeSpan.FromHours (2.5));
```

```
    Console.WriteLine (TimeSpan.FromHours (2.5));
    Console.WriteLine (TimeSpan.FromHours (-2.5));
```

```
// 02:30:00
// 02:30:00
// -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);
Console.WriteLine (nearlyTenDays.Hours);
Console.WriteLine (nearlyTenDays.Minutes);
Console.WriteLine (nearlyTenDays.Seconds);
Console.WriteLine (nearlyTenDays.Milliseconds);

// 9
// 23
// 59
// 59
// 0
```

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

```
Console.WriteLine (nearlyTenDays.TotalDays);          // 9.9999842592593
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` was added in Framework 3.5 and is functionally similar to `DateTime`. Its distinguishing feature is that it also stores a UTC offset; this allows more meaningful results when comparing values across different time zones.

more meaningful results when comparing values across different time zones.

An excellent article on the rationale behind the introduction of `DateTimeOffset` is available on the MSDN BCL blogs. The title is "A Brief History of DateTime," by Anthony Moore.

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 local time on the current computer

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`:

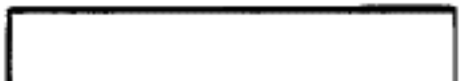
```
July 01 2007 03:00:00 -06:00
```

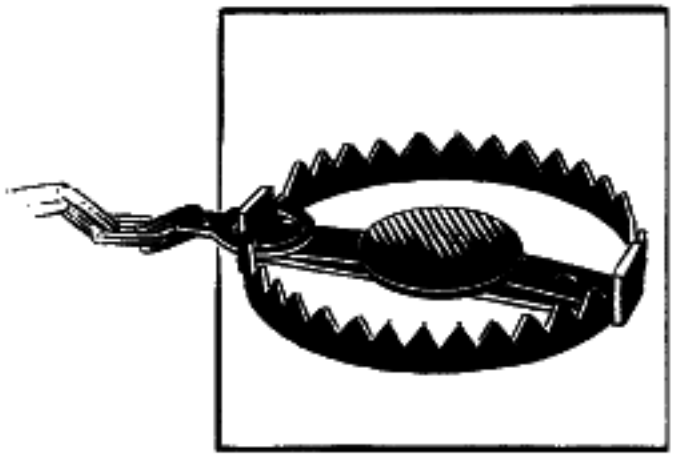
This influences equality comparisons, which is the main factor in choosing between `DateTime` and `DateTimeOffset`. Specifically:

-
-

`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*.





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 2007 09:00:00 +00:00 (GMT)
July 01 2007 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 right 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.

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.

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

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" on page 213.

SQL Server 2008 introduces 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, and millisecond:

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

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

```
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, nor, 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. For example:

```
DateTime d = new DateTime (5767, 1, 1,
                           new System.Globalization.HebrewCalendar());
Console.WriteLine (d);
// 12/12/2006 12:00:00 AM
```

```
// 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" on page 212.

Constructing a DateTimeOffset

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

or with an implicit cast:

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

The implicit cast from `DateTime` to `DateTimeOffset` is handy because most of the .NET Framework 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.

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

The current DateTime/DateTimeOffset

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

```
Console.WriteLine (DateTime.Now);
Console.WriteLine (DateTimeOffset.Now);

// 11/11/2007 1:23:45 PM
// 11/11/2007 1:23:45 PM -06:00
```

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

```
Console.WriteLine (DateTime.Today);      // 11/11/2007 12:00:00 AM
```

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

```
Console.WriteLine (DateTime.UtcNow);         // 11/11/2007 7:23:45 AM
Console.WriteLine (DateTimeOffset.UtcNow);   // 11/11/2007 7:23:45 AM +00:00
```

```
Console.WriteLine (DateTimeOffset.UtcNow);  // 11/11/2007 7:23:45 AM +00:00
```

The precision of all these methods depends on the operating system and is typically in the 10–20 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);
Console.WriteLine (dt.Month);
Console.WriteLine (dt.Day);
Console.WriteLine (dt.DayOfWeek);
```

```
Console.WriteLine (dt.DayOfWeek);
Console.WriteLine (dt.DayOfYear);

Console.WriteLine (dt.Hour);
Console.WriteLine (dt.Minute);
Console.WriteLine (dt.Second);
Console.WriteLine (dt.Millisecond);
Console.WriteLine (dt.Ticks);
Console.WriteLine (dt.TimeOfDay);

// 2000
// 2
```

```
// 2
// 3
// Thursday
// 34
// 10
// 20
// 30
// 0
// 630851700300000000
// 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`
`AddHours`

AddHours

AddMonths

AddMinutes

AddDays

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

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));          // 3/02/2000 11:50:30 AM
Console.WriteLine (dt + ts);              // 3/02/2000 11:50:30 AM
```

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 (2007, 1, 1);
DateTime nextYear = thisYear.AddYears (1);
TimeSpan oneYear = nextYear - thisYear;
```

Formatting and parsing

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

```
13/02/2000 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.

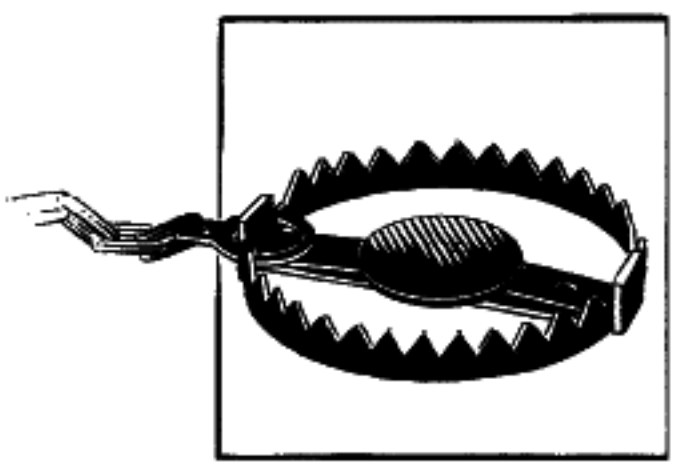
Calling ToString on a DateTimeOffset is the same, except that the offset is returned

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

```
3/02/2000 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 "Saturday, 17 February 2007". `ToShortTimeString` and `ToLongTimeString` return just the time portion, such as 17:10:10 (the former excludes seconds).

These four methods just described 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.



DateTimes and DateTimeOffsets 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 and ParseExact methods do the reverse of ToString, converting a string to a DateTime or DateTimeOffset. The Parse method is also overloaded to accept a format provider.

Null DateTime and DateTimeOffset values

Null DateTime and DateTimeOffset Values

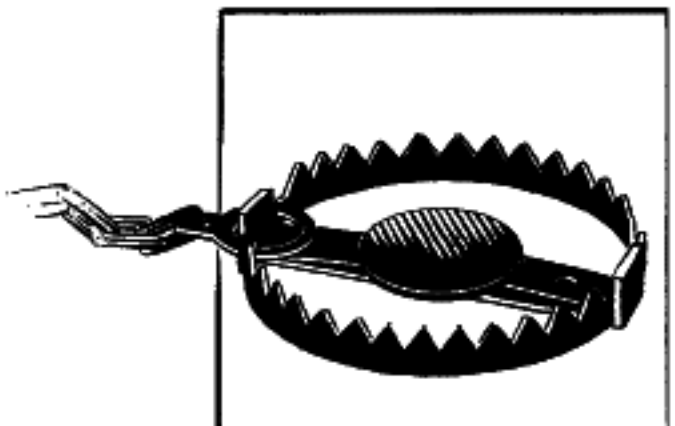
Because `DateTime` and `DateTimeOffset` are structs, they are not intrinsically nullable. When you need nullability, there are two ways around this:

-
-

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 types were introduced).



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!

Dates and Time Zones

In this section, we examine in more detail how time zones influence `DateTime` and `DateTimeOffset`. We also look at the `TimeZone` and `TimeZoneInfo` types, which provide information on time zone offsets and daylight saving time.

DateTime and Time Zones

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 `DateTimeKind`s 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 (2000, 12, 12);  // Unspecified
DateTime utc = DateTime.SpecifyKind (d, DateTimeKind.Utc);
Console.WriteLine (utc);                   // 12/12/2000 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);
Console.WriteLine (utc.Offset);
Console.WriteLine (local == utc);

// -06:00:00 (in Central America)
// 00:00:00
                    // True
```

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

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

TimeZone and TimeZoneInfo

The `TimeZone` and `TimeZoneInfo` classes provide information on time zone names

The `TimeZone` and `TimeZoneInfo` classes provide information on time zone names, UTC offsets, and daylight saving time rules. `TimeZoneInfo` is the more powerful of the two and was new to Framework 3.5.

The biggest difference between the two types is that `TimeZone` lets you access only the current local time zone, whereas `TimeZoneInfo` provides access to all the world's time zones. Further, `TimeZoneInfo` exposes a richer (although at times, more awkward) rules-based model for describing daylight saving time.

TimeZone

The static `TimeZone.CurrentTimeZone` method returns a `TimeZone` object based on the current local settings. The following demonstrates the result if run in Western Australia:

```
TimeZone zone = TimeZone.CurrentTimeZone;
Console.WriteLine (zone.StandardName);
Console.WriteLine (zone.DaylightName);
```

```
Console.WriteLine (zone.DaylightName);

// W. Australia Standard Time
// W. Australia Daylight Time
```

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

```
DateTime dt1 = new DateTime (2008, 1, 1);
DateTime dt2 = new DateTime (2008, 6, 1);
Console.WriteLine (zone.IsDaylightSavingTime (dt1));        // True
Console.WriteLine (zone.IsDaylightSavingTime (dt2));        // False
```



```
Console.WriteLine (zone.GetUtcOffset (dt1));
Console.WriteLine (zone.GetUtcOffset (dt2));

// 09:00:00
// 08:00:00
```

```
// 08:00:00
```

The `GetDaylightChanges` method returns specific daylight saving time information for a given year:

```
DaylightTime day = zone.GetDaylightChanges (2008);
Console.WriteLine (day.Start);   // 26/10/2008 2:00:00 AM  (Note D/M/Y)
Console.WriteLine (day.End);     // 30/03/2008 3:00:00 AM
Console.WriteLine (day.Delta);   // 01:00:00
```

TimeZoneInfo

The `TimeZoneInfo` class works in a similar manner. `TimeZoneInfo.Local` returns the current local time zone:

```
TimeZoneInfo zone = TimeZoneInfo.Local;
Console.WriteLine (zone.StandardName);
Console.WriteLine (zone.DaylightName);
```

```
// W. Australia Standard Time
// W. Australia Daylight Time
```

`TimeZoneInfo` also provides `IsDaylightSavingTime` and `GetUtcOffset` methods—the difference is that they accept either a `DateTime` or a `DateTimeOffset`.

You can obtain a `TimeZoneInfo` for any of the world's time zones by calling `FindSystemTimeZoneById` with the zone ID. This feature is unique to `TimeZoneInfo`, as is everything else that we demonstrate from this point on. We'll stick with 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
```

```
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);
```


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:

tional 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 `TimeSpan`s representing the valid offset choices for an ambiguous `DateTime` or `DateTimeOffset`.

Unlike with `TimeZone`, you can't obtain simple dates from a `DateZoneInfo` 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 +
```

```
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

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 ("Rule: applies from " + rule.DateStart +
                                        " to " + rule.DateEnd);
```

```
    Console.WriteLine ("
    Console.WriteLine ("
    Console.WriteLine ("
    Console.WriteLine();
}
```

```
Delta: " + rule.DaylightDelta);
Start: " + FormatTransitionTime
          (rule.DaylightTransitionStart, false));
End:   " + FormatTransitionTime
          (rule.DaylightTransitionEnd, true));
```

```
(rule.DaylightTransitionEnd, true));
```

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;
```

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


The result with Western Australia is interesting in that it demonstrates both fixed and floating date rules—as well as an absent end date:

```
Rule: applies from 1/01/2006 12:00:00 AM to 31/12/2006 12:00:00 AM
   Delta: 01:00:00
   Start: 3 December at 02:00:00
```

```
Delta: 01:00:00
Start: 3 December at 02:00:00
End:   -
```

```
Rule: applies from 1/01/2007 12:00:00 AM to 31/12/2009 12:00:00 AM
   Delta: 01:00:00
   Start: The last Sunday in October at 02:00:00
   End:   The last Sunday in March at 03:00:00
```


Western Australia is actually unique in this regard. Here's how we found it:

```
from zone in TimeZoneInfo.GetSystemTimeZones()
let rules = zone.GetAdjustmentRules()
where
  rules.Any
   (r => r.DaylightTransitionEnd.IsFixedDateRule) &&
  rules.Any
   (r => !r.DaylightTransitionEnd.IsFixedDateRule)
select zone
```

```
select zone
```

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 `DateTimes`, daylight saving can be problematic.

The rules can be summarized as follows:

-
-
-

Daylight saving impacts 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 `DateTimes`.

You can always reliably round-trip between UTC and local times (on the same computer)—even as the clocks turn back.

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. When the clocks go back, the same hour (or more precisely, `Delta`) repeats itself. We can demonstrate this by instantiating a `DateTime` right in the "twilight zone" on your computer, and then subtracting `Delta` (this example requires that you practice daylight saving time to be interesting!):

```
DaylightTime changes = TimeZone.CurrentTimeZone.GetDaylightChanges (2010);
TimeSpan halfDelta = new TimeSpan (changes.Delta.Ticks / 2);
DateTime utc1 = changes.End.ToUniversalTime() - halfDelta;
DateTime utc2 = utc1 - changes.Delta;
```

Converting these variables to local times demonstrates why you should use UTC and not local time if your code relies on time moving forward:

```
DateTime loc1 = utc1.ToLocalTime();  // (Pacific Standard Time)
DateTime loc2 = utc2.ToLocalTime();
Console.WriteLine (loc1);            // 2/11/2010 1:30:00 AM
Console.WriteLine (loc2);            // 2/11/2010 1:30:00 AM
Console.WriteLine (loc1 == loc2);    // True
```


Despite `loc1` and `loc2` reporting as equal, they are different inside. `DateTime` reserves a special bit for indicating on which side of the twilight zone an ambiguous local date lies! This bit is ignored in comparison—as we just saw—but comes into play when you format the `DateTime` unambiguously:

```
Console.Write (loc1.ToString ("o"));  // 2010-11-02T02:30:00.0000000-08:00
Console.Write (loc2.ToString ("o"));  // 2010-11-02T02:30:00.0000000-07:00
```

This bit also is read when you convert back to UTC, ensuring perfect round-tripping between local and UTC times:

```
Console.WriteLine (loc1.ToUniversalTime() == utc1);
```

```
Console.WriteLine (loc1.ToUniversalTime() == utc1);
Console.WriteLine (loc2.ToUniversalTime() == utc2);

// True
// True
```

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, the .NET Framework provides a variety of mechanisms:

The need to format or parse arises frequently in programming, in a variety of situations. Hence, the .NET Framework 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. The .NET Framework 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. In the section following, we describe `XmlConvert` and type converters, as well as 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. For example:

```
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:

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

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:

flying 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);
```

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:

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
```

```
// $$ 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.

All format strings for numbers and dates are listed in "Standard Format Strings and Parsing Flags" on page 225.

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.

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

- `NumberFormatInfo`
- `DateTimeFormatInfo`
- `CultureInfo`

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 Great *B*ritain):

Britain):

```
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
```

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

- The currency symbol is ¤ instead of $.

- 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();
```

```
                    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 and Text Handling" on page 193:

```
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` (`AppendFormat`), and to a `TextWriter` for I/O (see Chapter 14).

(via

(via

`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. For example:

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

This is equivalent to:

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

```
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 work: 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 of these 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"));
```

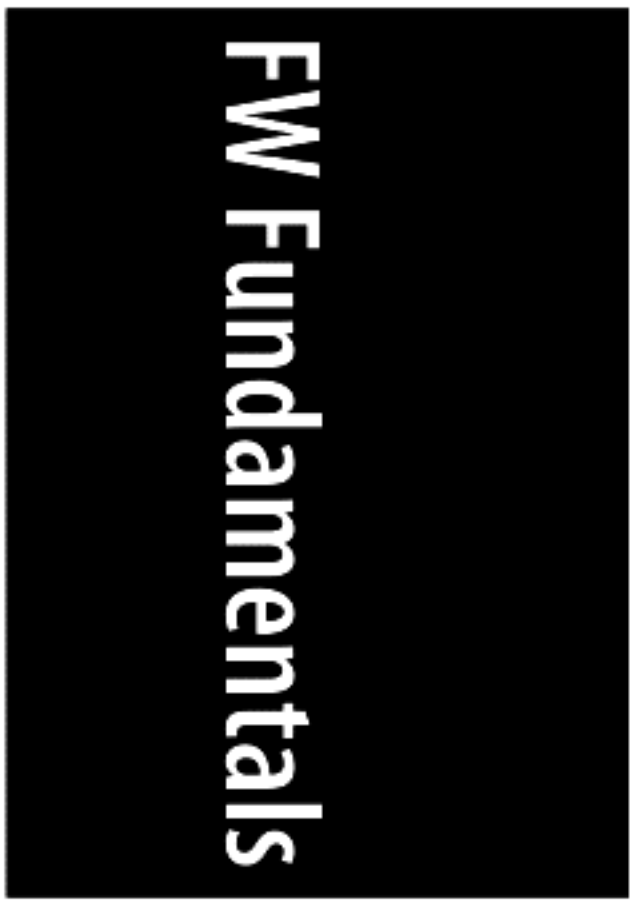
The next section lists all `NumberStyles` and `DateTimeStyles` members—as well as the

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 `DateTimeInfo` 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:

```
// Program can be downloaded from http://www.albahari.com/nutshell/

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

```
  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 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);
      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` to convert the input number to a string—with `InvariantCulture`. It would have been much simpler just to call `ToString()` on `arg`, but then `CurrentCulture` would have been used instead. The

to a string—with InvariantCulture. It would have been much 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);
```

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 one two three point four five
```

Custom format providers can be used 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 lists all standard numeric format strings.

Table 6-2. Standard numeric format strings

Letter	Meaning	Sample input	Result	Notes

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`
				Switches to exponential notation for small or large numbers.
				G3 limits precision to three digits in *total* (before + after point).

F	Fixed point	`2345.678, "F2"` `2345.6, "F2"`	`2345.68` `2345.60`	F2 rounds to two decimal places.
N	Fixed point with *group separator* ("Numeric")	`2345.678, "N2"` `2345.6, "N2"`	`2,345.68` `2,345.60`	As above, with group (1,000s) separator (details from format provider).
D	Pad with leading zeros	`123, "D5"` `123, "D1"`	`00123` `123`	For integral types only. D5 pads left to five digits; does not truncate.
E or e	Force exponential notation	`56789, "E"` `56789, "e"`	`5.678900E+004` `5.678900e+004`	Six-digit default precision.

E or e	Force exponential notation	56789, "E"	5.678900E+004	Six-digit default precision.
		56789, "e"	5.678900e+004	
		56789, "E2"	5.68E+004	

Letter	Meaning	Sample input	Result	Notes
C	Currency	1.23, "C"	$1.23	C with no digit uses default number

Letter	Meaning	Sample input	Result	Notes
C	Currency	`1.2, "C"` `1.2, "C4"`	`$1.20` `$1.2000`	C with no digit uses default number of D.P. from format provider.
P	Percent	`.503, "P"` `.503, "P0"`	`50.30 %` `50 %`	Uses symbol and layout from format provider. Decimal places can optionally be overridden.

X or x	Hexadecimal	`47, "X"`
		`47, "x"`
		`47, "X4"`
			`2F`
			`2f`
			`002F`
				X for uppercase hex digits; x for lowercase hex digits.
				Integrals only.

R	Round-trip	`1f / 3f, "R"`	`0.33333343`	For the `float` and `double` types, R squeezes out all digits to ensure exact round-tripping.

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.

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"` format string exists 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	Notes

Specifier	Meaning	Sample input	Result	Notes
#	Digit placeholder	`12.345, ".##"` `12.345, ".####"`	`12.35` `12.345`	Limits digits after D.P.

0	Zero placeholder	`12.345, ".00"`
		`12.345, ".0000"`
		`99, "000.00"`
			`12.35`
			`12.3500`
			`099.00`
				As above, but also pads with zeros before and after D.P.

.	Decimal point			Indicates D.P.

Specifier	Meaning	Sample input	Result	Notes
				Actual symbol comes from `NumberFormatInfo`.
,	Group separator	`1234, "#,###,###"` `1234, "0,000,000"`	`1,234` `0,001,234`	Symbol comes from `NumberFormatInfo`.
, (as above)	Multiplier	`1000000, "#,"` `1000000, "#,,`	`1000` `1`	If comma is at end or before D.P., it acts as a multiplier—dividing result by 1,000, 1,000,000, etc.

(illegible)		(illegible)	(illegible)	dividing result by 1,000, 1,000,000, etc.
	Percent notation	`0.6, "00%"`	60%	First multiplies by 100 and then substitutes percent symbol obtained from `NumberFormatInfo`.

`E0, e0, E+0, e+0 E-0, e-0`	Exponent notation	`1234, "0E0"`
		`1234, "0E+0"`
		`1234, "0.00E00"`
		`1234, "0.00e00"`
			`1E0`
			`1E+3`
			`1.25E03`
			`1.25e03`	

`\`	Literal character quote	`50, @"\#0"`	`#50`	Use in conjunction with an @ prefix on the string—or use `\\`.
`'xx''xx'`	Literal string quote	`50, "0 '...'"`	`50 ...`	
`;`	Section separator	`15, "#;(#);zero"`	`15`	(If positive).
		`-5, "#;(#);zero"`	`-5`	(If negative).
		`0, "#;(#);zero"`	`zero`	(If zero).
Any other char	Literal	`35.2, "$0 . 00c"`	`$35 . 00c`	