Any other char	Literal	35.2, $0 . 00c	$35 . 00c	

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:

FW Fundamentals

```
AllowLeadingWhite
AllowLeadingSign
AllowParentheses
AllowThousands
AllowCurrencySymbol
```

```
AllowTrailingWhite
AllowTrailingSign
AllowDecimalPoint
AllowExponent
AllowHexSpecifier
```

`AllowHexSpecifier`

`NumberStyles` also defines these composite members:

`None`

`Integer`

`Float`

`Number`

`HexNumber`

`Currency`

`Any`

Any

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

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

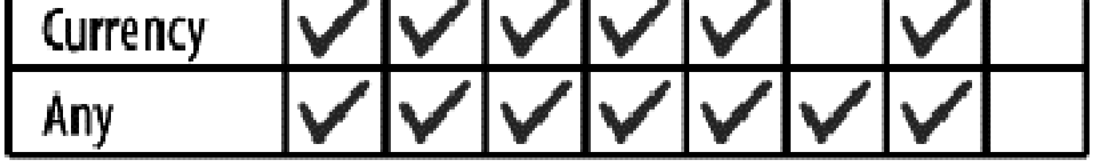
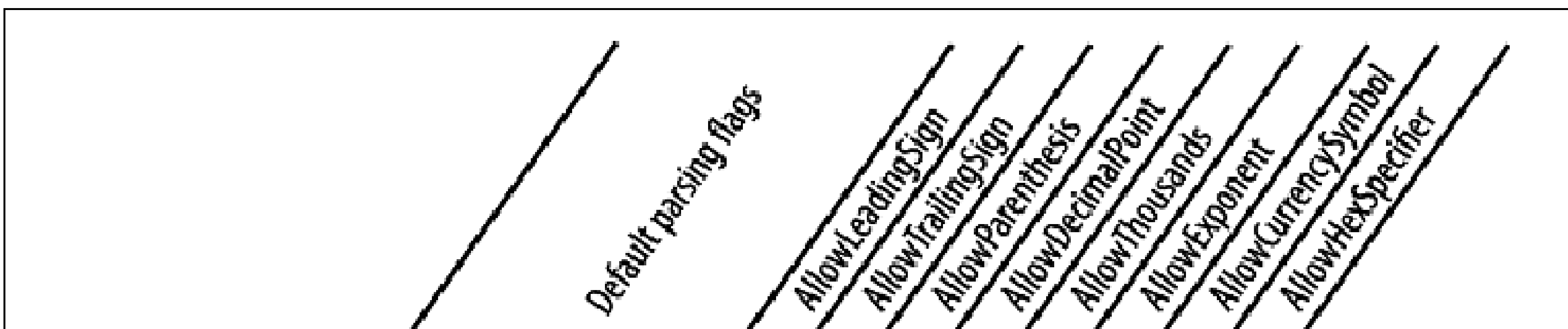


Figure 6-1. Composite NumberStyles

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

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);
```



	Defau	Allow	Allow	Allow	Allow	Allow	Allow	Allow	Allow
Integral types	`Integer`	✓							
`double` and `float`	`Float | AllowThousands`	✓			✓	✓	✓		
`decimal`	`Number`	✓	✓		✓	✓			

Figure 6-2. Default parsing flags for numeric types

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);
```

```
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. Those that do are listed in Table 6-4; those that don't are listed in Table 6-5. 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 + 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	January 02
y, Y	Year and month	2000 January

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

FW Fundamentals

s

u

Sortable; ISO 8601

"Universal" Sortable

Universal Sortable

2000-01-02T17:18:19

2000-01-02 17:18:19Z

Compatible with text-based sorting

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

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 fairly exhaustive and you can find it in the MSDN. An example of a custom format string is:

```
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 or any of your customers live outside the United States and Canada. 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 `DateTimeOffsets` 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 harder 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 we generate a culture-insensitive `DateTime` string `s` as follows:

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

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
```

```
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.

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 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).

space 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" on page 240 in Chapter 3, we describe 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

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

FW Fundamentals

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

— 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

The .NET Framework calls the following types *base types*:

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

All of 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.

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

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);
uint five  = Convert.ToUInt32 ("101", 2);
```

```
// Parse in hexadecimal
// 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` 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);
Console.WriteLine (result.GetType());

// 42
// 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" on page 240).

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

Base 64 conversions

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 `FromBase64String` does the reverse.

converts

from

a

byte

array

to base

64;

XmlConvert

If you're dealing with data that's originated from or destined for an XML file, `XmlConvert` (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 Framework 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`

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

```
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.

`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 XAML (Extensible Application Markup Language) documents—as used in Windows Presentation Foundation and Workflow Foundation.

In the .NET Framework, 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 an ASP.NET application in Visual Studio, if you assign a control a `BackColor` by typing **`"Beige"`** into the 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.

ers 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, *System.Drawing.dll*):

```
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

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 for converting in the other direction, such as `ToDouble`.

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

FW Fundamentals

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

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

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. This can be done *after* writing your program—we cover the details in "Resources and Satellite Assemblies" on page 663 in Chapter 17.

The .NET Framework 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`.

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" on page 203).

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`.

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, and with a period separator.

Dates are formatted as day/month/year, and with a 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.

> `Thread` and `CultureInfo` also support a `CurrentUICulture` property. This is concerned more about localization: we cover this in Chapter 17.

Working with Numbers

Conversions

We covered numeric conversions in previous chapters and sections; Table 6-7 summarizes all 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.To`*`Integral`*	`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 = d;`
Truncating numeric conversion	Explicit cast	`double d = 23.5;` `int i = (int) d;`
Rounding numeric conversion	`Convert.To`*`Integral`*	`double d = 23.5;`

		`int i = (int) d;`
Rounding numeric conversion (real to integral)	`Convert.To`*`Integral`*	`double d = 23.5;` `int i = Convert.ToInt32 (d);`

Math

Table 6-8 lists the 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`

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 new to .NET Framework 4.0. It lives in the new `System.Numerics` namespace in *System.Numerics.dll* and allows you to represent an arbitrarily large integer without any loss of precision.

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 cast from any other integral type to a **BigInteger**. For instance:

```
BigInteger twentyFive = 25;      // implicit cast 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 (100, '0'));
```

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

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

You can implicitly cast a **BigInteger** to a standard numeric type and explicitly cast in the other direction. For instance:

in the other direction. For instance:

```
double g1 = 1e100;
BigInteger g2 = (BigInteger) g1;
Console.WriteLine (g2);

// implicit cast
// explicit cast
```

You lose precision when converting a large `BigInteger` to another type, such as `double`. The output of the preceding example demonstrates this:

```
10000000000000000159028911097599180468360808563945281389781327557747...
```

`BigInteger` overloads all the arithmetic operators, as well as the comparison, equality, modulus (`%`), and negation 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

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.

Complex

The Complex struct is another specialized numeric type new to Framework 4.0, and is for representing complex numbers with real and imaginary components of type double. Complex resides in the *System.Numerics.dll* assembly (along with BigInteger).

`double`. `Complex` resides in the *System.Numerics.dll* assembly (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)
```

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

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

- Trigonometric (`Sin`, `Asin`, `Sinh`, `Tan`, etc.)
- Logarithms and exponentiations
- `Conjugate`

Random

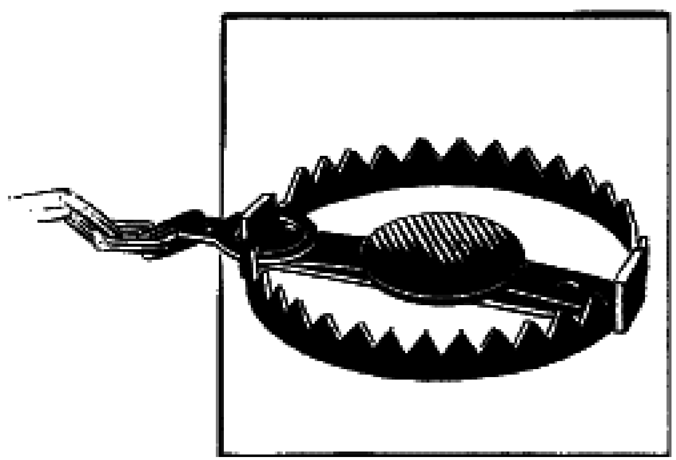
FW F

The Random class generates a pseudorandom sequence of random bytes, integers, or doubles.

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, 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—then it uses the current system time to make one up.



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

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" on page 862 in Chapter 21.

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.

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

```
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);
```

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

Enums

In Chapter 3, we described C#'s enum type, and showed how to combine members, test equality, use logical operators, and perform conversions. The Framework extends C#'s support for enums through the `System.Enum` type. This type has two roles: providing type unification for all `enum` types and defining static utility methods.

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

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

static void Main()
{
```

```
{
  Display (Nut.Macadamia);
  Display (Size.Large);
}
```

```
// Nut.Macadamia
// Size.Large
```



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

The static utility methods on System.Enum are primarily related to performing con-

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:

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:

FW Fu

```
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
```

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" on page 243.

The third approach is to call `Format` or `ToString` specifying the `"d"` or `"D"` format

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 this:

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

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

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" on page 225.

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 FormatException is thrown if the member is not found.

An optional third argument lets you perform case-insensitive parsing. A FormatException 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*.

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 `enums` 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.

constants.

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

```
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.

Tuples

Tuples

Framework 4.0 provides a new set of generic classes for holding a set of differently typed elements. These are called *tuples*:

```
public class Tuple <T1>
public class Tuple <T1, T2>
public class Tuple <T1, T2, T3>
public class Tuple <T1, T2, T3, T4>
public class Tuple <T1, T2, T3, T4, T5>
public class Tuple <T1, T2, T3, T4, T5, T6>
public class Tuple <T1, T2, T3, T4, T5, T6, T7>
public class Tuple <T1, T2, T3, T4, T5, T6, T7, TRest>
```

Each has read-only properties called `Item1`, `Item2`, and so on (one for each type parameter).

You can instantiate a tuple either via its constructor:

```
var t = new Tuple<int,string> (123, "Hello");
```

```
var t = new Tuple<int,string> (123, "Hello");
```

or via the static helper method `Tuple.Create`:

```
Tuple<int,string> t = Tuple.Create (123, "Hello");
```

The latter leverages generic type inference. You can combine this with implicit typing:

```
var t = Tuple.Create (123, "Hello");
```

You can then access the properties as follows (notice that each is statically typed):

```
Console.WriteLine (t.Item1 * 2);         // 246
Console.WriteLine (t.Item2.ToUpper());   // HELLO
```

Tuples are convenient in returning more than one value from a method—or creating collections of value *pairs* (we'll cover collections in the following chapter).

An alternative to tuples is to use an object array. However, you then lose static type safety, incur the cost of boxing/unboxing for value types, and require clumsy casts that cannot be validated by the compiler:

```
object[] items = { 123, "Hello" };
Console.WriteLine ( ((int)    items[0]) * 2        );   // 246
```

```
object[] items = { 123, "Hello" };
Console.WriteLine ( ((int)    items[0]) * 2       );   // 246
Console.WriteLine ( ((string) items[1]).ToUpper() );   // HELLO
```

Comparing Tuples

Tuples are classes (and therefore reference types). In keeping with this, comparing two distinct instances with the equality operator returns `false`. However, the `Equals` method is overridden to compare each individual element instead:

```
var t1 = Tuple.Create (123, "Hello");
var t2 = Tuple.Create (123, "Hello");
Console.WriteLine (t1 == t2);           // False
Console.WriteLine (t1.Equals (t2));     // True
```


You can also pass in a custom equality comparer (by virtue of tuples implementing `IStructuralEquatable`). We cover equality and order comparison later in this chapter.

The Guid Struct

The Guid Struct

The `Guid` struct represents a globally unique identifier: a 16-byte value that, if randomly generated, will almost certainly be 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.

To create a new random `Guid`, call the static `Guid.NewGuid` method:

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

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

```
public Guid (byte[] b);
public Guid (string g);
```

```
// Accepts a 16-byte array
```

```
// Accepts a 16-byte array
// 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

Equality Comparison

FW Fundamentals

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 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 must first look at the preliminary concept of value versus referential equality.

Value Versus Referential Equality

There are two kinds of equality:

Value equality

Value equality

Two values are *equivalent* in some sense.

Referential equality

Two references refer to *exactly the same object*.

By default:

- Value types use *value equality*.
- Reference types use *referential equality*.

Value types, in fact, can *only* use 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)
```

```
// 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
```

`DateTimeOffset` is a struct whose equality semantics have been tweaked. By default, structs exhibit a special kind of value equality called *structural equality*, where 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; }
```

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
```

We'll explain later in this section how reference types can be *customized* 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 uri1 = new Uri ("http://www.linqpad.net");
Uri uri2 = new Uri ("http://www.linqpad.net");
Console.WriteLine (uri1 == uri2);                   // True
```

Standard Equality Protocols

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

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

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*, and so 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 hard-wires `==` to the `int` type because `x` and `y` are both `int`:

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

```
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.

FW Fundam

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 so 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.

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)
{
  return 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);
}
```

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. For example:

```
object x = 3, y = 3;
```

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

```
// True
// False
// 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>
```

```
class Test <T>
{
  T _value;
  public void SetValue (T newValue)
```



```
  {
    if (!object.Equals (newValue, _value))
    {
      _value = newValue;
      OnValueChanged();
    }
  }
  protected virtual void OnValueChanged() { ... }
}
```

```
  protected virtual void OnValueChanged() { ... }
}
```

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

> 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" on page 304).

The static object.ReferenceEquals method

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

```
class Widget { ... }
```

`object.ReferenceEquals` method does just this:

```
class Widget { ... }

class Test
{
  static void Main()
  {
    Widget w1 = new Widget();
    Widget w2 = new Widget();
    Console.WriteLine (object.ReferenceEquals (w1, w2));
  }
}

// False
```

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.

cases, calling `object.ReferenceEquals` guarantees normal referential equality semantics.

FW Fundamentals

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 imple-

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 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:

nitions of equality. For example:

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

```
// False
// 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.

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, and happens when the author customizes `Equals` so that it performs value equality while leaving == to perform (default) referential equality. The `StringBuilder` class does exactly this:

```
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*.

- Value types use *value equality*.
- Reference types use *referential equality*.

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

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`).

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 unof five. Overloading the `==` operator and implementing `IEquatable<T>` allows unboxed equality comparisons, and this can speed things up by a factor of five again.

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.

32- or 64-bit references.

There's actually 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 of the fact that equality comparison and hashing are joined at the hip. We'll examine hashing in a moment.

How to override equality semantics

Here is a summary of the steps:

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

Overriding GetHashCode

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 where 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.

We describe hashtables

We describe hashtables in detail in "Dictionaries" on page 292 in Chapter 7.

Both reference and value types have default implementations of `GetHashCode`, meaning you don't need to override this method—*unless you override* `Equals`. The converse is also true: override `GetHashCode` and you must also override `Equals`.

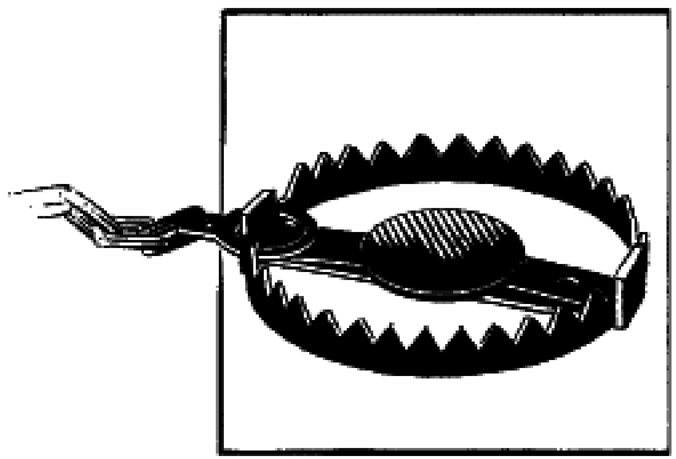
Here are the other rules for overriding `object.GetHashCode`:

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, `GetHashCode` should be written so as to minimize the likelihood of two different values returning the same hash code. 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 for structs simply performs a bitwise exclusive OR on each of the fields, which typically generates more duplicate codes than if you wrote the algorithm yourself.

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.



If an object's hash code 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 hash code calculations on immutable fields.

A complete example illustrating how to override `GetHashCode` is listed shortly.

Overriding Equals

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 !=

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)
```

```
Console.WriteLine (sb1 == sb2);          // False (referential equality)
Console.WriteLine (sb1.Equals (sb2));    // True  (value equality)
```

FW Fundamentals

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 `structs`.

Although it's possible to overload `!=` such that it means some-

Although it's possible to overload `!=` such that it means something other than `!(==)`, this is almost never done in practice, except in cases such as comparing `float.NaN`.

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 following example.

An example: The Area struct

Imagine 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.)

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)
{
  if (!(other is Area)) return false;
  return Equals ((Area) other);        // Calls method below
}

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

public override int GetHashCode()
{
  return Measure2 * 31 + Measure1;
}
```

```
// 31 = some prime number

  public static bool operator == (Area a1, Area a2)
  {
    return a1.Equals (a2);
  }
  public static bool operator != (Area a1, Area a2)
  {
    return !a1.Equals (a2);
  }
}
```


Here's another way to implement the Equals method, leveraging nullable types:

nullable types:

```
Area? otherArea = other as Area?;
return otherArea.HasValue && Equals (otherArea.Value);
```

In implementing `GetHashCode`, we've helped to improve the likelihood of uniqueness by multiplying the larger measure by some prime number (ignoring any overflow) before adding the two together. When there are more than two fields, the following pattern, suggested by Josh Bloch, gives good results while being performant:

```
int hash = 17;   // 17 = some prime number
hash = hash * 31 + field1.GetHashCode();   // 31 = another prime number
hash = hash * 31 + field2.GetHashCode();
hash = hash * 31 + field3.GetHashCode();
...
return hash;
```

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

```
Area a1 = new Area (5, 10);
Area a2 = new Area (10, 5);
```

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

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

Pluggable equality comparers

If you want a type to take on different equality semantics just for a particular 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" on page 304.

Order Comparison