Chapter 4. Advanced C#

In this chapter, we cover advanced C# topics that build on concepts explored in Chapters 2 and 3. You should read the first four sections sequentially; you can read the remaining sections in any order.

Delegates

A delegate is an object that knows how to call a method.

A *delegate type* defines the kind of method that *delegate instances* can call. Specifically, it defines the method's *return type* and its *parameter types*. The following defines a delegate type called `Transformer`:

```
delegate int Transformer (int x);
```

`Transformer` is compatible with any method with an `int` return type and a single `int` parameter, such as this:

```
static int Square (int x) { return x * x; }
```

Or, more tersely:

```
static int Square (int x) => x * x;
```

Assigning a method to a delegate variable creates a delegate *instance*:

```
Transformer t = Square;
```

You can invoke a delegate instance in the same way as a method:

```
int answer = t(3);    // answer is 9
```

Here's a complete example:

```
delegate int Transformer (int x);

class Test
{
  static void Main()
  {
    Transformer t = Square;          // Create delegate instance
    int result = t(3);               // Invoke delegate
    Console.WriteLine (result);      // 9
  }
  static int Square (int x) => x * x;
}
```

A delegate instance literally acts as a delegate for the caller: the caller invokes the delegate and then the delegate calls the target method. This indirection decouples the caller from the target method.

The statement:

```
Transformer t = Square;
```

is shorthand for:

```
Transformer t = new Transformer (Square);
```

> **NOTE**
>
> Technically, we are specifying a *method group* when we refer to `Square` without brackets or arguments. If the method is overloaded, C# will pick the correct overload based on the signature of the delegate to which it's being assigned.

The expression:

```
t(3)
```

is shorthand for:

```
t.Invoke(3)
```

> **NOTE**
>
> A delegate is similar to a *callback*, a general term that captures constructs such as C function pointers.

Writing Plug-in Methods with Delegates

A delegate variable is assigned a method at runtime. This is useful for writing plug-in methods. In this example, we have a utility method

named `Transform` that applies a transform to each element in an integer array. The `Transform` method has a delegate parameter, which you can use for specifying a plug-in transform:

```
public delegate int Transformer (int x);

class Util
{
  public static void Transform (int[] values, Transformer t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t (values[i]);
  }
}

class Test
{
  static void Main()
  {
    int[] values = { 1, 2, 3 };
    Util.Transform (values, Square);      // Hook in the Square method
    foreach (int i in values)
      Console.Write (i + "  ");           // 1   4   9
  }

  static int Square (int x) => x * x;
}
```

Our `Transform` method is a *higher-order function* because it's a function that takes a function as an argument. (A method that *returns* a delegate would also be a higher-order function.)

Multicast Delegates

All delegate instances have *multicast* capability. This means that a delegate instance can reference not just a single target method, but also a list of target methods. The + and += operators combine delegate instances:

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

The last line is functionally the same as the following:

```
d = d + SomeMethod2;
```

Invoking `d` will now call both `SomeMethod1` and `SomeMethod2`. Delegates are invoked in the order in which they are added.

The `-` and `-=` operators remove the right delegate operand from the left delegate operand:

```
d -= SomeMethod1;
```

Invoking `d` will now cause only `SomeMethod2` to be invoked.

Calling + or += on a delegate variable with a `null` value works, and it is equivalent to assigning the variable to a new value:

```
SomeDelegate d = null;
d += SomeMethod1;       // Equivalent (when d is null) to d =
SomeMethod1;
```

Similarly, calling `-=` on a delegate variable with a single matching target is equivalent to assigning `null` to that variable.

> **NOTE**
>
> Delegates are *immutable*, so when you call `+=` or `-=`, you're in fact creating a *new* delegate instance and assigning it to the existing variable.

If a multicast delegate has a nonvoid return type, the caller receives the return value from the last method to be invoked. The preceding methods are still called, but their return values are discarded. For most scenarios in which multicast delegates are used, they have `void` return types, so this subtlety does not arise.

> **NOTE**
>
> All delegate types implicitly derive from `System.MulticastDelegate`, which inherits from `System.Delegate`. C# compiles `+`, `-`, `+=`, and `-=` operations made on a delegate to the static `Combine` and `Remove` methods of the `System.Delegate` class.

MULTICAST DELEGATE EXAMPLE

Suppose that you wrote a method that took a long time to execute. That method could regularly report progress to its caller by invoking a delegate. In this example, the `HardWork` method has a `ProgressReporter` delegate parameter, which it invokes to indicate progress:

```
public delegate void ProgressReporter (int percentComplete);

public class Util
{
  public static void HardWork (ProgressReporter p)
  {
    for (int i = 0; i < 10; i++)
    {
      p (i * 10);                           // Invoke delegate
      System.Threading.Thread.Sleep (100);  // Simulate hard work
    }
  }
}
```

To monitor progress, the `Main` method creates a multicast delegate instance `p`, such that progress is monitored by two independent methods:

```
class Test
{
  static void Main()
  {
    ProgressReporter p = WriteProgressToConsole;
    p += WriteProgressToFile;
    Util.HardWork (p);
  }

  static void WriteProgressToConsole (int percentComplete)
    => Console.WriteLine (percentComplete);

  static void WriteProgressToFile (int percentComplete)
    => System.IO.File.WriteAllText ("progress.txt",
                                    percentComplete.ToString());
}
```

Instance Versus Static Method Targets

When an *instance* method is assigned to a delegate object, the latter must maintain a reference not only to the method, but also to the *instance* to which the method belongs. The `System.Delegate` class's `Target` property represents this instance (and will be null for a delegate referencing a static method). Here's an example:

```
public delegate void ProgressReporter (int percentComplete);

class Test
{
  static void Main()
  {
    X x = new X();
    ProgressReporter p = x.InstanceProgress;
    p(99);                                  // 99
    Console.WriteLine (p.Target == x);      // True
    Console.WriteLine (p.Method);           // Void
InstanceProgress(Int32)
  }
}

class X
{
  public void InstanceProgress (int percentComplete)
    => Console.WriteLine (percentComplete);
}
```

Generic Delegate Types

A delegate type can contain generic type parameters:

```
public delegate T Transformer<T> (T arg);
```

With this definition, we can write a generalized `Transform` utility method that works on any type:

```
public class Util
{
  public static void Transform<T> (T[] values, Transformer<T> t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t (values[i]);
  }
}

class Test
{
  static void Main()
  {
    int[] values = { 1, 2, 3 };
    Util.Transform (values, Square);      // Hook in Square
    foreach (int i in values)
      Console.Write (i + "  ");           // 1   4   9
  }

  static int Square (int x) => x * x;
}
```

The Func and Action Delegates

With generic delegates, it becomes possible to write a small set of delegate types that are so general they can work for methods of any return type and any (reasonable) number of arguments. These delegates are the `Func` and `Action` delegates, defined in the `System` namespace (the `in` and `out` annotations indicate *variance*, which we cover in the context of delegates shortly):

```
delegate TResult Func <out TResult>                ();
delegate TResult Func <in T, out TResult>          (T arg);
delegate TResult Func <in T1, in T2, out TResult>  (T1 arg1, T2 arg2);
... and so on, up to T16

delegate void Action                 ();
delegate void Action <in T>          (T arg);
delegate void Action <in T1, in T2>  (T1 arg1, T2 arg2);
... and so on, up to T16
```

These delegates are extremely general. The `Transformer` delegate in our previous example can be replaced with a `Func` delegate that takes a single argument of type `T` and returns a same-typed value:

```
public static void Transform<T> (T[] values, Func<T,T> transformer)
{
  for (int i = 0; i < values.Length; i++)
    values[i] = transformer (values[i]);
}
```

The only practical scenarios not covered by these delegates are `ref`/`out` and pointer parameters.

> **NOTE**
>
> Prior to Framework 2.0, the `Func` and `Action` delegates did not exist (because generics did not exist). It's for this historical reason that much of the Framework uses custom delegate types rather than `Func` and `Action`.

Delegates Versus Interfaces

A problem that you can solve with a delegate can also be solved with an interface. For instance, we can rewrite our original example with an interface called `ITransformer` instead of a delegate:

```
public interface ITransformer
{
  int Transform (int x);
}

public class Util
{
 public static void TransformAll (int[] values, ITransformer t)
 {
   for (int i = 0; i < values.Length; i++)
     values[i] = t.Transform (values[i]);
 }
}

class Squarer : ITransformer
{
  public int Transform (int x) => x * x;
}
...

static void Main()
{
  int[] values = { 1, 2, 3 };
  Util.TransformAll (values, new Squarer());
  foreach (int i in values)
    Console.WriteLine (i);
}
```

A delegate design might be a better choice than an interface design if one or more of these conditions are true:

- The interface defines only a single method.
- Multicast capability is needed.
- The subscriber needs to implement the interface multiple times.

In the `ITransformer` example, we don't need to multicast. However, the interface defines only a single method. Furthermore, our subscriber might need to implement `ITransformer` multiple times, to support different transforms, such as square or cube. With interfaces, we're forced into writing a separate type per transform because `Test` can implement `ITransformer` only once. This is quite cumbersome:

```
class Squarer : ITransformer
{
  public int Transform (int x) => x * x;
}

class Cuber : ITransformer
{
  public int Transform (int x) => x * x * x;
}
...

static void Main()
{
  int[] values = { 1, 2, 3 };
  Util.TransformAll (values, new Cuber());
  foreach (int i in values)
    Console.WriteLine (i);
}
```

Delegate Compatibility

TYPE COMPATIBILITY

Delegate types are all incompatible with one another, even if their signatures are the same:

```
delegate void D1();
delegate void D2();
...

D1 d1 = Method1;
D2 d2 = d1;                             // Compile-time error
```

> **NOTE**
>
> The following, however, is permitted:
>
> ```
> D2 d2 = new D2 (d1);
> ```

Delegate instances are considered equal if they have the same method targets:

```
delegate void D();
...

D d1 = Method1;
D d2 = Method1;
Console.WriteLine (d1 == d2);         // True
```

Multicast delegates are considered equal if they reference the same methods *in the same order*.

PARAMETER COMPATIBILITY

When you call a method, you can supply arguments that have more specific types than the parameters of that method. This is ordinary polymorphic behavior. For the same reason, a delegate can have more specific parameter types than its method target. This is called *contravariance*. Here's an example:

```
delegate void StringAction (string s);

class Test
{
  static void Main()
  {
    StringAction sa = new StringAction (ActOnObject);
    sa ("hello");
  }

  static void ActOnObject (object o) => Console.WriteLine (o);   //
hello
}
```

(As with type parameter variance, delegates are variant only for *reference conversions*.)

A delegate merely calls a method on someone else's behalf. In this case, the `StringAction` is invoked with an argument of type `string`. When the argument is then relayed to the target method, the argument is implicitly upcast to an `object`.

> **NOTE**
>
> The standard event pattern is designed to help you utilize contravariance through its use of the common `EventArgs` base class. For example, you can have a single method invoked by two different delegates, one passing a `MouseEventArgs` and the other passing a `KeyEventArgs`.

RETURN TYPE COMPATIBILITY

If you call a method, you might get back a type that is more specific than what you asked for. This is ordinary polymorphic behavior. For the same reason, a delegate's target method might return a more specific type than described by the delegate. This is called *covariance*:

```
delegate object ObjectRetriever();

class Test
{
  static void Main()
  {
    ObjectRetriever o = new ObjectRetriever (RetrieveString);
    object result = o();
    Console.WriteLine (result);      // hello
  }
  static string RetrieveString() => "hello";
}
```

`ObjectRetriever` expects to get back an `object`, but an `object` *subclass* will also do: delegate return types are *covariant*.

GENERIC DELEGATE TYPE PARAMETER VARIANCE

In Chapter 3, we saw how generic interfaces support covariant and contravariant type parameters. The same capability exists for delegates, too.

If you're defining a generic delegate type, it's good practice to do the following:

- Mark a type parameter used only on the return value as covariant (`out`).
- Mark any type parameters used only on parameters as contravariant (`in`).

Doing so allows conversions to work naturally by respecting inheritance relationships between types.

The following delegate (defined in the `System` namespace) has a covariant `TResult`:

```
delegate TResult Func<out TResult>();
```

allowing:

```
Func<string> x = ...;
Func<object> y = x;
```

The following delegate (defined in the `System` namespace) has a contravariant `T`:

```
delegate void Action<in T> (T arg);
```

allowing:

```
Action<object> x = ...;
Action<string> y = x;
```

Events

When using delegates, two emergent roles commonly appear: *broadcaster* and *subscriber*.

The *broadcaster* is a type that contains a delegate field. The broadcaster decides when to broadcast, by invoking the delegate.

The *subscribers* are the method target recipients. A subscriber decides when to start and stop listening by calling `+=` and `-=` on the broadcaster's delegate. A subscriber does not know about, or interfere with, other subscribers.

Events are a language feature that formalizes this pattern. An `event` is a construct that exposes just the subset of delegate features required for the broadcaster/subscriber model. The main purpose of events is to *prevent subscribers from interfering with one another*.

The easiest way to declare an event is to put the `event` keyword in front of a delegate member:

```
// Delegate definition
public delegate void PriceChangedHandler (decimal oldPrice,
                                          decimal newPrice);
public class Broadcaster
```

```
{
  // Event declaration
  public event PriceChangedHandler PriceChanged;
}
```

Code within the `Broadcaster` type has full access to `PriceChanged` and can treat it as a delegate. Code outside of `Broadcaster` can perform only `+=` and `-=` operations on the `PriceChanged` event.

Consider the following example. The `Stock` class fires its `PriceChanged` event every time the `Price` of the `Stock` changes:

```
public delegate void PriceChangedHandler (decimal oldPrice,
                                          decimal newPrice);
public class Stock
{
  string symbol;
  decimal price;

  public Stock (string symbol) => this.symbol = symbol;

  public event PriceChangedHandler PriceChanged;

  public decimal Price
  {
    get => price;
    set
    {
      if (price == value) return;      // Exit if nothing has changed
      decimal oldPrice = price;
      price = value;
      if (PriceChanged != null)           // If invocation list not
        PriceChanged (oldPrice, price);   // empty, fire event.
    }
```

```
  }
}
```

HOW DO EVENTS WORK ON THE INSIDE?

Three things happen under the hood when you declare an event as follows:

```
public class Broadcaster
```

```
{
  public event PriceChangedHandler PriceChanged;
}
```

First, the compiler translates the event declaration into something close to the following:

```
PriceChangedHandler priceChanged;   // private delegate
public event PriceChangedHandler PriceChanged
{
  add    { priceChanged += value; }
  remove { priceChanged -= value; }
}
```

The `add` and `remove` keywords denote explicit *event accessors*—which act rather like property accessors. We describe how to write these later.

Second, the compiler looks *within* the `Broadcaster` class for references to `PriceChanged` that perform operations other than `+=` or `-=`, and redirects them to the underlying `priceChanged` delegate field.

Third, the compiler translates `+=` and `-=` operations on the event to calls to the event's `add` and `remove` accessors. Interestingly, this makes the behavior of `+=` and `-=` unique when applied to events: unlike in other scenarios, it's not simply a shortcut for `+` and `-` followed by an assignment.

If we remove the `event` keyword from our example so that `PriceChanged` becomes an ordinary delegate field, our example would give the same results. However, `Stock` would be less robust insomuch as subscribers could do the following things to interfere with one another:

- Replace other subscribers by reassigning `PriceChanged` (instead of using the `+=` operator).
- Clear all subscribers (by setting `PriceChanged` to `null`).
- Broadcast to other subscribers by invoking the delegate.

> **NOTE**
>
> Events in Windows Runtime (WinRT) libraries have slightly different semantics in that attaching to an event returns a token, which is required to detach from the event. The compiler transparently bridges this gap (by maintaining an internal dictionary of tokens) so that you can consume WinRT events as though they were ordinary CLR events.

Standard Event Pattern

In almost all cases for which events are defined in the .NET Core library, their definition adheres to a standard pattern designed to provide consistency across library and user code. At the core of the standard event pattern is `System.EventArgs`: a predefined Framework class with no members (other than the static `Empty` property). `EventArgs` is a base class for conveying information for an

event. In our `Stock` example, we would subclass `EventArgs` to convey the old and new prices when a `PriceChanged` event is fired:

```
public class PriceChangedEventArgs : System.EventArgs
{
  public readonly decimal LastPrice;
  public readonly decimal NewPrice;

  public PriceChangedEventArgs (decimal lastPrice, decimal newPrice)
  {
    LastPrice = lastPrice;
    NewPrice = newPrice;
  }
}
```

For reusability, the `EventArgs` subclass is named according to the information it contains (rather than the event for which it will be used). It typically exposes data as properties or as read-only fields.

With an `EventArgs` subclass in place, the next step is to choose or define a delegate for the event. There are three rules:

- It must have a `void` return type.
- It must accept two arguments: the first of type `object`, and the second a subclass of `EventArgs`. The first argument indicates the event broadcaster, and the second argument contains the extra information to convey.
- Its name must end with `EventHandler`.

The Framework defines a generic delegate called `System.EventHandler<>` that satisfies these rules:

```
public delegate void EventHandler<TEventArgs>
  (object source, TEventArgs e) where TEventArgs : EventArgs;
```

> **NOTE**
>
> Before generics existed in the language (prior to C# 2.0), we would have had to instead write a custom delegate as follows:
>
> ```
> public delegate void PriceChangedHandler
>
>   (object sender, PriceChangedEventArgs e);
> ```
>
> For historical reasons, most events within the Framework use delegates defined in this way.

The next step is to define an event of the chosen delegate type. Here, we use the generic `EventHandler` delegate:

```
public class Stock
{
  ...
  public event EventHandler<PriceChangedEventArgs>
PriceChanged;
}
```

Finally, the pattern requires that you write a protected virtual method that fires the event. The name must match the name of the event, prefixed with the word *On*, and then accept a single `EventArgs` argument:

```
public class Stock
{
  ...

  public event EventHandler<PriceChangedEventArgs> PriceChanged;

  protected virtual void OnPriceChanged
(PriceChangedEventArgs e)
  {
    if (PriceChanged != null) PriceChanged (this, e);
  }
}
```

NOTE

To work robustly in multithreaded scenarios (Chapter 14), you need to assign the delegate to a temporary variable before testing and invoking it:

```
var temp = PriceChanged;

if (temp != null) temp (this, e);
```

We can achieve the same functionality without the `temp` variable with the null-conditional operator:

```
PriceChanged?.Invoke (this, e);
```

Being both thread-safe and succinct, this is the best general way to invoke events.

This provides a central point from which subclasses can invoke or override the event (assuming the class is not sealed).

Here's the complete example:

```
using System;

public class PriceChangedEventArgs : EventArgs
{
  public readonly decimal LastPrice;
  public readonly decimal NewPrice;

  public PriceChangedEventArgs (decimal lastPrice, decimal newPrice)
  {
    LastPrice = lastPrice; NewPrice = newPrice;
  }
}

public class Stock
{
  string symbol;
  decimal price;

  public Stock (string symbol) => this.symbol = symbol;

  public event EventHandler<PriceChangedEventArgs> PriceChanged;

  protected virtual void OnPriceChanged
(PriceChangedEventArgs e)
  {
    PriceChanged?.Invoke (this, e);
  }

  public decimal Price
  {
    get => price;
```

```
    set
    {
      if (price == value) return;
      decimal oldPrice = price;
      price = value;
      OnPriceChanged (new PriceChangedEventArgs (oldPrice, price));
    }
  }
}

class Test
{
  static void Main()
  {
    Stock stock = new Stock ("THPW");
    stock.Price = 27.10M;
    // Register with the PriceChanged event
    stock.PriceChanged += stock_PriceChanged;
    stock.Price = 31.59M;
  }

  static void stock_PriceChanged (object sender, PriceChangedEventArgs
e)
  {
    if ((e.NewPrice - e.LastPrice) / e.LastPrice > 0.1M)
      Console.WriteLine ("Alert, 10% stock price increase!");
  }
}
```

The predefined nongeneric `EventHandler` delegate can be used when an event doesn't carry extra information. In this example, we rewrite `Stock` such that the `PriceChanged` event is fired after the price changes, and no information about the event is necessary, other than it happened. We also make use of the `EventArgs.Empty` property in order to avoid unnecessarily instantiating an instance of `EventArgs`:

```
public class Stock
{
  string symbol;
  decimal price;

  public Stock (string symbol) { this.symbol = symbol; }

  public event EventHandler PriceChanged;

  protected virtual void OnPriceChanged (EventArgs e)
  {
    PriceChanged?.Invoke (this, e);
  }

  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      price = value;
      OnPriceChanged (EventArgs.Empty);
    }
  }
}
```

Event Accessors

An event's *accessors* are the implementations of its `+=` and `-=` functions. By default, accessors are implemented implicitly by the compiler. Consider this event declaration:

```
public event EventHandler PriceChanged;
```

The compiler converts this to the following:

- A private delegate field
- A public pair of event accessor functions (`add_PriceChanged` and `remove_PriceChanged`) whose implementations forward the `+=` and `-=` operations to the private delegate field

You can take over this process by defining *explicit* event accessors. Here's a manual implementation of the `PriceChanged` event from our previous example:

```
private EventHandler priceChanged;         // Declare a private delegate

public event EventHandler PriceChanged
{
  add    { priceChanged += value; }
  remove { priceChanged -= value; }
}
```

This example is functionally identical to C#'s default accessor implementation (except that C# also ensures thread safety around updating the delegate via a lock-free compare-and-swap algorithm; see *http://albahari.com/threading*). By defining event accessors ourselves, we instruct C# not to generate default field and accessor logic.

With explicit event accessors, you can apply more complex strategies to the storage and access of the underlying delegate. There are three scenarios for which this is useful:

- When the event accessors are merely relays for another class that is broadcasting the event.

- When the class exposes many events, for which most of the time very few subscribers exist, such as a Windows control. In such cases, it is better to store the subscriber's delegate instances in a dictionary because a dictionary will contain less storage overhead than dozens of null delegate field references.

- When explicitly implementing an interface that declares an event.

Here is an example that illustrates the last point:

```
public interface IFoo { event EventHandler Ev; }

class Foo : IFoo
{
  private EventHandler ev;

  event EventHandler IFoo.Ev
  {
    add    { ev += value; }
    remove { ev -= value; }
  }
}
```

> **NOTE**
>
> The `add` and `remove` parts of an event are compiled to `add_`*XXX* and `remove_`*XXX* methods.

Event Modifiers

Like methods, events can be virtual, overridden, abstract, or sealed. Events can also be static:

```
public class Foo
{
  public static event EventHandler<EventArgs> StaticEvent;
  public virtual event EventHandler<EventArgs> VirtualEvent;
}
```

Lambda Expressions

A *lambda expression* is an unnamed method written in place of a delegate instance. The compiler immediately converts the lambda expression to either of the following:

- A delegate instance.
- An *expression tree*, of type `Expression<TDelegate>`, representing the code inside the lambda expression in a traversable object model. This allows the lambda expression to be interpreted later at runtime (see "Building Query Expressions" in Chapter 8).

Given the following delegate type:

```
delegate int Transformer (int i);
```

we could assign and invoke the lambda expression `x => x * x` as follows:

```
Transformer sqr = x => x * x;
Console.WriteLine (sqr(3));    // 9
```

> **NOTE**
>
> Internally, the compiler resolves lambda expressions of this type by writing a private method and then moving the expression's code into that method.

A lambda expression has the following form:

```
(parameters) => expression-or-statement-block
```

For convenience, you can omit the parentheses if and only if there is exactly one parameter of an inferable type.

In our example, there is a single parameter, `x`, and the expression is `x * x`:

```
x => x * x;
```

Each parameter of the lambda expression corresponds to a delegate parameter, and the type of the expression (which may be `void`) corresponds to the return type of the delegate.

In our example, `x` corresponds to parameter `i`, and the expression `x * x` corresponds to the return type `int`, therefore being compatible with the `Transformer` delegate:

```
delegate int Transformer (int i);
```

A lambda expression's code can be a *statement block* instead of an expression. We can rewrite our example as follows:

```
x => { return x * x; };
```

Lambda expressions are used most commonly with the `Func` and `Action` delegates, so you will most often see our earlier expression written as follows:

```
Func<int,int> sqr = x => x * x;
```

Here's an example of an expression that accepts two parameters:

```
Func<string,string,int> totalLength = (s1, s2) => s1.Length + s2.Length;
int total = totalLength ("hello", "world");   // total is 10;
```

Explicitly Specifying Lambda Parameter Types

The compiler can usually *infer* the type of lambda parameters contextually. When this is not the case, you must specify the type of each parameter explicitly. Consider the following two methods:

```
void Foo<T> (T x)         {}
void Bar<T> (Action<T> a) {}
```

The following code will fail to compile, because the compiler cannot infer the type of `x`:

```
Bar (x => Foo (x));     // What type is x?
```

We can fix this by explicitly specifying x's type as follows:

```
Bar ((int x) => Foo (x));
```

This particular example is simple enough that it can be fixed in two other ways:

```
Bar<int> (x => Foo (x));   // Specify type parameter for Bar
Bar<int> (Foo);            // As above, but with method group
```

Capturing Outer Variables

A lambda expression can reference the local variables and parameters of the method in which it's defined (*outer variables*):

```
static void Main()
{
  int factor = 2;
  Func<int, int> multiplier = n => n * factor;
  Console.WriteLine (multiplier (3));           // 6
}
```

Outer variables referenced by a lambda expression are called *captured variables*. A lambda expression that captures variables is called a *closure*.

> **NOTE**
>
> Variables can also be captured by anonymous methods and local methods. The rules for captured variables, in these cases, are the same.

Captured variables are evaluated when the delegate is actually *invoked*, not when the variables were *captured*:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
factor = 10;
Console.WriteLine (multiplier (3));           // 30
```

Lambda expressions can themselves update captured variables:

```
int seed = 0;
Func<int> natural = () => seed++;
Console.WriteLine (natural());           // 0
Console.WriteLine (natural());           // 1
Console.WriteLine (seed);                // 2
```

Captured variables have their lifetimes extended to that of the delegate. In the following example, the local variable `seed` would ordinarily disappear from scope when `Natural` finished executing. But because `seed` has been *captured*, its lifetime is extended to that of the capturing delegate, `natural`:

```
static Func<int> Natural()
{
  int seed = 0;
```

```
  return () => seed++;      // Returns a closure
}

static void Main()
{
  Func<int> natural = Natural();
  Console.WriteLine (natural());      // 0
  Console.WriteLine (natural());      // 1
}
```

A local variable *instantiated* within a lambda expression is unique per invocation of the delegate instance. If we refactor our previous example to instantiate `seed` *within* the lambda expression, we get a different (in this case, undesirable) result:

```
static Func<int> Natural()
{
  return() => { int seed = 0; return seed++; };
}

static void Main()
{
  Func<int> natural = Natural();
  Console.WriteLine (natural());           // 0
  Console.WriteLine (natural());           // 0
}
```

NOTE

Capturing is internally implemented by “hoisting” the captured variables into fields of a private class. When the method is called, the class is instantiated and lifetime-bound to the delegate instance.

CAPTURING ITERATION VARIABLES

When you capture the iteration variable of a `for` loop, C# treats that variable as though it were declared *outside* the loop. This means that the *same* variable is captured in each iteration. The following program writes `333` instead of writing `012`:

```
Action[] actions = new Action[3];

for (int i = 0; i < 3; i++)
  actions [i] = () => Console.Write (i);

foreach (Action a in actions) a();     // 333
```

Each closure (shown in boldface) captures the same variable, `i`. (This actually makes sense when you consider that `i` is a variable whose value persists between loop iterations; you can even explicitly change `i` within the loop body if you want.) The consequence is that when the delegates are later invoked, each delegate sees `i`'s value at the time of *invocation*—which is 3. We can illustrate this better by expanding the `for` loop, as follows:

```
Action[] actions = new Action[3];
int i = 0;
actions[0] = () => Console.Write (i);
i = 1;
actions[1] = () => Console.Write (i);
i = 2;
actions[2] = () => Console.Write (i);
i = 3;
foreach (Action a in actions) a();    // 333
```

The solution, if we want to write `012`, is to assign the iteration variable to a local variable that's scoped *within* the loop:

```
Action[] actions = new Action[3];
for (int i = 0; i < 3; i++)
{
  int loopScopedi = i;
  actions [i] = () => Console.Write (loopScopedi);
}
foreach (Action a in actions) a();     // 012
```

Because `loopScopedi` is freshly created on every iteration, each closure captures a *different* variable.

> **NOTE**
>
> Prior to C# 5.0, `foreach` loops worked in the same way:
>
> ```
> Action[] actions = new Action[3];
> int i = 0;
>
> foreach (char c in "abc")
>   actions [i++] = () => Console.Write (c);
>
> foreach (Action a in actions) a();   // ccc in C# 4.0
> ```
>
> This caused considerable confusion: unlike with a `for` loop, the iteration variable in a `foreach` loop is immutable, and so you would expect it to be treated as local to the loop body. The good news is that it's been fixed since C# 5.0, and the preceding example now writes "abc".

Lambda Expressions Versus Local Methods

The functionality of local methods (see "Local methods" in Chapter 3) overlaps with that of lambda expressions. Local methods have the following three advantages:

- They can be recursive (they can call themselves), without ugly hacks
- They avoid the clutter of specifying a delegate type
- They incur slightly less overhead

Local methods are more efficient because they avoid the indirection of a delegate (which costs some CPU cycles and a memory allocation). They can also access local variables of the containing method without the compiler having to "hoist" the captured variables into a hidden class.

However, in many cases you *need* a delegate—most commonly when calling a higher-order function, that is, a method with a delegate-typed parameter:

```
public void Foo (Func<int,bool> predicate) { ... }
```

(You can see plenty more of these in Chapter 8). In such cases, you need a delegate anyway, and it's in precisely these cases that lambda expressions are usually terser and cleaner.

Anonymous Methods

Anonymous methods are a C# 2.0 feature was mostly subsumed by C# 3.0's lambda expressions. An anonymous method is like a lambda expression, but it lacks the following features:

- Implicitly typed parameters
- Expression syntax (an anonymous method must always be a statement block)
- The ability to compile to an expression tree, by assigning to `Expression<T>`

To write an anonymous method, you include the `delegate` keyword followed (optionally) by a parameter declaration and then a method body. For example, given this delegate:

```
delegate int Transformer (int i);
```

we could write and call an anonymous method as follows:

```
Transformer sqr = delegate (int x) {return x * x;};
Console.WriteLine (sqr(3));                          // 9
```

The first line is semantically equivalent to the following lambda expression:

```
Transformer sqr =      (int x) => {return x * x;};
```

Or, simply:

```
Transformer sqr =           x  => x * x;
```

Anonymous methods capture outer variables in the same way lambda expressions do.

> **NOTE**
>
> A unique feature of anonymous methods is that you can omit the parameter declaration entirely—even if the delegate expects it. This can be useful in declaring events with a default empty handler:
>
> ```
> public event EventHandler Clicked = delegate { };
> ```
>
> This avoids the need for a null check before firing the event. The following is also legal:
>
> ```
> // Notice that we omit the parameters:
> Clicked += delegate { Console.WriteLine ("clicked"); };
> ```

try Statements and Exceptions

A `try` statement specifies a code block subject to error-handling or cleanup code. The `try` *block* must be followed by one or more `catch` *blocks*, a `finally` *block*, or both. The `catch` block executes when an error is thrown in the `try` block. The `finally` block executes after execution leaves the `try` block (or if present, the `catch` block), to perform cleanup code, regardless of whether an exception was thrown.

A `catch` block has access to an `Exception` object that contains information about the error. You use a `catch` block to either compensate for the error or *rethrow* the exception. You rethrow an exception if you merely want to log the problem or if you want to rethrow a new, higher-level exception type.

A `finally` block adds determinism to your program: the CLR endeavors to always execute it. It's useful for cleanup tasks such as closing network connections.

A `try` statement looks like this:

```
try
{
  ... // exception may get thrown within execution of this block
}
catch (ExceptionA ex)
{
  ... // handle exception of type ExceptionA
}
catch (ExceptionB ex)
{
  ... // handle exception of type ExceptionB
}
finally
{
  ... // cleanup code
}
```

Consider the following program:

```
class Test
```

```
{
  static int Calc (int x) => 10 / x;

  static void Main()
  {
    int y = Calc (0);
    Console.WriteLine (y);
  }
}
```

Because `x` is zero, the runtime throws a `DivideByZeroException`, and our program terminates. We can prevent this by catching the exception as follows:

```
class Test
{
  static int Calc (int x) => 10 / x;

  static void Main()
  {
    try
    {
      int y = Calc (0);
      Console.WriteLine (y);
    }
    catch (DivideByZeroException ex)
    {
      Console.WriteLine ("x cannot be zero");
    }
    Console.WriteLine ("program completed");
  }
}

OUTPUT:
x cannot be zero
program completed
```

> **NOTE**
>
> This is a simple example to illustrate exception handling. We could deal with this particular scenario better in practice by checking explicitly for the divisor being zero before calling `Calc`.
>
> Checking for preventable errors is preferable to relying on `try`/`catch` blocks because exceptions are relatively expensive to handle, taking hundreds of clock cycles or more.

When an exception is thrown within a `try` statement, the CLR performs a test:

Does the `try` *statement have any compatible* `catch` *blocks?*

- If so, execution jumps to the compatible `catch` block, followed by the `finally` block (if present), and then execution continues normally.
- If not, execution jumps directly to the `finally` block (if present), then the CLR looks up the call stack for other `try` blocks; if found, it repeats the test.

If no function in the call stack takes responsibility for the exception, the program terminates.

The catch Clause

A `catch` clause specifies what type of exception to catch. This must either be `System.Exception` or a subclass of `System.Exception`.

Catching `System.Exception` catches all possible errors. This is useful in the following circumstances:

- Your program can potentially recover regardless of the specific exception type.
- You plan to rethrow the exception (perhaps after logging it).
- Your error handler is the last resort, prior to termination of the program.

More typically, though, you catch *specific exception types* in order to avoid having to deal with circumstances for which your handler wasn't designed (e.g., an `OutOfMemoryException`).

You can handle multiple exception types with multiple `catch` clauses (again, this example could be written with explicit argument checking rather than exception handling):

```
class Test
{
  static void Main (string[] args)
  {
    try
    {
      byte b = byte.Parse (args[0]);
      Console.WriteLine (b);
    }
    catch (IndexOutOfRangeException ex)
    {
      Console.WriteLine ("Please provide at least one argument");
    }
    catch (FormatException ex)
```

```
    {
      Console.WriteLine ("That's not a number!");
    }
    catch (OverflowException ex)
    {
      Console.WriteLine ("You've given me more than a byte!");
    }
  }
}
```

Only one `catch` clause executes for a given exception. If you want to include a safety net to catch more general exceptions (such as `System.Exception`), you must put the more-specific handlers *first*.

An exception can be caught without specifying a variable, if you don’t need to access its properties:

```
catch (OverflowException)   // no variable
{
  ...
}
```

Furthermore, you can omit both the variable and the type (meaning that all exceptions will be caught):

```
catch { ... }
```

EXCEPTION FILTERS

You can specify an *exception filter* in a `catch` clause by adding a `when` clause:

```
catch (WebException ex) when (ex.Status ==
WebExceptionStatus.Timeout)
{
  ...
}
```

If a `WebException` is thrown in this example, the Boolean expression following the `when` keyword is then evaluated. If the result is false, the `catch` block in question is ignored and any subsequent `catch` clauses are considered. With exception filters, it can be meaningful to catch the same exception type again:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{ ... }
catch (WebException ex) when (ex.Status ==
WebExceptionStatus.SendFailure)
{ ... }
```

The Boolean expression in the `when` clause can be side-effecting, as with a method that logs the exception for diagnostic purposes.

The finally Block

A `finally` block always executes—regardless of whether an exception is thrown and whether the `try` block runs to completion. You typically use `finally` blocks for cleanup code.

A `finally` block executes after any of the following:

- A `catch` block finishes (or throws a new exception)

- The `try` block finishes (or throws an exception for which there's no `catch` block)
- Control leaves the `try` block because of a `jump` statement (e.g., `return` or `goto`)

The only things that can defeat a `finally` block are an infinite loop or the process ending abruptly.

A `finally` block helps add determinism to a program. In the following example, the file that we open *always* gets closed, regardless of whether:

- The `try` block finishes normally
- Execution returns early because the file is empty (`EndOfStream`)
- An `IOException` is thrown while reading the file

```
static void ReadFile()
{
  StreamReader reader = null;    // In System.IO namespace
  try
  {
    reader = File.OpenText ("file.txt");
    if (reader.EndOfStream) return;
    Console.WriteLine (reader.ReadToEnd());
  }
  finally
  {
    if (reader != null) reader.Dispose();
  }
}
```

In this example, we closed the file by calling `Dispose` on the `StreamReader`. Calling `Dispose` on an object, within a `finally` block, is a standard convention throughout .NET Core and is supported explicitly in C# through the `using` statement.

THE USING STATEMENT

Many classes encapsulate unmanaged resources, such as file handles, graphics handles, or database connections. These classes implement `System.IDisposable`, which defines a single parameterless method named `Dispose` to clean up these resources. The `using` statement provides an elegant syntax for calling `Dispose` on an `IDisposable` object within a `finally` block. Thus:

```
using (StreamReader reader = File.OpenText ("file.txt"))
{
  ...
}
```

is precisely equivalent to:

```
{
  StreamReader reader = File.OpenText ("file.txt");
  try
  {
    ...
  }
  finally
  {
    if (reader != null)
      ((IDisposable)reader).Dispose();
```

```
  }
}
```

`USING` DECLARATIONS (C# 8)

If you omit the brackets and statement block following a `using` statement, it becomes a *using declaration*. The resource is then disposed when execution falls outside the *enclosing* statement block:

```
if (File.Exists ("file.txt"))
{
  using var reader = File.OpenText ("file.txt");
  Console.WriteLine (reader.ReadLine());
  ...
}
```

In this case, `reader` will be disposed when execution falls outside the `if` statement block.

Throwing Exceptions

Exceptions can be thrown either by the runtime or in user code. In this example, `Display` throws a `System.ArgumentNullException`:

```
class Test
{
  static void Display (string name)
  {
    if (name == null)
      throw new ArgumentNullException (nameof (name));

    Console.WriteLine (name);
  }
```

```
  static void Main()
  {
    try { Display (null); }
    catch (ArgumentNullException ex)
    {
      Console.WriteLine ("Caught the exception");
    }
  }
}
```

THROW EXPRESSIONS

`throw` can also appear as an expression in expression-bodied functions:

```
public string Foo() => throw new NotImplementedException();
```

A `throw` expression can also appear in a ternary conditional expression:

```
string ProperCase (string value) =>
  value == null ? throw new ArgumentException ("value") :
  value == "" ? "" :
  char.ToUpper (value[0]) + value.Substring (1);
```

RETHROWING AN EXCEPTION

You can capture and rethrow an exception as follows:

```
try {  ...  }
catch (Exception ex)
{
  // Log error
```

```
  ...
  throw;           // Rethrow same exception
}
```

> **NOTE**
>
> If we replaced `throw` with `throw ex`, the example would still work, but the `StackTrace` property of the newly propagated exception would no longer reflect the original error.

Rethrowing in this manner lets you log an error without *swallowing* it. It also lets you back out of handling an exception should circumstances turn out to be beyond what you expected:

```
using System.Net;       // (See Chapter 16)
...

string s = null;
using (WebClient wc = new WebClient())
  try { s = wc.DownloadString ("http://www.albahari.com/nutshell/");  }
  catch (WebException ex)
  {
    if (ex.Status == WebExceptionStatus.Timeout)
      Console.WriteLine ("Timeout");
    Else
      throw;     // Can't handle other sorts of WebException,
so rethrow
  }
```

This can be written more tersely with an exception filter:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
```

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

The other common scenario is to rethrow a more specific exception type:

```
try
{
  ... // Parse a DateTime from XML element data
}
catch (FormatException ex)
{
  throw new XmlException ("Invalid DateTime", ex);
}
```

Notice that when we constructed `XmlException`, we passed in the original exception, `ex`, as the second argument. This argument populates the `InnerException` property of the new exception and aids debugging. Nearly all types of exception offer a similar constructor.

Rethrowing a *less*-specific exception is something you might do when crossing a trust boundary, so as not to leak technical information to potential hackers.

Key Properties of System.Exception

The most important properties of `System.Exception` are the following:

`StackTrace`

A string representing all the methods that are called from the origin of the exception to the `catch` block.

`Message`
: A string with a description of the error.

`InnerException`
: The inner exception (if any) that caused the outer exception. This, itself, can have another `InnerException`.

> **NOTE**
>
> All exceptions in C# are runtime exceptions—there is no equivalent to Java's compile-time checked exceptions.

Common Exception Types

The following exception types are used widely throughout the CLR and .NET Core. You can throw these yourself or use them as base classes for deriving custom exception types.

`System.ArgumentException`
: Thrown when a function is called with a bogus argument. This generally indicates a program bug.

`System.ArgumentNullException`
: Subclass of `ArgumentException` that's thrown when a function argument is (unexpectedly) `null`.

`System.ArgumentOutOfRangeException`

Subclass of `ArgumentException` that's thrown when a (usually numeric) argument is too big or too small. For example, this is thrown when passing a negative number into a function that accepts only positive values.

`System.InvalidOperationException`
: Thrown when the state of an object is unsuitable for a method to successfully execute, regardless of any particular argument values. Examples include reading an unopened file or getting the next element from an enumerator for which the underlying list has been modified partway through the iteration.

`System.NotSupportedException`
: Thrown to indicate that a particular functionality is not supported. A good example is calling the `Add` method on a collection for which `IsReadOnly` returns `true`.

`System.NotImplementedException`
: Thrown to indicate that a function has not yet been implemented.

`System.ObjectDisposedException`
: Thrown when the object upon which the function is called has been disposed.

Another commonly encountered exception type is `NullReferenceException`. The CLR throws this exception when you attempt to access a member of an object whose value is `null` (indicating a bug in your code). You can throw a `NullReferenceException` directly (for testing purposes) as follows:

```
throw null;
```

The Try XXX Method Pattern

When writing a method, you have a choice, when something goes wrong, to return some kind of failure code or throw an exception. In general, you throw an exception when the error is outside the normal workflow—or if you expect that the immediate caller won't be able to cope with it. Occasionally, though, it can be best to offer both choices to the consumer. An example of this is the `int` type, which defines two versions of its `Parse` method:

```
public int Parse    (string input);
public bool TryParse (string input, out int returnValue);
```

If parsing fails, `Parse` throws an exception; `TryParse` returns `false`.

You can implement this pattern by having the *XXX* method call the `Try`*XXX* method:

```
public return-type XXX (input-type input)
{
  return-type returnValue;
  if (!TryXXX (input, out returnValue))
    throw new YYYException (...)
  return returnValue;
}
```

Alternatives to Exceptions

As with `int.TryParse`, a function can communicate failure by sending an error code back to the calling function via a return type or parameter. Although this can work with simple and predictable

failures, it becomes clumsy when extended to all errors, polluting method signatures and creating unnecessary complexity and clutter. It also cannot generalize to functions that are not methods, such as operators (e.g., the division operator) or properties. An alternative is to place the error in a common place where all functions in the call stack can see it (e.g., a static method that stores the current error per thread). This, though, requires each function to participate in an error-propagation pattern, which is cumbersome and, ironically, itself error prone.

Enumeration and Iterators

Enumeration

An *enumerator* is a read-only, forward-only cursor over a *sequence of values*. C# treats a type as an enumerator if it does any of the following:

- Implements `System.Collections.IEnumerator`
- Implements `System.Collections.Generic.IEnumerator<T>`
- Has a public parameterless method named `MoveNext` and property called `Current`

The `foreach` statement iterates over an *enumerable* object. An enumerable object is the logical representation of a sequence. It is not itself a cursor, but an object that produces cursors over itself. C# treats a type as enumerable if it does any of the following:

- Implements `System.Collections.IEnumerable`
- Implements `System.Collections.Generic.IEnumerable<T>`
- Has a public parameterless method named `GetEnumerator` that returns an *enumerator*

The enumeration pattern is as follows:

```
class Enumerator   // Typically implements IEnumerator or
IEnumerator<T>
{
  public IteratorVariableType Current { get {...} }
  public bool MoveNext() {...}
}

class Enumerable   // Typically implements IEnumerable or
IEnumerable<T>
{
  public Enumerator GetEnumerator() {...}
}
```

Here is the high-level way of iterating through the characters in the word *beer* using a `foreach` statement:

```
foreach (char c in "beer")
  Console.WriteLine (c);
```

Here is the low-level way of iterating through the characters in *beer* without using a `foreach` statement:

```
using (var enumerator = "beer".GetEnumerator())
```

```
while (enumerator.MoveNext())
{
  var element = enumerator.Current;
  Console.WriteLine (element);
}
```

If the enumerator implements `IDisposable`, the `foreach` statement also acts as a `using` statement, implicitly disposing the enumerator object.

Chapter 7 explains the enumeration interfaces in further detail.

Collection Initializers

You can instantiate and populate an enumerable object in a single step:

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

List<int> list = new List<int> {1, 2, 3};
```

The compiler translates this to the following:

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

List<int> list = new List<int>();
list.Add (1);
list.Add (2);
list.Add (3);
```

This requires that the enumerable object implements the `System.Collections.IEnumerable` interface, and that it has an `Add` method that has the appropriate number of parameters for the call. You can similarly initialize dictionaries (see "Dictionaries" in Chapter 7) as follows:

```
var dict = new Dictionary<int, string>()
{
  { 5, "five" },
  { 10, "ten" }
};
```

Or, more succinctly:

```
var dict = new Dictionary<int, string>()
{
  [3] = "three",
  [10] = "ten"
};
```

The latter is valid not only with dictionaries, but with any type for which an indexer exists.

Iterators

Whereas a `foreach` statement is a *consumer* of an enumerator, an iterator is a *producer* of an enumerator. In this example, we use an iterator to return a sequence of Fibonacci numbers (where each number is the sum of the previous two):

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

class Test
{
  static void Main()
  {
    foreach (int fib in Fibs(6))
      Console.Write (fib + "  ");
  }

  static IEnumerable<int> Fibs (int fibCount)
  {
    for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
    {
      yield return prevFib;
      int newFib = prevFib + curFib;
      prevFib = curFib;
      curFib = newFib;
    }
  }
}

OUTPUT: 1  1  2  3  5  8
```

Whereas a `return` statement expresses “Here’s the value you asked me to return from this method,” a `yield return` statement expresses “Here’s the next element you asked me to yield from this enumerator.” On each `yield` statement, control is returned to the caller, but the callee’s state is maintained so that the method can continue executing as soon as the caller enumerates the next element. The lifetime of this state is bound to the enumerator such that the state can be released when the caller has finished enumerating.

> **NOTE**
>
> The compiler converts iterator methods into private classes that implement `IEnumerable<T>` and/or `IEnumerator<T>`. The logic within the iterator block is "inverted" and spliced into the `MoveNext` method and `Current` property on the compiler-written enumerator class. This means that when you call an iterator method, all you're doing is instantiating the compiler-written class; none of your code actually runs! Your code runs only when you start enumerating over the resultant sequence, typically with a `foreach` statement.

Iterators can be local methods (see "Local methods" in Chapter 3).

Iterator Semantics

An iterator is a method, property, or indexer that contains one or more `yield` statements. An iterator must return one of the following four interfaces (otherwise, the compiler will generate an error):

```
// Enumerable interfaces
System.Collections.IEnumerable
System.Collections.Generic.IEnumerable<T>

// Enumerator interfaces
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

An iterator has different semantics, depending on whether it returns an *enumerable* interface or an *enumerator* interface. We describe this in Chapter 7.

Multiple yield statements are permitted:

```
class Test
{
  static void Main()
  {
    foreach (string s in Foo())
      Console.WriteLine(s);         // Prints "One","Two","Three"
  }

  static IEnumerable<string> Foo()
  {
    yield return "One";
    yield return "Two";
    yield return "Three";
  }
}
```

YIELD BREAK

A `return` statement is illegal in an iterator block; instead you must use the `yield break` statement to indicate that the iterator block should exit early, without returning more elements. We can modify `Foo` as follows to demonstrate:

```
static IEnumerable<string> Foo (bool breakEarly)
{
  yield return "One";
  yield return "Two";

  if (breakEarly)
    yield break;

  yield return "Three";
}
```

ITERATORS AND TRY/CATCH/FINALLY BLOCKS

A `yield return` statement cannot appear in a `try` block that has a `catch` clause:

```
IEnumerable<string> Foo()
{
  try { yield return "One"; }    // Illegal
  catch { ... }
}
```

Nor can `yield return` appear in a `catch` or `finally` block. These restrictions are due to the fact that the compiler must translate iterators into ordinary classes with `MoveNext`, `Current`, and `Dispose` members, and translating exception handling blocks would create excessive complexity.

You can, however, yield within a `try` block that has (only) a `finally` block:

```
IEnumerable<string> Foo()
{
  try { yield return "One"; }    // OK
  finally { ... }
}
```

The code in the `finally` block executes when the consuming enumerator reaches the end of the sequence or is disposed. A `foreach` statement implicitly disposes the enumerator if you break early, making this a safe way to consume enumerators. When working with enumerators explicitly, a trap is to abandon enumeration early without disposing it, circumventing the `finally`

block. You can avoid this risk by wrapping explicit use of enumerators in a `using` statement:

```
string firstElement = null;
var sequence = Foo();
using (var enumerator = sequence.GetEnumerator())
  if (enumerator.MoveNext())
    firstElement = enumerator.Current;
```

Composing Sequences

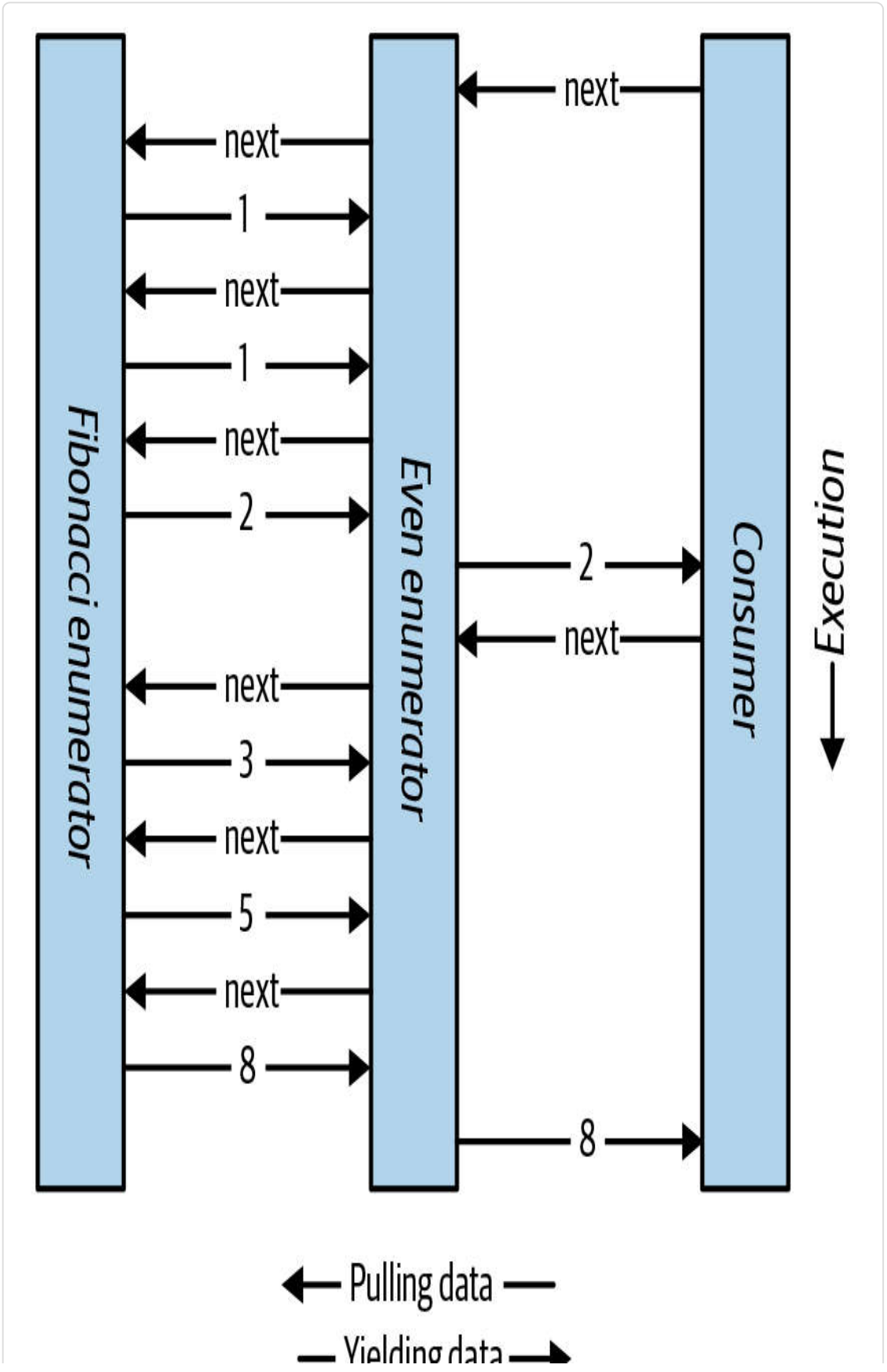
Iterators are highly composable. We can extend our example, this time to output even Fibonacci numbers only:

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

class Test
{
  static void Main()
  {
    foreach (int fib in EvenNumbersOnly (Fibs(6)))
      Console.WriteLine (fib);
  }

  static IEnumerable<int> Fibs (int fibCount)
  {
    for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
    {
      yield return prevFib;
      int newFib = prevFib + curFib;
      prevFib = curFib;
      curFib = newFib;
    }
  }
```

```
  static IEnumerable<int> EvenNumbersOnly (IEnumerable<int> sequence)
  {
    foreach (int x in sequence)
      if ((x % 2) == 0)
        yield return x;
  }
}
```

Each element is not calculated until the last moment—when requested by a `MoveNext()` operation. Figure 4-1 shows the data requests and output over time.

The composability of the iterator pattern is extremely useful in LINQ; we discuss the subject again in Chapter 8.


Fibonacci enumerator
Even enumerator
Consumer
Execution
next
1
next
1
next
2
next
3
next
5
next
8
next
2
next
8
Pulling data
Yielding data

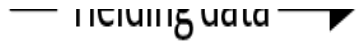


Figure 4-1. Composing sequences

Nullable Value Types

Reference types can represent a nonexistent value with a null reference. Value types, however, cannot ordinarily represent null values:

```
string s = null;       // OK, reference type
int i = null;          // Compile error, value type cannot be null
```

To represent null in a value type, you must use a special construct called a *nullable type*. A nullable type is denoted with a value type followed by the `?` symbol:

```
int? i = null;                    // OK, nullable type
Console.WriteLine (i == null);    // True
```

Nullable<T> Struct

`T?` translates into `System.Nullable<T>`, which is a lightweight immutable structure, having only two fields, to represent `Value` and `HasValue`. The essence of `System.Nullable<T>` is very simple:

```
public struct Nullable<T> where T : struct
{
  public T Value {get;}
  public bool HasValue {get;}
  public T GetValueOrDefault();
```

```
  public T GetValueOrDefault (T defaultValue);
  ...
}
```

The code:

```
int? i = null;
Console.WriteLine (i == null);              // True
```

translates to:

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine (! i.HasValue);           // True
```

Attempting to retrieve `Value` when `HasValue` is false throws an `InvalidOperationException`. `GetValueOrDefault()` returns `Value` if `HasValue` is true; otherwise, it returns `new T()` or a specified custom default value.

The default value of `T?` is `null`.

Implicit and Explicit Nullable Conversions

The conversion from `T` to `T?` is implicit, and from `T?` to `T` is explicit:

```
int? x = 5;        // implicit
int y = (int)x;    // explicit
```

The explicit cast is directly equivalent to calling the nullable object's `Value` property. Hence, an `InvalidOperationException` is thrown

if `HasValue` is false.

Boxing and Unboxing Nullable Values

When `T?` is boxed, the boxed value on the heap contains `T`, not `T?`. This optimization is possible because a boxed value is a reference type that can already express null.

C# also permits the unboxing of nullable value types with the `as` operator. The result will be `null` if the cast fails:

```
object o = "string";
int? x = o as int?;
Console.WriteLine (x.HasValue);   // False
```

Operator Lifting

The `Nullable<T>` struct does not define operators such as `<`, `>`, or even `==`. Despite this, the following code compiles and executes correctly:

```
int? x = 5;
int? y = 10;
bool b = x < y;      // true
```

This works because the compiler borrows or *lifts* the less-than operator from the underlying value type. Semantically, it translates the preceding comparison expression into this:

```
bool b = (x.HasValue && y.HasValue) ? (x.Value < y.Value) : false;
```

In other words, if both `x` and `y` have values, it compares via `int`'s less-than operator; otherwise, it returns `false`.

Operator lifting means that you can implicitly use `T`'s operators on `T?`. You can define operators for `T?` in order to provide special-purpose null behavior, but in the vast majority of cases, it's best to rely on the compiler automatically applying systematic nullable logic for you. Here are some examples:

```
int? x = 5;
int? y = null;

// Equality operator examples
Console.WriteLine (x == y);    // False
Console.WriteLine (x == null); // False
Console.WriteLine (x == 5);    // True
Console.WriteLine (y == null); // True
Console.WriteLine (y == 5);    // False
Console.WriteLine (y != 5);    // True

// Relational operator examples
Console.WriteLine (x < 6);     // True
Console.WriteLine (y < 6);     // False
Console.WriteLine (y > 6);     // False

// All other operator examples
Console.WriteLine (x + 5);     // 10
Console.WriteLine (x + y);     // null (prints empty line)
```

The compiler performs null logic differently depending on the category of operator. The following sections explain these different rules.

EQUALITY OPERATORS (== AND !=)

Lifted equality operators handle nulls just like reference types do. This means that two null values are equal:

```
Console.WriteLine (       null ==        null);   // True
Console.WriteLine ((bool?)null == (bool?)null);   // True
```

Further:

- If exactly one operand is null, the operands are unequal.
- If both operands are non-null, their `Value`s are compared.

RELATIONAL OPERATORS (<, <=, >=, >)

The relational operators work on the principle that it is meaningless to compare null operands. This means that comparing a null value to either a null or a non-null value returns `false`:

```
bool b = x < y;    // Translation:

bool b = (x.HasValue && y.HasValue)
         ? (x.Value < y.Value)
         : false;

// b is false (assuming x is 5 and y is null)
```

ALL OTHER OPERATORS (+, −, *, /, %, &, |, ^, <<, >>, +, ++, --, !, ~)

These operators return null when any of the operands are null. This pattern should be familiar to SQL users:

```
int? c = x + y;   // Translation:

int? c = (x.HasValue && y.HasValue)
         ? (int?) (x.Value + y.Value)
         : null;

// c is null (assuming x is 5 and y is null)
```

An exception is when the `&` and `|` operators are applied to `bool?`, which we discuss shortly.

MIXING NULLABLE AND NON-NULLABLE OPERATORS

You can mix and match nullable and non-nullable value types (this works because there is an implicit conversion from `T` to `T?`):

```
int? a = null;
int b = 2;
int? c = a + b;   // c is null - equivalent to a + (int?)b
```

bool? with & and | Operators

When supplied operands of type `bool?` the `&` and `|` operators treat `null` as an *unknown value*. So, `null | true` is true because:

- If the unknown value is false, the result would be true.
- If the unknown value is true, the result would be true.

Similarly, `null & false` is false. This behavior would be familiar to SQL users. The following example enumerates other combinations:

```
bool? n = null;
bool? f = false;
bool? t = true;
Console.WriteLine (n | n);    // (null)
Console.WriteLine (n | f);    // (null)
Console.WriteLine (n | t);    // True
Console.WriteLine (n & n);    // (null)
Console.WriteLine (n & f);    // False
Console.WriteLine (n & t);    // (null)
```

Nullable Value Types & Null Operators

Nullable value types work particularly well with the `??` operator (see "Null-Coalescing Operator" in Chapter 2), as illustrated in this example:

```
int? x = null;
int y = x ?? 5;        // y is 5

int? a = null, b = 1, c = 2;
Console.WriteLine (a ?? b ?? c);  // 1 (first non-null value)
```

Using `??` on a nullable value type is equivalent to calling `GetValueOrDefault` with an explicit default value, except that the expression for the default value is never evaluated if the variable is not null.

Nullable value types also work well with the null-conditional operator (see "Null-Conditional Operator" in Chapter 2). In the following example, `length` evaluates to null:

```
System.Text.StringBuilder sb = null;
int? length = sb?.ToString().Length;
```

We can combine this with the null-coalescing operator to evaluate to zero instead of null:

```
int length = sb?.ToString().Length ?? 0;  // Evaluates to 0 if sb is
null
```

Scenarios for Nullable Value Types

One of the most common scenarios for nullable value types is to represent unknown values. This frequently occurs in database programming, where a class is mapped to a table with nullable columns. If these columns are strings (e.g., an EmailAddress column on a Customer table), there is no problem because `string` is a reference type in the CLR, which can be null. However, most other SQL column types map to CLR struct types, making nullable value types very useful when mapping SQL to the CLR:

```
// Maps to a Customer table in a database
public class Customer
{
  ...
  public decimal? AccountBalance;
}
```

A nullable type can also be used to represent the backing field of what's sometimes called an *ambient property*. An ambient property, if null, returns the value of its parent:

```
public class Row
{
  ...
  Grid parent;
  Color? color;

  public Color Color
  {
    get { return color ?? parent.Color; }
    set { color = value == parent.Color ? (Color?)null : value; }
  }
}
```

Alternatives to Nullable Value Types

Before nullable value types were part of the C# language (i.e., before C# 2.0), there were many strategies to deal with nullable value types, examples of which still appear in .NET Core for historical reasons. One strategy is to designate a particular non-null value as the “null value”; an example is in the `string` and `array` classes. `string.IndexOf` returns the magic value of `-1` when the character is not found:

```
int i = "Pink".IndexOf ('b');
Console.WriteLine (i);         // -1
```

However, `Array.IndexOf` returns `-1` only if the index is 0-bounded. The more general formula is that `IndexOf` returns one less than the lower bound of the array. In the next example, `IndexOf` returns `0` when an element is not found:

```
// Create an array whose lower bound is 1 instead of 0:

Array a = Array.CreateInstance (typeof (string),
                                new int[] {2}, new int[] {1});
a.SetValue ("a", 1);
a.SetValue ("b", 2);
Console.WriteLine (Array.IndexOf (a, "c"));  // 0
```

Nominating a “magic value” is problematic for several reasons:

- It means that each value type has a different representation of null. In contrast, nullable value types provide one common pattern that works for all value types.
- There might be no reasonable designated value. In the previous example, −1 could not always be used. The same is true for our earlier example representing an unknown account balance.
- Forgetting to test for the magic value results in an incorrect value that might go unnoticed until later in execution—when it pulls an unintended magic trick. Forgetting to test `HasValue` on a null value, however, throws an `InvalidOperationException` on the spot.
- The ability for a value to be null is not captured in the *type*. Types communicate the intention of a program, allow the compiler to check for correctness, and enable a consistent set of rules enforced by the compiler.

Nullable Reference Types (C# 8)

Whereas *nullable value types* bring nullability to value types, *nullable reference types* do the opposite and bring (a degree of) *non-*

nullability to reference types, with the purpose of helping to avoid `NullReferenceException`s.

Nullable reference types introduce a level of safety that's enforced purely by the compiler, in the form of warnings when it detects code that's at risk of generating a `NullReferenceException`.

To enable nullable reference types, you must either add the `Nullable` element to your *.csproj* project file (if you want to enable it for the entire project):

```
<PropertyGroup>
  <Nullable>enable</Nullable>
</PropertyGroup>
```

and/or use the following directives in your code, in the places where it should take effect:

```
#nullable enable   // enables  nullable reference types from this point
on
#nullable disable  // disables nullable reference types from this point
on
#nullable restore  // resets   nullable reference types to project
setting
```

After being enabled, the compiler makes non-nullability the default: if you want a reference type to accept nulls, you must apply the `?` suffix to indicate a *nullable reference type*. In the following example, `s1` is non-nullable, whereas `s2` is nullable:

```
#nullable enable    // Enable nullable reference types

string s1 = null;   // Generates a compiler warning!
string? s2 = null;  // OK: s2 is nullable reference type
```

> **NOTE**
>
> Because nullable reference types are compile-time constructs, there's no runtime difference between `string` and `string?`. In contrast, nullable value types introduce something concrete into the type system, namely the `Nullable<T>` struct.

The following also generates a warning because `x` is not initialized:

```
class Foo { string x; }
```

The warning disappears if you initialize `x`, either via a field initializer, or via code in the constructor.

The Null-Forgiving Operator

The compiler also warns you upon dereferencing a nullable reference type, if it thinks a `NullReferenceException` might occur. In the following example, accessing the string's `Length` property generates a warning:

```
void Foo (string? s) => Console.Write (s.Length);
```

You can remove the warning with the *null-forgiving operator* (`!`):

```
void Foo (string? s) => Console.Write (s!.Length);
```

Our use of the null-forgiving operator in this example is dangerous in that we could end up throwing the very `NullReferenceException` we were trying to avoid in the first place. We could fix it as follows:

```
void Foo (string? s)
{
  if (s != null) Console.Write (s.Length);
}
```

Notice that now we don't need the null-forgiving operator. This is because the compiler performs *static flow analysis* and is smart enough to infer—at least in simple cases—when a dereference is safe and there's no chance of a `NullReferenceException`.

The compiler's ability to detect and warn is not bulletproof, and there are also limits to what's possible in terms of coverage. For instance, the compiler is unable to know whether an array's elements have been populated, and so the following does not generate a warning:

```
var strings = new string[10];
Console.WriteLine (strings[0].Length);
```

Separating the Annotation and Warning Contexts

Enabling nullable reference types via the `#nullable enable` directive (or the `<Nullable>enable</Nullable>` project setting) does two things:

- It enables the *nullable annotation context*, which tells the compiler to treat all reference-type variable declarations as non-nullable unless suffixed by the `?` symbol.
- It enables the *nullable warning context*, which tells the compiler to generate warnings upon encountering code at risk of throwing a `NullReferenceException`.

It can sometimes be useful to separate these two concepts and enable *just* the annotation context, or (less usefully) *just* the warning context:

```
#nullable enable annotations    // Enable the annotation context
// OR:
#nullable enable warnings       // Enable the warning context
```

(The same trick works with `#nullable disable` and `#nullable restore`.)

You can also do it via the project file:

```
<Nullable>annotations</Nullable>
<!-- OR -->
<Nullable>warnings</Nullable>
```

Enabling just the annotation context for a particular class or assembly can be a good first step in introducing nullable reference types into a legacy codebase. By correctly annotating public members, your class or assembly can act as a *good citizen* to other classes or assemblies—so that *they* can benefit fully from nullable reference types—without having to deal with warnings in your own class or assembly.

Treating Nullable Warnings as Errors

In greenfield projects, it makes sense to fully enable the nullable context from the outset. You might want to take the additional step of treating nullable warnings as errors so that your project cannot compile until all null-warnings have been resolved:

```
<PropertyGroup>
  <Nullable>enable</Nullable>
  <WarningsAsErrors>CS8600;CS8602;CS8603</WarningsAsErrors>
</PropertyGroup>
```

Extension Methods

Extension methods allow an existing type to be extended with new methods without altering the definition of the original type. An extension method is a static method of a static class, where the `this` modifier is applied to the first parameter. The type of the first parameter will be the type that is extended:

```
public static class StringHelper
{
  public static bool IsCapitalized (this string s)
  {
    if (string.IsNullOrEmpty(s)) return false;
    return char.IsUpper (s[0]);
  }
}
```

The `IsCapitalized` extension method can be called as though it were an instance method on a string, as follows:

```
Console.WriteLine ("Perth".IsCapitalized());
```

An extension method call, when compiled, is translated back into an ordinary static method call:

```
Console.WriteLine (StringHelper.IsCapitalized ("Perth"));
```

The translation works as follows:

```
arg0.Method (arg1, arg2, ...);              // Extension method call
StaticClass.Method (arg0, arg1, arg2, ...); // Static method call
```

Interfaces can be extended, too:

```
public static T First<T> (this IEnumerable<T> sequence)
{
  foreach (T element in sequence)
    return element;

  throw new InvalidOperationException ("No elements!");
}
...
Console.WriteLine ("Seattle".First());   // S
```

Extension Method Chaining

Extension methods, like instance methods, provide a tidy way to chain functions. Consider the following two functions:

```
public static class StringHelper
```

```
{
  public static string Pluralize (this string s) {...}
  public static string Capitalize (this string s) {...}
}
```

`x` and `y` are equivalent, and both evaluate to `"Sausages"`, but `x` uses extension methods, whereas `y` uses static methods:

```
string x = "sausage".Pluralize().Capitalize();
string y = StringHelper.Capitalize (StringHelper.Pluralize ("sausage"));
```

Ambiguity and Resolution

NAMESPACES

An extension method cannot be accessed unless its class is in scope, typically by its namespace being imported. Consider the extension method `IsCapitalized` in the following example:

```
using System;

namespace Utils
{
  public static class StringHelper
  {
    public static bool IsCapitalized (this string s)
    {
      if (string.IsNullOrEmpty(s)) return false;
      return char.IsUpper (s[0]);
    }
  }
}
```

To use `IsCapitalized`, the following application must import `Utils` in order to avoid a compile-time error:

```
namespace MyApp
{
  using Utils;

  class Test
  {
    static void Main() => Console.WriteLine ("Perth".IsCapitalized());
  }
}
```

EXTENSION METHODS VERSUS INSTANCE METHODS

Any compatible instance method will always take precedence over an extension method. In the following example, `Test`'s `Foo` method will always take precedence, even when called with an argument `x` of type `int`:

```
class Test
{
  public void Foo (object x) { }    // This method always wins
}

static class Extensions
{
  public static void Foo (this Test t, int x) { }
}
```

The only way to call the extension method in this case is via normal static syntax; in other words, `Extensions.Foo(...)`.

EXTENSION METHODS VERSUS EXTENSION METHODS

If two extension methods have the same signature, the extension method must be called as an ordinary static method to disambiguate the method to call. If one extension method has more specific arguments, however, the more specific method takes precedence.

To illustrate, consider the following two classes:

```
static class StringHelper
{
  public static bool IsCapitalized (this string s) {...}
}
static class ObjectHelper
{
  public static bool IsCapitalized (this object s) {...}
}
```

The following code calls `StringHelper`'s `IsCapitalized` method:

```
bool test1 = "Perth".IsCapitalized();
```

Classes and structs are considered more specific than interfaces.

Anonymous Types

An anonymous type is a simple class created by the compiler on the fly to store a set of values. To create an anonymous type, use the `new` keyword followed by an object initializer, specifying the properties and values the type will contain; for example:

```
var dude = new { Name = "Bob", Age = 23 };
```

The compiler translates this to (approximately) the following:

```
internal class AnonymousGeneratedTypeName
{
  private string name;  // Actual field name is irrelevant
  private int    age;   // Actual field name is irrelevant

  public AnonymousGeneratedTypeName (string name, int age)
  {
    this.name = name; this.age = age;
  }

  public string  Name { get { return name; } }
  public int     Age  { get { return age;  } }

  // The Equals and GetHashCode methods are overridden (see Chapter 6).
  // The ToString method is also overridden.
}
...

var dude = new AnonymousGeneratedTypeName ("Bob", 23);
```

You must use the `var` keyword to reference an anonymous type because it doesn't have a name.

The property name of an anonymous type can be inferred from an expression that is itself an identifier (or ends with one); thus:

```
int Age = 23;
var dude = new { Name = "Bob", Age, Age.ToString().Length };
```

is equivalent to:

```
var dude = new { Name = "Bob", Age = Age, Length =
Age.ToString().Length };
```

Two anonymous type instances declared within the same assembly will have the same underlying type if their elements are named and typed identically:

```
var a1 = new { X = 2, Y = 4 };
var a2 = new { X = 2, Y = 4 };
Console.WriteLine (a1.GetType() == a2.GetType());   // True
```

Additionally, the `Equals` method is overridden to perform equality comparisons:

```
Console.WriteLine (a1 == a2);         // False
Console.WriteLine (a1.Equals (a2));   // True
```

You can create arrays of anonymous types, as follows:

```
var dudes = new[]
{
  new { Name = "Bob", Age = 30 },
  new { Name = "Tom", Age = 40 }
};
```

A method cannot (usefully) return an anonymously typed object, because it is illegal to write a method whose return type is `var`:

```
var Foo() => new { Name = "Bob", Age = 30 };  // Not legal!
```

Instead, you must use `object` or `dynamic`, and then whoever calls `Foo` must rely on dynamic binding, with loss of static type safety (and IntelliSense in Visual Studio).

```
dynamic Foo() => new { Name = "Bob", Age = 30 };  // No static type
safety.
```

Anonymous types are particularly useful when writing LINQ queries (see Chapter 8).

Tuples

Like anonymous types, tuples provide a simple way to store a set of values. The main purpose of tuples is to safely return multiple values from a method without resorting to `out` parameters (something you cannot do with anonymous types).

> **NOTE**
>
> Tuples do almost everything that anonymous types do and more. Their one disadvantage—as you'll see soon—is runtime type erasure with named elements.

The simplest way to create a *tuple literal* is to list the desired values in parentheses. This creates a tuple with *unnamed* elements, which you refer to as `Item1`, `Item2`, and so on:

```
var bob = ("Bob", 23);    // Allow compiler to infer the element
types

Console.WriteLine (bob.Item1);   // Bob
Console.WriteLine (bob.Item2);   // 23
```

Tuples are *value types*, with *mutable* (read/write) elements:

```
var joe = bob;                // joe is a *copy* of bob
joe.Item1 = "Joe";            // Change joe's Item1 from Bob to Joe
Console.WriteLine (bob);      // (Bob, 23)
Console.WriteLine (joe);      // (Joe, 23)
```

Unlike with anonymous types, you can specify a *tuple type* explicitly. Just list each of the element types in parentheses:

```
(string,int) bob  = ("Bob", 23);
```

This means that you can usefully return a tuple from a method:

```
static (string,int) GetPerson() => ("Bob", 23);

static void Main()
{
  (string,int) person = GetPerson();  // Could use 'var' instead if
we want
  Console.WriteLine (person.Item1);   // Bob
  Console.WriteLine (person.Item2);   // 23
}
```

Tuples play well with generics, so the following types are all legal:

```
Task<(string,int)>
Dictionary<(string,int),Uri>
IEnumerable<(int id, string name)>   // See below for naming elements
```

Naming Tuple Elements

You can optionally give meaningful names to elements when creating tuple literals:

```
var tuple = (name:"Bob", age:23);

Console.WriteLine (tuple.name);     // Bob
Console.WriteLine (tuple.age);      // 23
```

You can do the same when specifying *tuple types*:

```
static (string name, int age) GetPerson() => ("Bob", 23);

static void Main()
{
  var person = GetPerson();
  Console.WriteLine (person.name);    // Bob
  Console.WriteLine (person.age);     // 23
}
```

Note that you can still treat the elements as unnamed and refer to them as `Item1`, `Item2`, etc. (although Visual Studio hides these fields from IntelliSense).

Element names are automatically *inferred* from property or field names:

```
var now = DateTime.Now;
var tuple = (now.Day, now.Month, now.Year);
Console.WriteLine (tuple.Day);               // OK
```

Tuples are type compatible with one another if their element types match up (in order). Their element names need not:

```
(string name, int age, char sex)  bob1 = ("Bob", 23, 'M');
(string age,  int sex, char name) bob2 = bob1;   // No error!
```

Our particular example leads to confusing results:

```
Console.WriteLine (bob2.name);    // M
Console.WriteLine (bob2.age);     // Bob
Console.WriteLine (bob2.sex);     // 23
```

TYPE ERASURE

We stated previously that the C# compiler handles anonymous types by building custom classes with named properties for each of the elements. With tuples, C# works differently and uses a preexisting family of generic structs:

```
public struct ValueTuple<T1>
public struct ValueTuple<T1,T2>
public struct ValueTuple<T1,T2,T3>
...
```

Each of the `ValueTuple<>` structs has fields named `Item1`, `Item2`, and so on.

Hence, `(string,int)` is an alias for `ValueTuple<string,int>`, and this means that named tuple elements have no corresponding property names in the underlying types. Instead, the names exist only in the source code, and in the imagination of the compiler. At runtime, the names mostly disappear, so if you decompile a program that refers to named tuple elements, you'll see just references to `Item1`, `Item2`, and so on. Further, when you examine a tuple variable in a debugger after having assigned it to an `object` (or Dump it in LINQPad), the element names are not there. And for the most part, you cannot use *reflection* (Chapter 19) to determine a tuple's element names at runtime.

> **NOTE**
>
> We said that the names *mostly* disappear because there's an exception. With methods/properties that return named tuple types, the compiler emits the element names by applying a custom attribute called `TupleElementNamesAttribute` (see "Attributes") to the member's return type. This allows named elements to work when calling methods in a different assembly (for which the compiler does not have the source code).

ValueTuple.Create

You can also create tuples via a factory method on the (nongeneric) `ValueTuple` type:

```
ValueTuple<string,int> bob1 = ValueTuple.Create ("Bob", 23);
(string,int)           bob2 = ValueTuple.Create ("Bob", 23);
```

You cannot create named elements in this way, because element naming relies on compiler magic.

Deconstructing Tuples

Tuples implicitly support the deconstruction pattern (see "Deconstructors" in Chapter 3), so you can easily *deconstruct* a tuple into individual variables. So, instead of doing this:

```
var bob = ("Bob", 23);

string name = bob.Item1;
int age = bob.Item2;
```

you can do this:

```
var bob = ("Bob", 23);

(string name, int age) = bob;   // Deconstruct the bob tuple
into
                           // separate variables (name and
age).
Console.WriteLine (name);
Console.WriteLine (age);
```

The syntax for deconstruction is confusingly similar to the syntax for declaring a tuple with named elements. The following highlights the difference:

```
(string name, int age)      = bob;   // Deconstructing a tuple
(string name, int age) bob2 = bob;   // Declaring a new tuple
```

Here's another example, this time when calling a method, and with type inference (`var`):

```
static (string, int, char) GetBob() => ( "Bob", 23, 'M');

static void Main()
{
  var (name, age, sex) = GetBob();
  Console.WriteLine (name);        // Bob
  Console.WriteLine (age);         // 23
  Console.WriteLine (sex);         // M
}
```

Equality Comparison

As with anonymous types, the `ValueTuple<>` types override the `Equals` method to allow equality comparisons to work meaningfully:

```
var t1 = ("one", 1);
var t2 = ("one", 1);
Console.WriteLine (t1.Equals (t2));    // True
```

In addition, `ValueTuple<>` overloads the `==` and `!=` operators:

```
Console.WriteLine (t1 == t2);    // True (from C# 7.3)
```

They also override the `GetHashCode` method, making it practical to use tuples as keys in dictionaries. We cover equality comparison in detail in "Equality Comparison" in Chapter 6, and "Dictionaries" in Chapter 7.

The `ValueTuple<>` types also implement `IComparable` (see "Order Comparison" in Chapter 6), making it possible to use tuples as a sorting key.

The System.Tuple classes

You'll find another family of generic types in the `System` namespace called `Tuple` (rather than `ValueTuple`). These were introduced in .NET Framework 4.0 and are classes (whereas the `ValueTuple` types are structs). Defining tuples as classes was in retrospect considered a mistake: in the typical scenarios in which tuples are used, structs have a slight performance advantage (in that they avoid unnecessary memory allocations), with almost no downside. Hence, when Microsoft added language support for tuples (in C# 7), it ignored the existing `Tuple` types in favor of the new `ValueTuple`. You might still come across the `Tuple` classes in code written prior to C# 7. They have no special language support and are used as follows:

```
Tuple<string,int> t = Tuple.Create ("Bob", 23);  // Factory method
Console.WriteLine (t.Item1);       // Bob
Console.WriteLine (t.Item2);       // 23
```

Patterns

In Chapter 3, we demonstrated how to use the `is` operator to test whether a reference conversion will succeed:

```
if (obj is string)
  Console.WriteLine (((string)obj).Length);
```

Or, more concisely:

```
if (obj is string s)
  Console.WriteLine (s.Length);
```

This employs one kind of pattern called a *type pattern*. The `is` operator also supports other patterns that were introduced in C# 7 and C# 8, such as the *property pattern*:

```
if (obj is string { Length:4 })
  Console.WriteLine ("A string with 4 characters");
```

Patterns are supported in the following contexts:

- After the `is` operator (*`variable` `is` `pattern`*)
- In `switch` statements
- In `switch` expressions

We've already covered the type pattern (and briefly, the tuple pattern) in "Switching on types" in Chapter 2, and "The is operator" in Chapter 3. In this section, we cover more advanced patterns that were introduced in C# 7 and C# 8. Most of these patterns are intended for use in `switch` statements/expressions, where they do the following:

- Reduce the need for `when` clauses
- Let you use switches where you couldn't previously

> **NOTE**
>
> The patterns in this section are mild-to-moderately useful in some scenarios. Remember that you can always replace highly patterned `switch` expressions with simple `if` statements—or in some cases, the ternary conditional operator—and often without much extra code.

Property Patterns (C# 8)

A property pattern matches on one or more of an object's property values. We gave a simple example previously in the context of the `is` operator:

```
if (obj is string { Length:4 }) ...
```

However, this doesn't save much over the following:

```
if (obj is string s && s.Length == 4) ...
```

With `switch` statements and expressions, property patterns are more useful. Consider the `System.Uri` class, which represents a URI. It has properties that include `Scheme`, `Host`, `Port`, and `IsLoopback`. In writing a firewall, we could decide whether to allow or block a URI by employing a `switch` expression that uses property patterns:

```
bool ShouldAllow (Uri uri) => uri switch
{
  { Scheme: "http",  Port: 80  } => true,
  { Scheme: "https", Port: 443 } => true,
```

```
  { Scheme: "ftp",   Port: 21  } => true,
  { IsLoopback: true           } => true,
  _ => false
};
```

You can nest properties, making the following clause legal:

```
  { Scheme: string { Length: 4 }, Port: 80 } => true,
```

Matching is always based on type and equality. Should you need to apply some other operator (such as less-than), you must use a `when` clause:

```
  { Scheme: "http",  Port: 80  } when uri.Host.Length < 1000 =>
true,
```

You can combine the type pattern with the property pattern:

```
bool ShouldAllow (object uri) => uri switch
{
  Uri { Scheme: "http",  Port: 80  } => true,
  Uri { Scheme: "https", Port: 443 } => true,
  ...
```

As you might expect with type patterns, you can introduce a variable at the end of a clause and then consume that variable:

```
  Uri { Scheme: "http", Port: 80 } httpUri => httpUri.Host.Length <
1000,
```

You can also use that variable in a `when` clause:

```
  Uri { Scheme: "http", Port: 80 } httpUri
                                  when httpUri.Host.Length < 1000 =>
true,
```

A somewhat bizarre twist with property patterns is that you can also introduce variables at the *property* level:

```
  { Scheme: "http", Port: 80, Host: string host } => host.Length <
1000,
```

Implicit typing is permitted, so you can substitute `string` with `var`. Here's a complete example:

```
bool ShouldAllow (Uri uri) => uri switch
{
  { Scheme: "http",  Port: 80, Host: var host } => host.Length <
1000,
  { Scheme: "https", Port: 443               } => true,
  { Scheme: "ftp",   Port: 21                } => true,
  { IsLoopback: true                         } => true,
  _ => false
};
```

It's difficult to invent examples for which this saves more than a few characters. In our case, the alternative is actually shorter:

```
  { Scheme: "http",  Port: 80 } => uri.Host.Length < 1000,
```

Tuple Patterns (C# 8)

Tuple patterns provide a simple mechanism for switching on multiple values:

```
enum Season { Spring, Summer, Fall, Winter };

int AverageCelsiusTemperature (Season season, bool daytime) =>
  (season, daytime) switch
  {
    (Season.Spring, true) => 20,
    (Season.Spring, false) => 16,
    (Season.Summer, true) => 27,
    (Season.Summer, false) => 22,
    (Season.Fall, true) => 18,
    (Season.Fall, false) => 12,
    (Season.Winter, true) => 10,
    (Season.Winter, false) => -2,
    _ => throw new Exception ("Unexpected combination")
};
```

Positional Patterns (C# 8)

For types that define a `Deconstruct` method (see "Deconstructors" in Chapter 3), such as the `Point` class in the following example:

```
class Point
{
  public readonly int X, Y;
  public Point (int x, int y) => (X, Y) = (x, y);
  public void Deconstruct (out int x, out int y)
  {
    x = X; y = Y;
  }
}
```

you can use the object's positional properties for pattern matching:

```
var p = new Point (2, 3);
Console.WriteLine (p is (2, 3));   // true
```

With a switch:

```
string Print (object obj) => obj switch
{
  Point (0, 0)                      => "Empty point",
  Point (var x, var y) when x == y  => "Diagonal"
  ...
};
```

var Pattern

The `var` pattern was introduced in C# 7 and is a variation of the type pattern whereby you replace the type name with the `var` keyword. The conversion always succeeds, so its purpose is merely to let you reuse the variable that follows:

```
bool Test (int x, int y) =>
   x * y is var product && product > 10 && product < 100;
```

Without this feature, you'd need to do this:

```
bool Test (int x, int y)
{
  int product = x * y;
  return product > 10 && product < 100;
}
```

The ability to introduce and reuse an intermediate variable (`product`, in this case) in an expression-bodied method is convenient. Unfortunately, it works only when the method in question has a `bool` return type.

Constant Pattern

The *constant pattern* is the bread and butter of `switch` statements (and until C# 7, it was the *only* supported pattern). For consistency, you also can use the constant pattern with the `is` operator from C# 7, making the following legal:

```
void Foo (object obj)
{
  // C# won't let you use the == operator, because obj is object.
  // However, we can use 'is'
  if (obj is 3) ...
}
```

This is equivalent to the following:

```
void Foo (object obj)
{
  if (obj is int && (int)obj == 3) ...
}
```

Attributes

You’re already familiar with the notion of attributing code elements of a program with modifiers, such as `virtual` or `ref`. These constructs are built into the language. *Attributes* are an extensible

mechanism for adding custom information to code elements (assemblies, types, members, return values, parameters, and generic type parameters). This extensibility is useful for services that integrate deeply into the type system, without requiring special keywords or constructs in the C# language.

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

Attribute Classes

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

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

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

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

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

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

The C# language and .NET Core include a number of predefined attributes. We describe how to write your own attributes in Chapter 19.

Named and Positional Attribute Parameters

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

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

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

Named parameters correspond to public fields or public properties on the attribute type.

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

In Chapter 19, we describe the valid parameter types and rules for their evaluation.

Applying Attributes to Assemblies and Backing Fields

Implicitly, the target of an attribute is the code element it immediately precedes, which is typically a type or type member. You can also attach attributes, however, to an assembly. This requires that you explicitly specify the attribute's target. Here is how you can use the `AssemblyFileVersion` attribute to attach a version to the assembly:

```
[assembly: AssemblyFileVersion ("1.2.3.4")]
```

From C# 7.3, you can use the `field:` prefix to apply an attribute to the backing fields of an automatic property. This can be useful in controlling serialization:

```
[field:NonSerialized]
public int MyProperty { get; set; }
```

Specifying Multiple Attributes

You can specify multiple attributes for a single code element. You can list each attribute either within the same pair of square brackets (separated by a comma) or in separate pairs of square brackets (or a combination of the two). The following three examples are semantically identical:

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

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

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

Caller Info Attributes

You can tag optional parameters with one of three *caller info attributes*, which instruct the compiler to feed information obtained from the caller's source code into the parameter's default value:

- `[CallerMemberName]` applies the caller's member name
- `[CallerFilePath]` applies the path to the caller's source code file
- `[CallerLineNumber]` applies the line number in the caller's source code file

The `Foo` method in the following program demonstrates all three:

```
using System;
using System.Runtime.CompilerServices;

class Program
{
  static void Main() => Foo();

  static void Foo (
    [CallerMemberName] string memberName = null,
    [CallerFilePath] string filePath = null,
    [CallerLineNumber] int lineNumber = 0)
  {
    Console.WriteLine (memberName);
    Console.WriteLine (filePath);
    Console.WriteLine (lineNumber);
  }
}
```

Assuming that our program resides in *c:\source\test\Program.cs*, the output would be:

```
Main
c:\source\test\Program.cs
6
```

As with standard optional parameters, the substitution is done at the *calling site*. Hence, our `Main` method is syntactic sugar for this:

```
static void Main() => Foo ("Main", @"c:\source\test\Program.cs", 6);
```

Caller info attributes are useful for logging—and for implementing patterns such as firing a single change notification event whenever

any property on an object changes. In fact, there's a standard interface in .NET Core for this called `INotifyPropertyChanged` (in `System.ComponentModel`):

```
public interface INotifyPropertyChanged
{
  event PropertyChangedEventHandler PropertyChanged;
}

public delegate void PropertyChangedEventHandler
  (object sender, PropertyChangedEventArgs e);

public class PropertyChangedEventArgs : EventArgs
{
  public PropertyChangedEventArgs (string propertyName);
  public virtual string PropertyName { get; }
}
```

Notice that `PropertyChangedEventArgs` requires the name of the property that changed. By applying the `[CallerMemberName]` attribute, however, we can implement this interface and invoke the event without ever specifying property names:

```
public class Foo : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged = delegate {
};

  void RaisePropertyChanged ([CallerMemberName] string propertyName
= null)
  {
    PropertyChanged (this, new PropertyChangedEventArgs (propertyName));
  }
```

```
  string customerName;
  public string CustomerName
  {
    get { return customerName; }
    set
    {
      if (value == customerName) return;
      customerName = value;
      RaisePropertyChanged();
      // The compiler converts the above line to:
      // RaisePropertyChanged ("CustomerName");
    }
  }
}
```

Dynamic Binding

Dynamic binding defers *binding*—the process of resolving types, members, and operators—from compile time to runtime. Dynamic binding is useful when at compile time *you* know that a certain function, member, or operation exists, but the *compiler* does not. This commonly occurs when you are interoperating with dynamic languages (such as IronPython) and COM, as well as for scenarios in which you might otherwise use reflection.

A dynamic type is declared with the contextual keyword `dynamic`:

```
dynamic d = GetSomeObject();
d.Quack();
```

A dynamic type tells the compiler to relax. We expect the runtime type of `d` to have a `Quack` method. We just can't prove it statically.

Because `d` is dynamic, the compiler defers binding `Quack` to `d` until runtime. To understand what this means requires distinguishing between *static binding* and *dynamic binding*.

Static Binding versus Dynamic Binding

The canonical binding example is mapping a name to a specific function when compiling an expression. To compile the following expression, the compiler needs to find the implementation of the method named `Quack`:

```
d.Quack();
```

Let's suppose that the static type of `d` is `Duck`:

```
Duck d = ...
d.Quack();
```

In the simplest case, the compiler does the binding by looking for a parameterless method named `Quack` on `Duck`. Failing that, the compiler extends its search to methods taking optional parameters, methods on base classes of `Duck`, and extension methods that take `Duck` as its first parameter. If no match is found, you'll get a compilation error. Regardless of what method is bound, the bottom line is that the binding is done by the compiler, and the binding utterly depends on statically knowing the types of the operands (in this case, `d`). This makes it *static binding*.

Now let's change the static type of `d` to `object`:

```
object d = ...
d.Quack();
```

Calling `Quack` gives us a compilation error, because although the value stored in `d` can contain a method called `Quack`, the compiler cannot know it, because the only information it has is the type of the variable, which in this case is `object`. But let's now change the static type of `d` to `dynamic`:

```
dynamic d = ...
d.Quack();
```

A `dynamic` type is like `object`—it's equally nondescriptive about a type. The difference is that it lets you use it in ways that aren't known at compile time. A dynamic object binds at runtime based on its runtime type, not its compile-time type. When the compiler sees a dynamically bound expression (which in general is an expression that contains any value of type `dynamic`), it merely packages up the expression such that the binding can be done later at runtime.

At runtime, if a dynamic object implements `IDynamicMetaObjectProvider`, that interface is used to perform the binding. If not, binding occurs in almost the same way as it would have had the compiler known the dynamic object's runtime type. These two alternatives are called *custom binding* and *language binding*.

Custom Binding

Custom binding occurs when a dynamic object implements `IDynamicMetaObjectProvider` (IDMOP). Although you can implement IDMOP on types that you write in C#, and that is useful to do, the more common case is that you have acquired an IDMOP object from a dynamic language that is implemented in .NET on the Dynamic Language Runtime (DLR), such as IronPython or IronRuby. Objects from those languages implicitly implement IDMOP as a means by which to directly control the meanings of operations performed on them.

We discuss custom binders in greater detail in Chapter 20, but for now, let's write a simple one to demonstrate the feature:

```
using System;
using System.Dynamic;

public class Test
{
  static void Main()
  {
    dynamic d = new Duck();
    d.Quack();                  // Quack method was called
    d.Waddle();                 // Waddle method was called
  }
}

public class Duck : DynamicObject
{
  public override bool TryInvokeMember (
    InvokeMemberBinder binder, object[] args, out object result)
  {
    Console.WriteLine (binder.Name + " method was called");
    result = null;
    return true;
```

```
  }
}
```

The `Duck` class doesn't actually have a `Quack` method. Instead, it uses custom binding to intercept and interpret all method calls.

Language Binding

Language binding occurs when a dynamic object does not implement IDMOP. Language binding is useful when working around imperfectly designed types or inherent limitations in the .NET type system (we explore more scenarios in Chapter 20). A typical problem when using numeric types is that they have no common interface. We have seen that we can bind methods dynamically; the same is true for operators:

```
static dynamic Mean (dynamic x, dynamic y) => (x + y) / 2;

static void Main()
{
  int x = 3, y = 4;
  Console.WriteLine (Mean (x, y));
}
```

The benefit is obvious—you don't need to duplicate code for each numeric type. However, you lose static type safety, risking runtime exceptions rather than compile-time errors.

> **NOTE**
>
> Dynamic binding circumvents static type safety, but not runtime type safety. Unlike with reflection (Chapter 19), you can't circumvent member accessibility rules with dynamic binding.

By design, language runtime binding behaves as similarly as possible to static binding, had the runtime types of the dynamic objects been known at compile time. In our previous example, the behavior of our program would be identical if we hardcoded `Mean` to work with the `int` type. The most notable exception in parity between static and dynamic binding is for extension methods, which we discuss in "Uncallable Functions".

> **NOTE**
>
> Dynamic binding also incurs a performance hit. Because of the DLR's caching mechanisms, however, repeated calls to the same dynamic expression are optimized—allowing you to efficiently call dynamic expressions in a loop. This optimization brings the typical overhead for a simple dynamic expression on today's hardware down to less than 100 nanoseconds.

RuntimeBinderException

If a member fails to bind, a `RuntimeBinderException` is thrown. You can think of this like a compile-time error at runtime:

```
dynamic d = 5;
d.Hello();                  // throws RuntimeBinderException
```

The exception is thrown because the `int` type has no `Hello` method.

Runtime Representation of dynamic

There is a deep equivalence between the `dynamic` and `object` types. The runtime treats the following expression as `true`:

```
typeof (dynamic) == typeof (object)
```

This principle extends to constructed types and array types:

```
typeof (List<dynamic>) == typeof (List<object>)
typeof (dynamic[]) == typeof (object[])
```

Like an object reference, a dynamic reference can point to an object of any type (except pointer types):

```
dynamic x = "hello";
Console.WriteLine (x.GetType().Name);  // String

x = 123;  // No error (despite same variable)
Console.WriteLine (x.GetType().Name);  // Int32
```

Structurally, there is no difference between an object reference and a dynamic reference. A dynamic reference simply enables dynamic operations on the object it points to. You can convert from `object` to `dynamic` to perform any dynamic operation you want on an `object`:

```
object o = new System.Text.StringBuilder();
dynamic d = o;
```

```
d.Append ("hello");
Console.WriteLine (o);   // hello
```

NOTE

Reflecting on a type exposing (public) `dynamic` members reveals that those members are represented as annotated `objects`; for example:

```
public class Test
{
  public dynamic Foo;
}
```

is equivalent to:

```
public class Test
{
  [System.Runtime.CompilerServices.DynamicAttribute]
  public object Foo;
}
```

This allows consumers of that type to know that `Foo` should be treated as dynamic while allowing languages that don't support dynamic binding to fall back to `object`.

Dynamic Conversions

The `dynamic` type has implicit conversions to and from all other types:

```
int i = 7;
```

```
dynamic d = i;
long j = d;        // No cast required (implicit conversion)
```

For the conversion to succeed, the runtime type of the dynamic object must be implicitly convertible to the target static type. The preceding example worked because an `int` is implicitly convertible to a `long`.

The following example throws a `RuntimeBinderException` because an `int` is not implicitly convertible to a `short`:

```
int i = 7;
dynamic d = i;
short j = d;      // throws RuntimeBinderException
```

var Versus dynamic

The `var` and `dynamic` types bear a superficial resemblance, but the difference is deep:

- `var` says, "Let the *compiler* figure out the type."
- `dynamic` says, "Let the *runtime* figure out the type."

To illustrate:

```
dynamic x = "hello";  // Static type is dynamic, runtime type is
string
var y = "hello";      // Static type is string, runtime type is string
int i = x;            // Runtime error      (cannot convert string to
int)
int j = y;            // Compile-time error (cannot convert string to
int)
```

The static type of a variable declared with `var` can be `dynamic`:

```
dynamic x = "hello";
var y = x;            // Static type of y is dynamic
int z = y;            // Runtime error (cannot convert string to int)
```

Dynamic Expressions

Fields, properties, methods, events, constructors, indexers, operators, and conversions can all be called dynamically.

Trying to consume the result of a dynamic expression with a `void` return type is prohibited—just as with a statically typed expression. The difference is that the error occurs at runtime:

```
dynamic list = new List<int>();
var result = list.Add (5);         // RuntimeBinderException thrown
```

Expressions involving dynamic operands are typically themselves dynamic because the effect of absent type information is cascading:

```
dynamic x = 2;
var y = x * 3;       // Static type of y is dynamic
```

There are a couple of obvious exceptions to this rule. First, casting a dynamic expression to a static type yields a static expression:

```
dynamic x = 2;
var y = (int)x;      // Static type of y is int
```

Second, constructor invocations always yield static expressions—even when called with dynamic arguments. In this example, `x` is statically typed to a `StringBuilder`:

```
dynamic capacity = 10;
var x = new System.Text.StringBuilder (capacity);
```

In addition, there are a few edge cases for which an expression containing a dynamic argument is static, including passing an index to an array and delegate creation expressions.

Dynamic Calls Without Dynamic Receivers

The canonical use case for `dynamic` involves a dynamic *receiver*. This means that a dynamic object is the receiver of a dynamic function call:

```
dynamic x = ...;
x.Foo();          // x is the receiver
```

However, you can also call statically known functions with dynamic arguments. Such calls are subject to dynamic overload resolution, and can include the following:

- Static methods
- Instance constructors
- Instance methods on receivers with a statically known type

In the following example, the particular `Foo` that gets dynamically bound is dependent on the runtime type of the dynamic argument:

```
class Program
{
  static void Foo (int x)    => Console.WriteLine ("int");
  static void Foo (string x) => Console.WriteLine ("string");

  static void Main()
  {
    dynamic x = 5;
    dynamic y = "watermelon";

    Foo (x);                // 1
    Foo (y);                // 2
  }
}
```

Because a dynamic receiver is not involved, the compiler can statically perform a basic check to see whether the dynamic call will succeed. It checks whether a function with the correct name and number of parameters exists. If no candidate is found, you get a compile-time error:

```
class Program
{
  static void Foo (int x)    => Console.WriteLine ("int");
  static void Foo (string x) => Console.WriteLine ("string");

  static void Main()
  {
    dynamic x = 5;
    Foo (x, x);          // Compiler error - wrong number of parameters
    Fook (x);            // Compiler error - no such method name
```

```
  }
}
```

Static Types in Dynamic Expressions

It's obvious that dynamic types are used in dynamic binding. It's not so obvious that static types are also used—wherever possible—in dynamic binding. Consider the following:

```
class Program
{
  static void Foo (object x, object y) { Console.WriteLine ("oo"); }
  static void Foo (object x, string y) { Console.WriteLine ("os"); }
  static void Foo (string x, object y) { Console.WriteLine ("so"); }
  static void Foo (string x, string y) { Console.WriteLine ("ss"); }

  static void Main()
  {
    object o = "hello";
    dynamic d = "goodbye";
    Foo (o, d);               // os
  }
}
```

The call to `Foo(o,d)` is dynamically bound because one of its arguments, `d`, is `dynamic`. But because `o` is statically known, the binding—even though it occurs dynamically—will make use of that. In this case, overload resolution will pick the second implementation of `Foo` due to the static type of `o` and the runtime type of `d`. In other words, the compiler is "as static as it can possibly be."

Uncallable Functions

Some functions cannot be called dynamically. You cannot call the following:

- Extension methods (via extension method syntax)
- Members of an interface, if you need to cast to that interface to do so
- Base members hidden by a subclass

Understanding why this is so is useful in understanding dynamic binding.

Dynamic binding requires two pieces of information: the name of the function to call, and the object upon which to call the function. However, in each of the three uncallable scenarios, an *additional type* is involved, which is known only at compile time. As of this writing, there's no way to specify these additional types dynamically.

When calling extension methods, that additional type is implicit. It's the static class on which the extension method is defined. The compiler searches for it given the `using` directives in your source code. This makes extension methods compile-time-only concepts because `using` directives melt away upon compilation (after they've done their job in the binding process in mapping simple names to namespace-qualified names).

When calling members via an interface, you specify that additional type via an implicit or explicit cast. There are two scenarios for which you might want to do this: when calling explicitly implemented

interface members, and when calling interface members implemented in a type internal to another assembly. We can illustrate the former with the following two types:

```
interface IFoo   { void Test();        }
class Foo : IFoo { void IFoo.Test() {} }
```

To call the `Test` method, we must cast to the `IFoo` interface. This is easy with static typing:

```
IFoo f = new Foo();   // Implicit cast to interface
f.Test();
```

Now consider the situation with dynamic typing:

```
IFoo f = new Foo();
dynamic d = f;
d.Test();             // Exception thrown
```

The implicit cast shown in bold tells the *compiler* to bind subsequent member calls on `f` to `IFoo` rather than `Foo`—in other words, to view that object through the lens of the `IFoo` interface. However, that lens is lost at runtime, so the DLR cannot complete the binding. The loss is illustrated as follows:

```
Console.WriteLine (f.GetType().Name);    // Foo
```

A similar situation arises when calling a hidden base member: you must specify an additional type via either a cast or the `base` keyword

—and that additional type is lost at runtime.

Operator Overloading

You can overload operators to provide more natural syntax for custom types. Operator overloading is most appropriately used for implementing custom structs that represent fairly primitive data types. For example, a custom numeric type is an excellent candidate for operator overloading.

The following symbolic operators can be overloaded:

+ (unary)	- (unary)	!	~	++
--	+	-	*	/
%	&	|	^	<<
>>	==	!=	>	<
>=	<=			

The following operators are also overloadable:

- Implicit and explicit conversions (with the `implicit` and `explicit` keywords)
- The `true` and `false` *operators* (not *literals*).

The following operators are indirectly overloaded:

- The compound assignment operators (e.g., `+=`, `/=`) are implicitly overridden by overriding the noncompound operators (e.g., `+`, `/`).
- The conditional operators `&&` and `||` are implicitly overridden by overriding the bitwise operators `&` and `|`.

Operator Functions

You overload an operator by declaring an *operator function*. An operator function has the following rules:

- The name of the function is specified with the `operator` keyword followed by an operator symbol.
- The operator function must be marked `static` and `public`.
- The parameters of the operator function represent the operands.
- The return type of an operator function represents the result of an expression.
- At least one of the operands must be the type in which the operator function is declared.

In the following example, we define a struct called `Note` representing a musical note and then overload the `+` operator:

```
public struct Note
{
  int value;
  public Note (int semitonesFromA) { value = semitonesFromA; }
  public static Note operator + (Note x, int semitones)
  {
```

```
    return new Note (x.value + semitones);
  }
}
```

This overload allows us to add an `int` to a `Note`:

```
Note B = new Note (2);
Note CSharp = B + 2;
```

Overloading an operator automatically overloads the corresponding compound assignment operator. In our example, because we overrode +, we can use +=, too:

```
CSharp += 2;
```

Just as with methods and properties, C# allows operator functions comprising a single expression to be written more tersely with expression-bodied syntax:

```
public static Note operator + (Note x, int semitones)
                              => new Note (x.value + semitones);
```

Overloading Equality and Comparison Operators

Equality and comparison operators are sometimes overridden when writing structs, and in rare cases when writing classes. Special rules and obligations come with overloading the equality and comparison operators, which we explain in Chapter 6. A summary of these rules is as follows:

Pairing
: The C# compiler enforces operators that are logical pairs to both be defined. These operators are (== !=), (< >), and (<= >=).

`Equals` and `GetHashCode`
: In most cases, if you overload (==) and (!=), you must override the `Equals` and `GetHashCode` methods defined on `object` in order to get meaningful behavior. The C# compiler will give a warning if you do not do this. (See "Equality Comparison" in Chapter 6 for more details.)

`IComparable` and `IComparable<T>`
: If you overload (< >) and (<= >=), you should implement `IComparable` and `IComparable<T>`.

Custom Implicit and Explicit Conversions

Implicit and explicit conversions are overloadable operators. These conversions are typically overloaded to make converting between strongly related types (such as numeric types) concise and natural.

To convert between weakly related types, the following strategies are more suitable:

- Write a constructor that has a parameter of the type to convert from.
- Write `To`*`XXX`* and (static) `From`*`XXX`* methods to convert between types.

As explained in the discussion on types, the rationale behind implicit conversions is that they are guaranteed to succeed and not lose

information during the conversion. Conversely, an explicit conversion should be required either when runtime circumstances will determine whether the conversion will succeed, or if information might be lost during the conversion.

> **NOTE**
>
> Custom conversions are ignored by the `as` and `is` operators:
>
> ```
> Console.WriteLine (554.37 is Note);   // False
> Note n = 554.37 as Note;              // Error
> ```

In this example, we define conversions between our musical `Note` type and a `double` (which represents the frequency in hertz of that note):

```
...
// Convert to hertz
public static implicit operator double (Note x)
  => 440 * Math.Pow (2, (double) x.value / 12 );

// Convert from hertz (accurate to the nearest semitone)
public static explicit operator Note (double x)
  => new Note ((int) (0.5 + 12 * (Math.Log (x/440) / Math.Log(2) ) ));
...

Note n = (Note)554.37;  // explicit conversion
double x = n;           // implicit conversion
```

> **NOTE**
>
> Following our own guidelines, this example might be better implemented with a `ToFrequency` method (and a static `FromFrequency` method) instead of implicit and explicit operators.

Overloading true and false

The `true` and `false` operators are overloaded in the extremely rare case of types that are Boolean *in spirit*, but do not have a conversion to `bool`. An example is a type that implements three-state logic: by overloading `true` and `false`, such a type can work seamlessly with conditional statements and operators—namely, `if`, `do`, `while`, `for`, `&&`, `||`, and `?:`. The `System.Data.SqlTypes.SqlBoolean` struct provides this functionality:

```
SqlBoolean a = SqlBoolean.Null;
if (a)
  Console.WriteLine ("True");
else if (!a)
  Console.WriteLine ("False");
else
  Console.WriteLine ("Null");

OUTPUT:
Null
```

The following code is a reimplementation of the parts of `SqlBoolean` necessary to demonstrate the `true` and `false` operators:

```
public struct SqlBoolean
```

```
{
  public static bool operator true (SqlBoolean x)
    => x.m_value == True.m_value;

  public static bool operator false (SqlBoolean x)
    => x.m_value == False.m_value;

  public static SqlBoolean operator ! (SqlBoolean x)
  {
    if (x.m_value == Null.m_value)  return Null;
    if (x.m_value == False.m_value) return True;
    return False;
  }

  public static readonly SqlBoolean Null =  new SqlBoolean(0);
  public static readonly SqlBoolean False = new SqlBoolean(1);
  public static readonly SqlBoolean True =  new SqlBoolean(2);

  private SqlBoolean (byte value) { m_value = value; }
  private byte m_value;
}
```

Unsafe Code and Pointers

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

Pointer Basics

For every value type or reference type *V*, there is a corresponding pointer type *V**. A pointer instance holds the address of a variable.

Pointer types can be (unsafely) cast to any other pointer type. Following are the main pointer operators:

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

Unsafe Code

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

```
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
      *p++ &= 0xFF;
  }
}
```

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

The fixed Statement

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

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

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

```
class Test
{
```

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

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

The Pointer-to-Member Operator

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

```
struct Test
{
  int x;
  unsafe static void Main()
  {
    Test test = new Test();
    Test* p = &test;
    p->x = 9;
    System.Console.WriteLine (test.x);
  }
}
```

The stackalloc Keyword

You can allocate memory in a block on the stack explicitly by using the `stackalloc` keyword. Because it is allocated on the stack, its lifetime is limited to the execution of the method, just as with any other local variable (whose life hasn't been extended by virtue of being captured by a lambda expression, iterator block, or asynchronous function). The block can use the `[]` operator to index into memory:

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

In Chapter 24, we describe how you can use `Span<T>` to manage stack-allocated memory without using the `unsafe` keyword:

```
Span<int> a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
  Console.WriteLine (a[i]);
```

Fixed-Size Buffers

The `fixed` keyword has another use, which is to create fixed-size buffers within structs (this can be useful when calling an unmanaged function; see Chapter 24):

```
unsafe struct UnsafeUnicodeString
{
  public short Length;
  public fixed byte Buffer[30];   // Allocate block of 30 bytes
```

```
}

unsafe class UnsafeClass
{
  UnsafeUnicodeString uus;

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

Fixed-size buffers are not arrays: if `Buffer` was an array, it would consist of a reference to an object stored on the (managed) heap, rather than 30 bytes within the struct itself.

The `fixed` keyword is also used in this example to pin the object on the heap that contains the buffer (which will be the instance of `UnsafeClass`). Hence, `fixed` means two different things: fixed in *size*, and fixed in *place*. The two are often used together, in that a fixed-size buffer must be fixed in place to be used.

void*

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

`void*`. A `void*` cannot be dereferenced, and arithmetic operations cannot be performed on void pointers. Here's an example:

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

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

Pointers to Unmanaged Code

Pointers are also useful for accessing data outside the managed heap (such as when interacting with C Dynamic-Link Libraries [DLLs] or Component Object Model [COM]) or when dealing with data not in the main memory (such as graphics memory or a storage medium on an embedded device).

Preprocessor Directives

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

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

In this class, the statement in `Foo` is compiled as conditionally dependent upon the presence of the `DEBUG` symbol. If we remove the `DEBUG` symbol, the statement is not compiled. You can define preprocessor symbols within a source file (as we have done), or at a project level in the *.csproj* file:

```
<PropertyGroup>
  <DefineConstants>DEBUG;ANOTHERSYMBOL</DefineConstants>
</PropertyGroup>
```

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

follows if the `TESTMODE` symbol is defined and the `DEBUG` symbol is not defined:

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

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

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

Table 4-1. Preprocessor

c
e
s
s
o
r

d
i
r
e
c
t
i
v
e
s

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

`#elif symbol [operator symbol2]`	Combines `#else` branch and `#if` test
`#endif`	Ends conditional directives
`#warning text`	*`text`* of the warning to appear in compiler output
`#error text`	*`text`* of the error to appear in compiler output
`#pragma warning [disable | restore]`	Disables/restores compiler warning(s)
`#line [ number ["file"] | hidden]`	*`number`* specifies the line in source code; *`file`* is the filename to appear in computer output; `hidden` instructs debuggers to skip over code from this point until the next `#line` directive
`#region name`	Marks the beginning of an outline
`#endregion`	Ends an outline region
`#nullable option`	See "Nullable Reference Types (C# 8)"

Conditional Attributes

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

```
// file1.cs
#define DEBUG
using System;
using System.Diagnostics;
[Conditional("DEBUG")]
```

```
public class TestAttribute : Attribute {}

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

The compiler will incorporate the `[Test]` attributes only if the `DEBUG` symbol is in scope for *file2.cs*.

pragma warning

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

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

To this effect, the compiler allows you to selectively suppress warnings by using the `#pragma warning` directive. In this example, we instruct the compiler not to warn us about the field `Message` not being used:

```
public class Foo
{
```

```
  static void Main() { }

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

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

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

XML Documentation

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

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

You can do multiline comments either like this:

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

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

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

If you add the following option to your *.csproj* file:

```
<PropertyGroup>
  <DocumentationFile>SomeFile.xml</DocumentationFile>
</PropertyGroup>
```

the compiler extracts and collates documentation comments into the specified XML file. This has two main uses:

- If placed in the same folder as the compiled assembly, tools such as Visual Studio and LINQPad automatically read the XML file and use the information to provide IntelliSense member listings to consumers of the assembly of the same name.
- Third-party tools (such as Sandcastle and NDoc) can transform the XML file into an HTML help file.

Standard XML Documentation Tags

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

```
<summary>
```

```
<summary>...</summary>
```

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

<remarks>

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

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

<param>

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

Explains a parameter on a method.

<returns>

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

Explains the return value for a method.

<exception>

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

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

`<permission>`

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

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

`<example>`

```
<example>...</example>
```

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

`<c>`

```
<c>...</c>
```

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

`<code>`

```
<code>...</code>
```

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

`<see>`

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

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

`<seealso>`

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

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

`<paramref>`

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

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

`<list>`

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

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

`<para>`

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

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

`<include>`

```
<include file='filename' path='tagpath[@name="id"]'>...
</include>
```

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

User-Defined Tags

Little is special about the predefined XML tags recognized by the C# compiler, and you are free to define your own. The only special processing done by the compiler is on the `<param>` tag (in which it verifies the parameter name and that all the parameters on the method are documented) and the `cref` attribute (in which it verifies that the attribute refers to a real type or member and expands it to a fully qualified type or member ID). You can also use the `cref` attribute in your own tags; it is verified and expanded just as it is in the predefined `<exception>`, `<permission>`, `<see>`, and `<seealso>` tags.

Type or Member Cross-References

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

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

E	Event
!	Error

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

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

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

    /// P:NS.MyClass.aProperty
    short aProperty {get {...} set {...}}

    /// T:NS.MyClass.NestedType
    class NestedType {...};

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

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

    /// M:NS.MyClass.Z(System.Char[ ],System.Single[0:,0:])
    void Z(char[ ] p1, float[,] p2) {...}

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

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

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

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

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