```
Transformer t = Square;
```

is shorthand for:

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

and:

```
    t(3)
```

is shorthand for:

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

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

constructs such as C function pointers.

Writing Plug-in Methods with Delegates

A delegatevariable is assigned a method *dynamically*. 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, 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);
```

```
// Dynamically hook in Square
```

```
    foreach (int i in values)
      Console.Write (i + "  ");                // 1
    Console.Write (i + " ");                    // 1
  }
  static int Square (int x) { return x * x; }
}
```

```
4
9
```

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. For example:

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

```
d += SomeMethod2;
```

The last line is functionally the same as:

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

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

The - and -= operators remove the right delegate operand from the left delegate operand. For example:

```
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;
```

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

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

Delegates are *immutable*, so when you call += or -=, you're in fact creating a *new* delegate instance and assigning it to the ex-

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. In most scenarios in which multicast delegates are used, they have `void` return types, so this subtlety does not arise.

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 you wrote a routine that took a long time to execute. That routine could

Suppose you wrote a routine that took a long time to execute. That routine could regularly report progress to its caller by invoking a delegate. In this example, the HardWork routine 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:

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"
  {
    System.IO.File.WriteAllText ("progress.txt",
                                 percentComplete.ToString());
  }
}
```

Instance Versus Static Method Targets

When a delegate object is assigned to an *instance* method, the delegate object 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). For example:

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

class Test
{
  static void Main()
  {
```

```
  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

Generic Delegate Types

A delegate type may contain generic type parameters. For example:

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

Advanced C#

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

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);
    Util.Transform (values, Square);
    foreach (int i in values)
      Console.Write (i + "  ");
  }

  // Dynamically hook in Square

  // 1   4   9

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

The Func and Action Delegates

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 will cover 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>
delegate void Action <in T1, in T2>
... and so on, up to T16
```

```
();
```

```
();
(T arg);
(T1 arg1, T2 arg2);
```

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.

Delegates Versus Interfaces

A problem that can be solved with a delegate can also be solved with an interface. For instance, the following explains how to solve our filter problem using an ITransformer interface:

```
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) { return x * x; }
}
```

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

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

A delegate design may 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 may 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, since `Test` can implement `ITransformer` only once. This is quite cumbersome:

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

```
class Cuber : ITransformer
class Cuber : ITransformer
{
  public int Transform (int x) {return 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

Type compatibility

Delegate types are all incompatible with each other, even if their signatures are the same:

```
delegate void D1();
delegate void D2();
...
D1 d1 = Method1;
D2 d2 = d1;

// Compile-time error
```

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*

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 exactly 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);
  {
    StringAction sa = new StringAction (ActOnObject);
    sa ("hello");
  }

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

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 gets implicitly upcast to an object.

The standard event pattern is designed to help you leverage 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 two different delegates, one passing a MouseEventArgs and the other passing a KeyEventArgs.

Return type compatibility

If you call a method, you may get back a type that is more specific than what you asked for. This is ordinary polymorphic behavior. For exactly the same reason, the return type of a delegate can be less specific than the return type of its target method. This is called *covariance*. For example:

```
delegate object ObjectRetriever();
```

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

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

Generic delegate type parameter variance (C# 4.0)

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:

- 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) supports covariance:

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

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

allowing:

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

The following delegate (defined in the System namespace) supports contravariance:

```
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 each other*.

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

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

```
public class Broadcaster
{
  public event ProgressReporter Progress;
}
```

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

How Do Events Work on the Inside?

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

```
public class Broadcaster
{
  public event ProgressReporter Progress;
}
```

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

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

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

The **add** and **remove** keywords denote explicit *event accessors*—which act rather like property accessors. We'll 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.

shortcut for + and – followed by an assignment.

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
  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      if (PriceChanged != null)
        PriceChanged (price, value);
      price = value;
    }
  }
}
```

```
}
// Exit if nothing has changed
// If invocation list not empty,
// fire event.
```

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, in that subscribers could do the following things to interfere with each other:

- 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.

Standard Event Pattern

The .NET Framework defines a standard pattern for writing events. Its purpose is to provide consistency across both Framework and user code. At the core of the standard event pattern is System.EventArgs: a predefined Framework class with no to provide consistency across both Framework 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

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

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);
  }
```

```
    if (PriceChanged != null) PriceChanged (this, e);
  }
}
```

In multithreaded scenarios (Chapter 21), you need to assign the delegate to a temporary variable before testing and invoking it in order to be thread-safe:

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

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 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)
{
  if (PriceChanged != null) PriceChanged (this, e);
}

public decimal Price
```

```
  public decimal Price
  {
    get { return price; }
    set
    {
```

```
      if (price == value) return;
      OnPriceChanged (new PriceChangedEventArgs (price, value));
      price = value;
    }
  }
}
```

```
class Test
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)
{
```

```
protected virtual void OnPriceChanged (EventArgs e)
{
  if (PriceChanged != null) PriceChanged (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 private delegate field

A public pair of event accessor functions, 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—see Chapter 21). By defining event accessors ourselves, we instruct C# not to generate default field and accessor logic.

ter 21). By defining event accessors ourselves, we instruct C# not to generate default field and accessor logic.

Before C# 4.0, the compiler ensured thread safety by locking the containing type or instance around updating the delegate. This is a crude mechanism (as we'll see in "Thread Safety" on page 543 in Chapter 21), and it resulted in some people using explicit event accessors purely to work around this

Safety” on page 543 in Chapter 21), and it resulted in some people using explicit event accessors purely to work around this. The good news is that from C# 4.0, the compiler implements thread safety through a superior lock-free compare-and-swap algorithm.

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

●

●

●

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

When the class exposes a large number of events, where 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, since a dictionary will contain less storage overhead than dozens of null delegate field references.

When explicitly implementing an interface that declares an event.

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; }
  }
}
```

The add and remove parts of an event are compiled to add_*XXX*

The `add` and `remove` parts of an event are compiled to `add_XXX` and `remove_XXX` methods.

The `+=` and `-=` operations on an event are compiled to calls to the `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

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:

•

•

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” on page 361 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;
```

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

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

Internally, the compiler resolves lambda expressions of this type by writing a private method, and 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;
```

```
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;
```

Lambda expressions were introduced in C# 3.0.

Explicitly Specifying Lambda Parameter Types

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 expression:

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

The compiler uses *type inference* to infer that x is an int.

We could explicitly specify x's type as follows:

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

Capturing Outer Variables

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

```
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*.

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));
Console.WriteLine (multiplier (3));
```

```
// 30
```

Lambda expressions can themselves update captured variables:

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

```
// 0
// 1
// 1
// 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());
  Console.WriteLine (natural());
  Console.WriteLine (natural());
}
```

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

```
// 0
// 0
```

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 called, the class is instantiated and lifetime-bound to the delegate instance.

Capturing iteration variables

When you capture iteration variables in for and foreach statements, C# treats those iteration variables as though they 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`. 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[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 *inside* 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
```

This then causes the closure to capture a *different* variable on each iteration.

Anonymous Methods

Anonymous Methods

Anonymous methods are a C# 2.0 feature that has been subsumed largely by C# 3.0 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);
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;
=> x * x;
```

A unique feature of anonymous methods is that you can omit the parameter declaration entirely—even if the delegate expects them. 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:

```
    Clicked += delegate { Console.WriteLine ("clicked"); };  // No parameters
```

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

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 a `catch` *block*, a `finally` *block*, or both. The `catch` block executes when an error occurs in the `try` block. The `finally` block ex-

The try *block* must be followed by a catch *block*, a finally *block*, or both. The catch block executes when an error occurs in the try block. The finally block executes after execution leaves the try block (or if present, the catch block), to perform cleanup code, whether or not an error occurred.

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, by always executing no matter what. 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)
{
```

```
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) { return 10 / x; }
```

[redacted]

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

```
    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) { return 10 / x; }

  static void Main()
  {
    try
    {
      int y = Calc (0);
    {
      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
```

```
x cannot be zero
program completed
```

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`.

Exceptions are relatively expensive to handle, taking hundreds of clock cycles.

When an exception is thrown, the CLR performs a test:

Is execution currently within a `try` *statement that can catch the exception?*

●

●

If so, execution is passed to the compatible `catch` block. If the `catch` block successfully finishes executing, execution moves to the next statement after the `try` statement (if present, executing the `finally` block first).

try statement (if present, executing the finally block first).

If not, execution jumps back to the caller of the function, and the test is repeated (after executing any finally blocks that wrap the statement).

If no function takes responsibility for the exception, an error dialog is displayed to the user, and 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 when:

●
●
●

Your program can potentially recover regardless of the specific exception type.

You plan to rethrow the exception (perhaps after logging it).

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!");
}
```

```
        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 (StackOverflowException)   // no variable
{
   ...
}
```

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

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

In languages other than C#, it is possible (though not recommended) to throw an object that does not derive from `Exception`. The CLR automatically wraps that object in a `RuntimeWrappedException` class (which does derive from `Exception`).

Exception).

The finally Block

A finally block always executes—whether or not an exception is thrown and whether or not the try block runs to completion. finally blocks are typically used for cleanup code.

A finally block executes either:

-
-
-

After a catch block finishes

After control leaves the try block because of a jump statement (e.g., return or goto)

After the try block ends

After the try block ends

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
  {
```

```
  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 the .NET Framework and is supported explicitly in C# through the `using` statement.

The using statement

Many classes encapsulate unmanaged resources, such as file handles, graphics

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.

The following:

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

is precisely equivalent to:

```
StreamReader reader = File.OpenText ("file.txt");
try
{
  ...
}
finally
```

```
    }
    finally
    {
      if (reader != null)
       ((IDisposable)reader).Dispose();
    }
```

We cover the disposal pattern in more detail in Chapter 12.

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)
  {
```

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

    Console.WriteLine (name);
  }

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

Rethrowing an exception

You can capture and rethrow an exception as follows:

You can capture and rethrow an exception as follows:

```
try {  ...  }
catch (Exception ex)
{
  // Log error
  ...
  throw;              // Rethrow same exception
}
```

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 outside what you expected:

```
    using System.Net;       // (See Chapter 14)
    ...
```

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

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

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

Rethrowing an exception does not affect the StackTrace property of the exception (see the next section). When rethrowing a different exception, you can set the InnerException property with the original exception if doing so could aid debugging. Nearly all types of exceptions provide a constructor for this purpose.

Key Properties of System.Exception

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

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, may have another InnerException.

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 Framework. You can throw these yourself or use them as base classes for deriving

The following exception types are used widely throughout the CLR and .NET Framework. 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 where the underlying list has been modified partway through the iteration.

an unopened file or getting the next element from an enumerator where 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.

Common Patterns

The TryXXX method pattern

Ac

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 as follows:

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

The atomicity pattern

It can be desirable for an operation to be *atomic*, where it either successfully completes or fails without affecting state. An object becomes unusable when it enters an

indeterminate state that is the result of a half-finished operation. `finally` blocks facilitate writing atomic operations.

In the following example, we use an `Accumulator` class that has an `Add` method that adds an array of integers to its field `Total`. The `Add` method will cause an `OverflowException` if `Total` exceeds the maximum value for an `int`. The `Add` method is atomic, either successfully updating `Total` or failing, which leaves `Total` with its former value:

```
class Test
{
  static void Main()
  {
    Accumulator a = new Accumulator();
    try
    {
```

```
    {
      a.Add (4, 5);             // a.Total is now 9
      a.Add (1, int.MaxValue);  // Will cause OverflowException
    }
    catch (OverflowException)
    {
      Console.WriteLine (a.Total);  // a.Total is still 9
    }
  }
}
```

In the implementation of `Accumulator`, the `Add` method affects the `Total` field as it executes. However, if *anything goes wrong* during the method (e.g., a numeric overflow, a stack overflow, etc.), `Total` is restored to its initial value at the start of the method.

```
public class Accumulator
{
  public int Total { get; private set; }
public int Total { get; private set; }
public void Add (params int[] ints)
{
  bool success = false;
  int totalSnapshot = Total;
  try
  {
    foreach (int i in ints)
    {
      checked { Total += i; }
    }
    success = true;
  }
  finally
```

```
    }
    finally
    {
      if (! success)
        Total = totalSnapshot;
    }
  }
}
```

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 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 that 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*. An enumerator is an object that implements either of the following interfaces:

-
-

```
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

System.Collections.Generic.IEnumerator<T>

Technically, any object that has a method named MoveNext and a property called Current is treated as an enumerator; this relaxation exists to allow enumeration of value type elements in C# 1.0 without a boxing/unboxing overhead. This optimization is now obsolete with generics, and is, in fact, unsupported with C# 4.0's dynamic binding.