CHAPTER 10

Properties

In this chapter:

In this chapter, I'll talk about properties. Properties allow source code to call a method by using a simplified syntax. The common language runtime (CLR) offers two kinds of properties: parameterless properties, which are simply called *properties*, and parameterful properties, which are called different names by different programming languages. For example, C# calls parameterful properties *indexers*, and Microsoft Visual Basic calls them *default properties*. I'll also talk about initializing properties by using object and collection initializers, in addition to ways to package a bunch of properties together by using C#'s anonymous types and the `System.Tuple` type.

Parameterless Properties

Many types define state information that can be retrieved or altered. Frequently, this state information is implemented as field members of the type. For example, here's a type definition that contains two fields.

```
public sealed class Employee {
   public String Name; // The employee's name
   public Int32  Age;  // The employee's age
}
```

If you were to create an instance of this type, you could easily get or set any of this state information with code similar to the following.

```
Employee e = new Employee();
e.Name = "Jeffrey Richter";  // Set the employee's Name.
e.Age  = 48;                 // Set the employee's Age.

Console.WriteLine(e.Name);   // Displays "Jeffrey Richter"
```

Querying and setting an object's state information in the way I just demonstrated is very common. However, I would argue that the preceding code should never be implemented as shown. One of the hallmarks of object-oriented design and programming is *data encapsulation*. Data encapsulation means that your type's fields should never be publicly exposed because it's too easy to write code that improperly uses the fields, corrupting the object's state. For example, a developer could easily corrupt an `Employee` object with code like the following.

```
e.Age = -5; // How could someone be -5 years old?
```

There are additional reasons for encapsulating access to a type's data field. For example, you might want access to a field to execute some side effect, cache some value, or lazily create some internal object. You might also want access to the field to be thread-safe. Or perhaps the field is a logical field whose value isn't represented by bytes in memory but whose value is instead calculated using some algorithm.

For any of these reasons, when designing a type, I strongly suggest that all of your fields be `private`. Then, to allow a user of your type to get or set state information, you expose methods for that specific purpose. Methods that wrap access to a field are typically called accessor methods. These accessor methods can optionally perform sanity checking and ensure that the object's state is never corrupted. For example, I'd rewrite the previous class as follows.

```
public sealed class Employee {
   private String m_Name;    // Field is now private
   private Int32  m_Age;     // Field is now private

   public String GetName() {
      return(m_Name);
   }

   public void SetName(String value) {
      m_Name = value;
   }

   public Int32 GetAge() {
      return(m_Age);
   }

   public void SetAge(Int32 value) {
      if (value < 0)
         throw new ArgumentOutOfRangeException("value", value.ToString(),
            "The value must be greater than or equal to 0");
      m_Age = value;
   }
}
```

Although this is a simple example, you should still be able to see the enormous benefit you get from encapsulating the data fields. You should also be able to see how easy it is to make read-only or write-only properties: just don't implement one of the accessor methods. Alternatively, you could allow only derived types to modify the value by marking the `SetXxx` method as `protected`.

Encapsulating the data as shown earlier has two disadvantages. First, you have to write more code because you now have to implement additional methods. Second, users of the type must now call methods rather than simply refer to a single field name.

```
e.SetName("Jeffrey Richter");      // Updates the employee's name
String EmployeeName = e.GetName(); // Retrieves the employee's name
e.SetAge(48);                      // Updates the employee's age
e.SetAge(-5);                      // Throws ArgumentOutOfRangeException
Int32 EmployeeAge = e.GetAge();    // Retrieves the employee's age
```

Personally, I think these disadvantages are quite minor. Nevertheless, programming languages and the CLR offer a mechanism called *properties* that alleviates the first disadvantage a little and removes the second disadvantage entirely.

The class shown here uses properties and is functionally identical to the class shown earlier.

```
public sealed class Employee {
   private String m_Name;
   private Int32  m_Age;

   public String Name {
      get { return(m_Name); }
      set { m_Name = value; } // The 'value' keyword always identifies the new value.
   }

   public Int32 Age {
      get { return(m_Age); }
      set {
         if (value < 0)    // The 'value' keyword always identifies the new value.
            throw new ArgumentOutOfRangeException("value", value.ToString(),
               "The value must be greater than or equal to 0");
         m_Age = value;
      }
   }
}
```

As you can see, properties complicate the definition of the type slightly, but the fact that they allow you to write your code as follows more than compensates for the extra work.

```
e.Name = "Jeffrey Richter";   // "Sets" the employee name
String EmployeeName = e.Name; // "Gets" the employee's name
e.Age = 48;                   // "Sets" the employee's age
e.Age = -5;                   // Throws ArgumentOutOfRangeException
Int32 EmployeeAge = e.Age;    // "Gets" the employee's age
```

You can think of properties as smart fields: fields with additional logic behind them. The CLR supports static, instance, abstract, and virtual properties. In addition, properties can be marked with any accessibility modifier (discussed in Chapter 6, "Type and Member Basics") and defined within an interface (discussed in Chapter 13, "Interfaces").

Each property has a name and a type (which can't be `void`). It isn't possible to overload properties (that is, have two properties with the same name if their types are different). When you define a property, you typically specify both a `get` and a `set` method. However, you can leave out the `set` method to define a read-only property or leave out the `get` method to define a write-only property.

It's also quite common for the property's `get/set` methods to manipulate a private field defined within the type. This field is commonly referred to as the *backing field*. The `get` and `set` methods don't have to access a backing field, however. For example, the `System.Threading.Thread` type offers a `Priority` property that communicates directly with the operating system; the `Thread` object doesn't maintain a field for a thread's priority. Another example of properties without backing fields are those read-only properties calculated at run time—for example, the length of a zero-terminated array or the area of a rectangle when you have its height and width.

When you define a property, depending on its definition, the compiler will emit either two or three of the following items into the resulting managed assembly:

- A method representing the property's `get` accessor method. This is emitted only if you define a `get` accessor method for the property.
- A method representing the property's `set` accessor method. This is emitted only if you define a `set` accessor method for the property.
- A property definition in the managed assembly's metadata. This is always emitted.

Refer back to the `Employee` type shown earlier. As the compiler compiles this type, it comes across the `Name` and `Age` properties. Because both properties have `get` and `set` accessor methods, the compiler emits four method definitions into the `Employee` type. It's as though the original source were written as follows.

```
public sealed class Employee {
   private String m_Name;
   private Int32  m_Age;

   public String get_Name(){
      return m_Name;
   }
   public void   set_Name(String value) {
      m_Name = value; // The argument 'value' always identifies the new value.
   }

   public Int32 get_Age() {
      return m_Age;
   }

   public void  set_Age(Int32 value) {
      if (value < 0)      // The 'value' always identifies the new value.
         throw new ArgumentOutOfRangeException("value", value.ToString(),
           "The value must be greater than or equal to 0");
      m_Age = value;
   }
}
```

The compiler automatically generates names for these methods by prepending `get_` or `set_` to the property name specified by the developer.

C# has built-in support for properties. When the C# compiler sees code that's trying to get or set a property, the compiler actually emits a call to one of these methods. If you're using a programming language that doesn't directly support properties, you can still access properties by calling the desired accessor method. The effect is exactly the same; it's just that the source code doesn't look as pretty.

In addition to emitting the accessor methods, compilers also emit a property definition entry into the managed assembly's metadata for each property defined in the source code. This entry contains some flags and the type of the property, and it refers to the `get` and `set` accessor methods. This information exists simply to draw an association between the abstract concept of a "property" and its accessor methods. Compilers and other tools can use this metadata, which can be obtained by using the `System.Reflection.PropertyInfo` class. The CLR doesn't use this metadata information and requires only the accessor methods at run time.

Automatically Implemented Properties

If you are creating a property to simply encapsulate a backing field, then C# offers a simplified syntax known as *automatically implemented properties* (AIPs), as shown here for the `Name` property.

```
public sealed class Employee {
   // This property is an automatically implemented property
   public String Name { get; set; }

   private Int32  m_Age;


   public Int32 Age {
      get { return(m_Age); }
      set {
         if (value < 0)    // The 'value' keyword always identifies the new value.
            throw new ArgumentOutOfRangeException("value", value.ToString(),
               "The value must be greater than or equal to 0");
         m_Age = value;
      }
   }
}
```

When you declare a property and do not provide an implementation for the `get/set` methods, then the C# compiler will automatically declare for you a private field. In this example, the field will be of type `String`, the type of the property. And, the compiler will automatically implement the `get_Name` and `set_Name` methods for you to return the value in the field and to set the field's value, respectively.

You might wonder what the value of doing this is, as opposed to just declaring a `public String` field called `Name`. Well, there is a big difference. Using the AIP syntax means that you have created a property. Any code that accesses this property is actually calling `get` and `set` methods. If you decide later to implement the `get` and/or `set` method yourself instead of accepting the compiler's default

implementation, then any code that accesses the property will not have to be recompiled. However, if you declared `Name` as a field and then you later change it to a property, then all code that accessed the field will have to be recompiled so that it now accesses the property methods.

- Personally, I do not like the compiler's AIP feature, so I usually avoid it for the following reason: The syntax for a field declaration can include initialization so that you are declaring and initializing the field in one line of code. However, there is no convenient syntax to set an AIP to an initial value. Therefore, you must explicitly initialize each AIP in each constructor method.
- The runtime serialization engines persist the name of the field in a serialized stream. The name of the backing field for an AIP is determined by the compiler, and it could actually change the name of this backing field every time you recompile your code, negating the ability to deserialize instances of any types that contain an AIP. Do not use the AIP feature with any type you intend to serialize or deserialize.
- When debugging, you cannot put a breakpoint on an AIP `get` or `set` method, so you cannot easily detect when an application is getting or setting this property. You can set breakpoints on manually implemented properties, which can be quite handy when tracking down bugs.

You should also know that when you use AIPs, the property must be readable and writable; that is, the compiler must produce both `get` and `set` methods. This makes sense because a write-only field is not useful without the ability to read its value; likewise, a read-only field would always have its default value. In addition, because you do not know the name of the compiler-generated backing field, your code must always access the property by using the property name. And, if you decide you want to explicitly implement one of the accessor methods, then you must explicitly implement both accessor methods and you are not using the AIP feature anymore. For a single property, the AIP feature is an all-or-nothing deal.

Defining Properties Intelligently

Personally, I don't like properties and I wish that they were not supported in the Microsoft .NET Framework and its programming languages. The reason is that properties look like fields, but they are methods. This has been known to cause a phenomenal amount of confusion. When a programmer sees code that appears to be accessing a field, there are many assumptions that the programmer makes that may not be true for a property. For example:

- A property may be read-only or write-only; field access is always readable and writable. If you define a property, it is best to offer both `get` and `set` accessor methods.
- A property method may throw an exception; field access never throws an exception.
- A property cannot be passed as an `out` or `ref` parameter to a method; a field can. For example, the following code will not compile.

```
using System;

public sealed class SomeType {
   private static String Name {
```

```
      get { return null; }
      set {}
   }

   static void MethodWithOutParam(out String n) { n = null; }

   public static void Main() {
      // For the line of code below, the C# compiler emits the following:
      // error CS0206: A property, indexer or dynamic member access may not
      // be passed as an out or ref parameter
      MethodWithOutParam(out Name);
   }
}
```

- A property method can take a long time to execute; field access always completes immediately. A common reason to use properties is to perform thread synchronization, which can stop the thread forever, and therefore, a property should not be used if thread synchronization is required. In that situation, a method is preferred. Also, if your class can be accessed remotely (for example, your class is derived from `System.MarshalByRefObject`), calling the property method will be very slow, and therefore, a method is preferred to a property. In my opinion, classes derived from `MarshalByRefObject` should never use properties.

- If called multiple times in a row, a property method may return a different value each time; a field returns the same value each time. The `System.DateTime` class has a read-only `Now` property that returns the current date and time. Each time you query this property, it will return a different value. This is a mistake, and Microsoft wishes that they could fix the class by making `Now` a method instead of a property. `Environment`'s `TickCount` property is another example of this mistake.

- A property method may cause observable side effects; field access never does. In other words, a user of a type should be able to set various properties defined by a type in any order he or she chooses without noticing any different behavior in the type.

- A property method may require additional memory or return a reference to something that is not actually part of the object's state, so modifying the returned object has no effect on the original object; querying a field always returns a reference to an object that is guaranteed to be part of the original object's state. Working with a property that returns a copy can be very confusing to developers, and this characteristic is frequently not documented.

It has come to my attention that people use properties far more often than they should. If you examine this list of differences between properties and fields, you'll see that there are very few circumstances in which defining a property is actually useful and will not cause confusion for developers. The only thing that properties buy you is some simplified syntax; there is no performance benefit compared to calling a non-property method, and understandability of the code is reduced. If I had been involved in the design of the .NET Framework and compilers, I would have not offered properties at all; instead, I would have programmers actually implement `GetXxx` and `SetXxx` methods as desired. Then, if compilers wanted to offer some special, simplified syntax for calling these methods, so be it. But I'd want the compiler to use syntax that is different from field access syntax so that programmers really understand what they are doing—a method call.

Properties and the Visual Studio Debugger

Microsoft Visual Studio allows you to enter an object's property in the debugger's watch window. When you do this, every time you hit a breakpoint, the debugger calls into the property's `get` accessor method and displays the returned value. This can be quite helpful in tracking down bugs, but it can also cause bugs to occur and hurt your debugging performance. For example, let's say that you have created a `FileStream` for a file on a network share and then you add `FileStream`'s `Length` property to the debugger's watch window. Now, every time you hit a breakpoint, the debugger will call `Length`'s `get` accessor method, which internally makes a network request to the server to get the current length of the file!

Similarly, if your property's `get` accessor method has a side effect, then this side effect will execute every time you hit a breakpoint. For example, let's say that your property's `get` accessor method increments a counter every time it is called; this counter will now be incremented every time you hit a breakpoint, too. Because of these potential problems, Visual Studio allows you to turn off property evaluation for properties shown in watch windows. To turn property evaluation off in Visual Studio, select Tools, Options, Debugging, and General and in the list box in Figure 10-1, and clear the Enable Property Evaluation And Other Implicit Function Calls check box. Note that even with this item cleared, you can add the property to the watch window and manually force Visual Studio to evaluate it by clicking the force evaluation circle in the watch window's Value column.

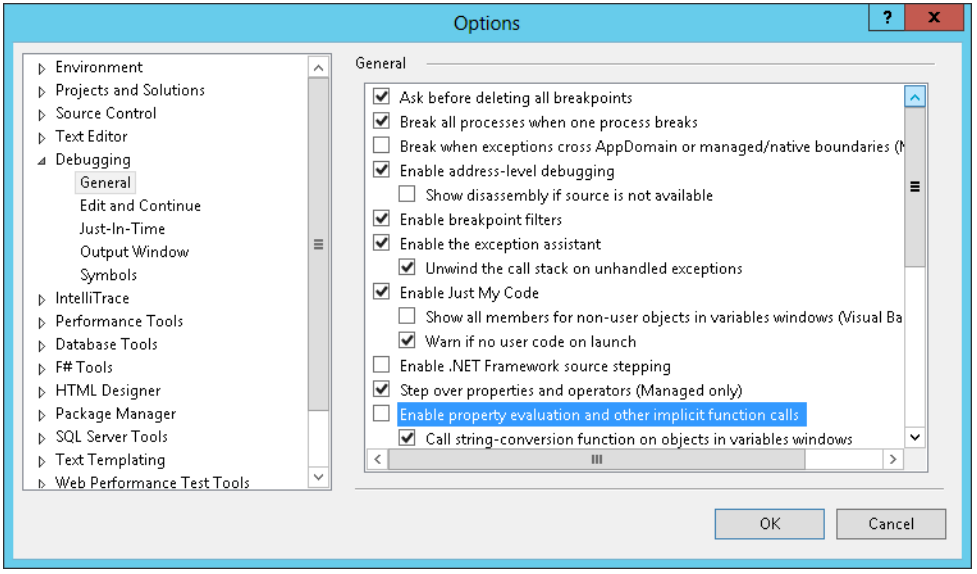

FIGURE 10-1 The Visual Studio General Debugger settings.

Object and Collection Initializers

It is very common to construct an object and then set some of the object's public properties (or fields). To simplify this common programming pattern, the C# language supports a special object initialization syntax. The following is an example.

```
Employee e = new Employee() { Name = "Jeff", Age = 45 };
```

With this one statement, I am constructing an `Employee` object, calling its parameterless constructor, and then setting its public `Name` property to `"Jeff"` and its public `Age` property to 45. In fact, the preceding code is identical to the following, which you could verify by examining the Intermediate Language (IL) for both of these code fragments.

```
Employee _tempVar = new Employee();
_tempVar.Name = "Jeff";
_tempVar.Age = 45;

// Only assign to e if the assignments above don't throw an exception.
// This prevents e from referring to a partially initialized object.
Employee e = _tempVar;
```

The real benefit of the object initializer syntax is that it allows you to code in an expression context (as opposed to a statement context), permitting composability of functions, which in turn increases code readability. For example, I can now write the following.

```
String s = new Employee() { Name = "Jeff", Age = 45 }.ToString().ToUpper();
```

So now, in one statement, I have constructed an `Employee` object, called its constructor, initialized two public properties, and then, using the resulting expression, called `ToString` on it followed by calling `ToUpper`. For more about composability of functions, see the "Extension Methods" section in Chapter 8, "Methods."

As a small side note, C# also lets you omit the parentheses before the open brace if you want to call a parameterless constructor. The following line produces the same IL as the preceding line.

```
String s = new Employee { Name = "Jeff", Age = 45 }.ToString().ToUpper();
```

If a property's type implements the `IEnumerable` or `IEnumerable<T>` interface, then the property is considered to be a collection, and initializing a collection is an additive operation as opposed to a replacement operation. For example, suppose I have the following class definition.

```
public sealed class Classroom {
   private List<String> m_students = new List<String>();
   public List<String> Students { get { return m_students; } }

   public Classroom() {}
}
```

I can now have code that constructs a `Classroom` object and initializes the `Students` collection as follows.

```
public static void M() {
   Classroom classroom = new Classroom {
      Students = { "Jeff", "Kristin", "Aidan", "Grant" }
   };

   // Show the 4 students in the classroom
   foreach (var student in classroom.Students)
      Console.WriteLine(student);
}
```

When compiling this code, the compiler sees that the `Students` property is of type `List<String>` and that this type implements the `IEnumerable<String>` interface. Now, the compiler assumes that the `List<String>` type offers a method called `Add` (because most collection classes actually offer an `Add` method that adds items to the collection). The compiler then generates code to call the collection's `Add` method. So, the preceding code is converted by the compiler into the following.

```
public static void M() {
   Classroom classroom = new Classroom();
   classroom.Students.Add("Jeff");
   classroom.Students.Add("Kristin");
   classroom.Students.Add("Aidan");
   classroom.Students.Add("Grant");

   // Show the 4 students in the classroom
   foreach (var student in classroom.Students)
      Console.WriteLine(student);
}
```

If the property's type implements `IEnumerable` or `IEnumerable<T>` but the type doesn't offer an `Add` method, then the compiler does not let you use the collection initialize syntax to add items to the collection; instead, the compiler issues something like the following message: `error CS0117: 'System.Collections.Generic.IEnumerable<string>' does not contain a definition for 'Add'.`

Some collection's `Add` methods take multiple arguments—for example, `Dictionary`'s `Add` method.

```
public void Add(TKey key, TValue value);
```

You can pass multiple arguments to an `Add` method by using nested braces in a collection initializer, as follows.

```
var table = new Dictionary<String, Int32> {
   { "Jeffrey", 1 }, { "Kristin", 2 }, { "Aidan", 3 }, { "Grant", 4 }
};
```

The preceding line is identical to the following.

```
var table = new Dictionary<String, Int32>();
table.Add("Jeffrey", 1);
table.Add("Kristin", 2);
table.Add("Aidan", 3);
table.Add("Grant", 4);
```

Anonymous Types

C#'s anonymous type feature allows you to automatically declare an immutable tuple type by using a very simple and succinct syntax. A *tuple type* is a type that contains a collection of properties that are usually related to each other in some way.[1] In the top line of the following code, I am defining a class with two properties (`Name` of type `String`, and `Year` of type `Int32`), constructing an instance of this type, and setting its `Name` property to `"Jeff"` and its `Year` property to `1964`.

```
// Define a type, construct an instance of it, & initialize its properties
var o1 = new { Name = "Jeff", Year = 1964 };

// Display the properties on the console:
Console.WriteLine("Name={0}, Year={1}", o1.Name, o1.Year);// Displays: Name=Jeff, Year=1964
```

This top line of code creates an anonymous type because I did not specify a type name after the `new` keyword, so the compiler will create a type name for me automatically and not tell me what it is (which is why it is called an anonymous type). The line of code uses the object initializer syntax discussed in the previous section to declare the properties and also to initialize these properties. Also, because I (the developer) do not know the name of the type at compile time, I do not know what type to declare the variable `o1` as. However, this is not a problem, because I can use C#'s implicitly typed local variable feature (`var`), as discussed in Chapter 9, "Parameters," to have the compiler infer the type from the expression on the right of the assignment operator (=).

Now, let's focus on what the compiler is actually doing. When you write a line of code like this:

```
var o = new { property1 = expression1, ..., propertyN = expressionN };
```

the compiler infers the type of each expression, creates private fields of these inferred types, creates public read-only properties for each of the fields, and creates a constructor that accepts all these expressions. The constructor's code initializes the private read-only fields from the expression results passed in to it. In addition, the compiler overrides `Object`'s `Equals`, `GetHashCode`, and `ToString` methods and generates code inside all these methods. In effect, the class that the compiler generates looks like the following.

```
[CompilerGenerated]
internal sealed class <>f__AnonymousType0<...>: Object {
   private readonly t1 f1;
   public  t1 p1 { get { return f1; } }

   ...
```

[1] The term originated as an abstraction of the sequence: single, double, triple, quadruple, quintuple, n-tuple.

```
   private readonly tn fn;
   public  tn pn { get { return fn; } }

   public <>f__AnonymousType0<...>(t1 a1, ..., tn an) {
      f1 = a1; ...; fn = an; // Set all fields
   }

   public override Boolean Equals(Object value) {
      // Return false if any fields don't match; else true
   }

   public override Int32 GetHashCode() {
      // Returns a hash code generated from each fields' hash code
   }

   public override String ToString() {
      // Return comma-separated set of property name = value pairs
   }
}
```

The compiler generates `Equals` and `GetHashCode` methods so that instances of the anonymous type can be placed in a hash table collection. The properties are `readonly` as opposed to read/write to help prevent the object's hashcode from changing. Changing the hashcode for an object used as a key in a hashtable can prevent the object from being found. The compiler generates the `ToString` method to help with debugging. In the Visual Studio debugger, you can place the mouse cursor over a variable that refers to an instance of an anonymous type, and Visual Studio will invoke the `ToString` method and show the resulting string in a datatip window. By the way, Visual Studio's IntelliSense will suggest the property names as you write code in the editor—a very nice feature.

The compiler supports two additional syntaxes for declaring a property inside an anonymous type where it can infer the property names and types from variables.

```
String Name = "Grant";
DateTime dt = DateTime.Now;

// Anonymous type with two properties
//  1. String Name property set to Grant
//  2. Int32 Year property set to the year inside the dt
var o2 = new { Name, dt.Year };
```

In this example, the compiler determines that the first property should be called `Name`. Because `Name` is the name of a local variable, the compiler sets the type of the property to be the same type as the local variable: `String`. For the second property, the compiler uses the name of the field/property: `Year`. `Year` is an `Int32` property of the `DateTime` class, and therefore the `Year` property in the anonymous type will also be an `Int32`. Now, when the compiler constructs an instance of this anonymous type, it will set the instance's `Name` property to the same value that is in the `Name` local variable so the `Name` property will refer to the same `"Grant"` string. The compiler will set the instance's `Year` property to the same value that is returned from `dt`'s `Year` property.

The compiler is very intelligent about defining anonymous types. If the compiler sees that you are defining multiple anonymous types in your source code that have the identical structure, the compiler will create just one definition for the anonymous type and create multiple instances of that type. By "same structure," I mean that the anonymous types have the same type and name for each property and that these properties are specified in the same order. In the preceding code examples, the type of variable `o1` and the type of variable `o2` will be the same type because the two lines of code are defining an anonymous type with a `Name/String` property and a `Year/Int32` property, and `Name` comes before `Year`.

Because the two variables are of the same type, we get to do some cool things, such as checking whether the two objects contain equal values and assigning a reference to one object into the other's variable, as follows.

```
// One type allows equality and assignment operations.
Console.WriteLine("Objects are equal: " + o1.Equals(o2));
o1 = o2;  // Assignment
```

Also, because of this type identity, we can create an implicitly typed array (discussed in the "Initializing Array Elements" section in Chapter 16, "Arrays") of anonymous types.

```
// This works because all of the objects are of the same anonymous type
var people = new[] {
   o1,  // From earlier in this section
   new { Name = "Kristin", Year = 1970 },
   new { Name = "Aidan", Year = 2003 },
   new { Name = "Grant", Year = 2008 }
};

// This shows how to walk through the array of anonymous types (var is required)
foreach (var person in people)
   Console.WriteLine("Person={0}, Year={1}", person.Name, person.Year);
```

Anonymous types are most commonly used with the Language Integrated Query (LINQ) technology, where you perform a query that results in a collection of objects that are all of the same anonymous type. Then, you process the objects in the resulting collection. All this takes place in the same method. Here is an example that returns all the files in my document directory that have been modified within the past seven days.

```
String myDocuments = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
var query =
         from pathname in Directory.GetFiles(myDocuments)
         let LastWriteTime = File.GetLastWriteTime(pathname)
         where LastWriteTime > (DateTime.Now - TimeSpan.FromDays(7))
         orderby LastWriteTime
         select new { Path = pathname, LastWriteTime };// Set of anonymous type objects

foreach (var file in query)
   Console.WriteLine("LastWriteTime={0}, Path={1}", file.LastWriteTime, file.Path);
```

Instances of anonymous types are not supposed to leak outside of a method. A method cannot be prototyped as accepting a parameter of an anonymous type because there is no way to specify the anonymous type. Similarly, a method cannot indicate that it returns a reference to an anonymous type. Although it is possible to treat an instance of an anonymous type as an `Object` (because all anonymous types are derived from `Object`), there is no way to cast a variable of type `Object` back into an anonymous type because you don't know the name of the anonymous type at compile time. If you want to pass a tuple around, then you should consider using the `System.Tuple` type discussed in the next section.

The `System.Tuple` Type

In the `System` namespace, Microsoft has defined several generic `Tuple` types (all derived from `Object`) that differ by arity (the number of generic parameters). Here is what the simplest and most complex ones essentially look like.

```
// This is the simplest:
[Serializable]
public class Tuple<T1> {
   private T1 m_Item1;
   public Tuple(T1 item1) { m_Item1 = item1; }
   public T1 Item1 { get { return m_Item1; } }
}


// This is the most complex:
[Serializable]
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> {
   private T1 m_Item1; private T2 m_Item2; private T3 m_Item3; private T4 m_Item4;
   private T5 m_Item5; private T6 m_Item6; private T7 m_Item7; private TRest m_Rest;

   public Tuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5, T6 item6, T7 item7,
      TRest rest) {
      m_Item1 = item1; m_Item2 = item2; m_Item3 = item3; m_Item4 = item4;
      m_Item5 = item5; m_Item6 = item6; m_Item7 = item7; m_Rest = rest;
   }

   public T1 Item1 { get { return m_Item1; } }
   public T2 Item2 { get { return m_Item2; } }
   public T3 Item3 { get { return m_Item3; } }
   public T4 Item4 { get { return m_Item4; } }
   public T5 Item5 { get { return m_Item5; } }
   public T6 Item6 { get { return m_Item6; } }
   public T7 Item7 { get { return m_Item7; } }
   public TRest Rest { get { return m_Rest; } }
}
```

Like anonymous types, after a `Tuple` is created, it is immutable (all properties are read-only). I don't show it here, but the `Tuple` classes also offer `CompareTo`, `Equals`, `GetHashCode`, and `ToString` methods, as well as a `Size` property. In addition, all the `Tuple` types implement the `IStructural-Equatable`, `IStructuralComparable`, and `IComparable` interfaces so that you can compare two `Tuple` objects with each other to see how their fields compare with each other. Refer to the SDK documentation to learn more about these methods and interfaces.

Here is an example of a method that uses a `Tuple` type to return two pieces of information to a caller.

```
// Returns minimum in Item1 & maximum in Item2
private static Tuple<Int32, Int32> MinMax(Int32 a, Int32 b) {
   return new Tuple<Int32, Int32>(Math.Min(a, b), Math.Max(a, b));
}

// This shows how to call the method and how to use the returned Tuple
private static void TupleTypes() {
   var minmax = MinMax(6, 2);
   Console.WriteLine("Min={0}, Max={1}", minmax.Item1, minmax.Item2); // Min=2, Max=6
}
```

Of course, it is very important that the producer and consumer of the `Tuple` have a clear understanding of what is being returned in the `Item#` properties. With anonymous types, the properties are given actual names based on the source code that defines the anonymous type. With `Tuple` types, the properties are assigned their `Item#` names by Microsoft and you cannot change this at all. Unfortunately, these names have no real meaning or significance, so it is up to the producer and consumer to assign meanings to them. This also reduces code readability and maintainability so you should add comments to your code explaining what the producer/consumer understanding is.

The compiler can only infer generic types when calling a generic method, not when you are calling a constructor. For this reason, the `System` namespace also includes a non-generic, static `Tuple` class containing a bunch of static `Create` methods that can infer generic types from arguments. This class acts as a factory for creating `Tuple` objects, and it exists simply to simplify your code. Here is a rewrite of the `MinMax` method shown earlier using the static `Tuple` class.

```
// Returns minimum in Item1 & maximum in Item2
private static Tuple<Int32, Int32> MinMax(Int32 a, Int32 b) {
   return Tuple.Create(Math.Min(a, b), Math.Max(a, b));       // Simpler syntax
}
```

If you want to create a `Tuple` with more than eight elements in it, then you would pass another `Tuple` for the `Rest` parameter as follows.

```
var t = Tuple.Create(0, 1, 2, 3, 4, 5, 6, Tuple.Create(7, 8));
Console.WriteLine("{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}",
   t.Item1, t.Item2, t.Item3, t.Item4, t.Item5, t.Item6, t.Item7,
   t.Rest.Item1.Item1, t.Rest.Item1.Item2);
```

Note In addition to anonymous types and the `Tuple` types, you might want to take a look at the `System.Dynamic.ExpandoObject` class (defined in the `System.Core.dll` assembly). When you use this class with C#'s `dynamic` type (discussed in Chapter 5, "Primitive, Reference, and Value Types"), you have another way of grouping a set of properties (key/value pairs) together. The result is not compile-time type-safe, but the syntax looks nice (although you get no IntelliSense support), and you can pass `ExpandoObject` objects between C# and dynamic languages like Python. Here's some sample code that uses an `ExpandoObject`.

```
dynamic e = new System.Dynamic.ExpandoObject();
e.x = 6; // Add an Int32 'x' property whose value is 6
e.y = "Jeff";    // Add a String 'y' property whose value is "Jeff"
e.z = null;      // Add an Object 'z' property whose value is null

// See all the properties and their values:
foreach (var v in (IDictionary<String, Object>)e)
   Console.WriteLine("Key={0}, V={1}", v.Key, v.Value);


// Remove the 'x' property and its value
var d = (IDictionary<String, Object>)e;
d.Remove("x");
```

Parameterful Properties

In the previous section, the `get` accessor methods for the properties accepted no parameters. For this reason, I called these properties *parameterless properties*. These properties are easy to understand because they have the feel of accessing a field. In addition to these field-like properties, programming languages also support what I call *parameterful properties*, whose `get` accessor methods accept one or more parameters and whose `set` accessor methods accept two or more parameters. Different programming languages expose parameterful properties in different ways. Also, languages use different terms to refer to parameterful properties: C# calls them indexers and Visual Basic calls them *default properties*. In this section, I'll focus on how C# exposes its indexers by using parameterful properties.

In C#, parameterful properties (indexers) are exposed using an array-like syntax. In other words, you can think of an indexer as a way for the C# developer to overload the `[]` operator. Here's an example of a `BitArray` class that allows array-like syntax to index into the set of bits maintained by an instance of the class.

```
using System;

public sealed class BitArray {
   // Private array of bytes that hold the bits
   private Byte[] m_byteArray;
   private Int32  m_numBits;

   // Constructor that allocates the byte array and sets all bits to 0
   public BitArray(Int32 numBits) {
```

```
      // Validate arguments first.
      if (numBits <= 0)
         throw new ArgumentOutOfRangeException("numBits must be > 0");

      // Save the number of bits.
      m_numBits = numBits;

      // Allocate the bytes for the bit array.
      m_byteArray = new Byte[(numBits + 7) / 8];
   }

   // This is the indexer (parameterful property).
   public Boolean this[Int32 bitPos] {

      // This is the indexer's get accessor method.
      get {
         // Validate arguments first
         if ((bitPos < 0) || (bitPos >= m_numBits))
            throw new ArgumentOutOfRangeException("bitPos");

         // Return the state of the indexed bit.
         return (m_byteArray[bitPos / 8] & (1 << (bitPos % 8))) != 0;
      }

      // This is the indexer's set accessor method.
      set {
         if ((bitPos < 0) || (bitPos >= m_numBits))
            throw new ArgumentOutOfRangeException("bitPos", bitPos.ToString());
         if (value) {
            // Turn the indexed bit on.
            m_byteArray[bitPos / 8] = (Byte)
               (m_byteArray[bitPos / 8] | (1 << (bitPos % 8)));
         } else {
            // Turn the indexed bit off.
            m_byteArray[bitPos / 8] = (Byte)
               (m_byteArray[bitPos / 8] & ~(1 << (bitPos % 8)));
         }
      }
   }
}
```

Using the `BitArray` class's indexer is incredibly simple.

```
// Allocate a BitArray that can hold 14 bits.
BitArray ba = new BitArray(14);

// Turn all the even-numbered bits on by calling the set accessor.
for (Int32 x = 0; x < 14; x++) {
   ba[x] = (x % 2 == 0);
}

// Show the state of all the bits by calling the get accessor.
for (Int32 x = 0; x < 14; x++) {
   Console.WriteLine("Bit " + x + " is " + (ba[x] ? "On" : "Off"));
}
```

In the `BitArray` example, the indexer takes one `Int32` parameter, `bitPos`. All indexers must have at least one parameter, but they can have more. These parameters (as well as the return type) can be of any data type (except `void`). An example of an indexer that has more than one parameter can be found in the `System.Drawing.Imaging.ColorMatrix` class, which ships in the `System.Drawing.dll` assembly.

It's quite common to create an indexer to look up values in an associative array. In fact, the `System.Collections.Generic.Dictionary` type offers an indexer that takes a key and returns the value associated with the key. Unlike parameterless properties, a type can offer multiple, overloaded indexers as long as their signatures differ.

Like a parameterless property's `set` accessor method, an indexer's `set` accessor method also contains a hidden parameter, called `value` in C#. This parameter indicates the new value desired for the "indexed element."

The CLR doesn't differentiate parameterless properties and parameterful properties; to the CLR, each is simply a pair of methods and a piece of metadata defined within a type. As mentioned earlier, different programming languages require different syntax to create and use parameterful properties. The fact that C# requires `this[...]` as the syntax for expressing an indexer was purely a choice made by the C# team. What this choice means is that C# allows indexers to be defined only on instances of objects. C# doesn't offer syntax allowing a developer to define a static indexer property, although the CLR does support static parameterful properties.

Because the CLR treats parameterful properties just as it does parameterless properties, the compiler will emit either two or three of the following items into the resulting managed assembly:

- A method representing the parameterful property's `get` accessor method. This is emitted only if you define a `get` accessor method for the property.
- A method representing the parameterful property's `set` accessor method. This is emitted only if you define a `set` accessor method for the property.
- A property definition in the managed assembly's metadata, which is always emitted. There's no special parameterful property metadata definition table because, to the CLR, parameterful properties are just properties.

For the `BitArray` class shown earlier, the compiler compiles the indexer as though the original source code were written as follows.

```
public sealed class BitArray {

   // This is the indexer's get accessor method.
   public Boolean get_Item(Int32 bitPos) { /* ... */ }

   // This is the indexer's set accessor method.
   public void    set_Item(Int32 bitPos, Boolean value)  { /* ... */ }
}
```

The compiler automatically generates names for these methods by prepending `get_` and `set_` to the *indexer name*. Because the C# syntax for an indexer doesn't allow the developer to specify an *indexer name*, the C# compiler team had to choose a default name to use for the accessor methods; they chose `Item`. Therefore, the method names emitted by the compiler are `get_Item` and `set_Item`.

When examining the .NET Framework Reference documentation, you can tell if a type offers an indexer by looking for a property named `Item`. For example, the `System.Collections.Generic.List` type offers a public instance property named `Item`; this property is `List`'s indexer.

When you program in C#, you never see the name of `Item`, so you don't normally care that the compiler has chosen this name for you. However, if you're designing an indexer for a type that code written in other programming languages will be accessing, you might want to change the default name, `Item`, given to your indexer's `get` and `set` accessor methods. C# allows you to rename these methods by applying the `System.Runtime.CompilerServices.IndexerNameAttribute` custom attribute to the indexer. The following code demonstrates how to do this.

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

public sealed class BitArray {

   [IndexerName("Bit")]
   public Boolean this[Int32 bitPos] {
      // At least one accessor method is defined here
   }
}
```

Now the compiler will emit methods called `get_Bit` and `set_Bit` instead of `get_Item` and `set_Item`. When compiling, the C# compiler sees the `IndexerName` attribute, and this tells the compiler how to name the methods and the property metadata; the attribute itself is not emitted into the assembly's metadata.[2]

Here's some Visual Basic code that demonstrates how to access this C# indexer.

```
' Construct an instance of the BitArray type.
Dim ba as New BitArray(10)

' Visual Basic uses () instead of [] to specify array elements.
Console.WriteLine(ba(2))       ' Displays True or False

' Visual Basic also allows you to access the indexer by its name.
Console.WriteLine(ba.Bit(2))   ' Displays same as previous line
```

In C#, a single type can define multiple indexers as long as the indexers all take different parameter sets. In other programming languages, the `IndexerName` attribute allows you to define multiple indexers with the same signature because each can have a different name. The reason C# won't allow you to do this is because its syntax doesn't refer to the indexer by name; the compiler wouldn't know which indexer you were referring to. Attempting to compile the following C# source code causes the

[2] For this reason, the `IndexerNameAttribute` class is not part of the ECMA standardization of the CLI and the C# language.

compiler to generate the following message: `error CO111: Type 'SomeType' already defines a member called 'this' with the same parameter types.`

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

public sealed class SomeType {

   // Define a get_Item accessor method.
   public Int32 this[Boolean b] {
      get { return 0; }
   }

   // Define a get_Jeff accessor method.
   [IndexerName("Jeff")]
   public String this[Boolean b] {
      get { return null; }
   }
}
```

You can clearly see that C# thinks of indexers as a way to overload the `[]` operator, and this operator can't be used to disambiguate parameterful properties with different method names and identical parameter sets.

By the way, the `System.String` type is an example of a type that changed the name of its indexer. The name of `String`'s indexer is `Chars` instead of `Item`. This read-only property allows you to get an individual character within a string. For programming languages that don't use `[]` operator syntax to access this property, `Chars` was decided to be a more meaningful name.

Selecting the Primary Parameterful Property

C#'s limitations with respect to indexers brings up the following two questions:

- What if a type is defined in a programming language that does allow the developer to define several parameterful properties?
- How can this type be consumed from C#?

The answer to both questions is that a type must select one of the parameterful property names to be the default property by applying an instance of `System.Reflection.DefaultMemberAttribute` to the class itself. For the record, `DefaultMemberAttribute` can be applied to a class, a structure, or an interface. In C#, when you compile a type that defines a parameterful property, the compiler automatically applies an instance of `DefaultMember` attribute to the defining type and takes it into account when you use the `IndexerName` attribute. This attribute's constructor specifies the name that is to be used for the type's default parameterful property.

So, in C#, if you define a type that has a parameterful property and you don't specify the `IndexerName` attribute, the defining type will have a `DefaultMember` attribute indicating `Item`. If you apply the `IndexerName` attribute to a parameterful property, the defining type will have a `DefaultMember` attribute indicating the string name specified in the `IndexerName` attribute. Remember, C# won't compile the code if it contains parameterful properties with different names.

For a language that supports several parameterful properties, one of the property method names must be selected and identified by the type's `DefaultMember` attribute. This is the only parameterful property that C# will be able to access.

When the C# compiler sees code that is trying to get or set an indexer, the compiler actually emits a call to one of these methods. Some programming languages might not support parameterful properties. To access a parameterful property from one of these languages, you must call the desired accessor method explicitly. To the CLR, there's no difference between parameterless properties and parameterful properties, so you use the same `System.Reflection.PropertyInfo` class to find the association between a parameterful property and its accessor methods.

The Performance of Calling Property Accessor Methods

For simple `get` and `set` accessor methods, the just-in-time (JIT) compiler *inlines* the code so that there's no run-time performance hit as a result of using properties rather than fields. Inlining is when the code for a method (or accessor method, in this case) is compiled directly in the method that is making the call. This removes the overhead associated with making a call at run time at the expense of making the compiled method's code bigger. Because property accessor methods typically contain very little code, inlining them can make the native code smaller and can make it execute faster.

Note The JIT compiler does not inline property methods when debugging code because inlined code is harder to debug. This means that the performance of accessing a property can be fast in a release build and slow in a debug build. Field access is fast in both debug and release builds.

Property Accessor Accessibility

Occasionally, when designing a type, it is desired to have one accessibility for a `get` accessor method and a different accessibility for a `set` accessor method. The most common scenario is to have a public `get` accessor and a protected `set` accessor.

```
public class SomeType {
   private String m_name;
   public String Name {
      get { return m_name; }
      protected set {m_name = value; }
   }
}
```

As you can see from the code above, the `Name` property is itself declared as a `public` property, and this means that the `get` accessor method will be public and therefore callable by all code. However, notice that the `set` accessor is declared as `protected` and will be callable only from code defined within `SomeType` or from code in a class that is derived from `SomeType`.

When defining a property with accessor methods that have different accessibilities, C# syntax requires that the property itself must be declared with the least-restrictive accessibility and that more restrictive accessibility be applied to just one of the accessor methods. In the example above, the property is `public`, and the `set` accessor is `protected` (more restrictive than `public`).

Generic Property Accessor Methods

Because properties are really just methods, and because C# and the CLR allow methods to be generic, sometimes people want to define properties that introduce their own generic type parameters (as opposed to using the enclosing type's generic type parameter). However, C# does not allow this. The main reason why properties cannot introduce their own generic type parameters is because they don't make sense conceptually. A property is supposed to represent a characteristic of an object that can be queried or set. Introducing a generic type parameter would mean that the behavior of the querying/setting could be changed, but conceptually, a property is not supposed to have behavior. If you want your object to expose some behavior—generic or not—define a method, not a property.