3

C# Object-Oriented Programming

IN THIS CHAPTER

Creating Classes

This chapter discusses object-oriented programming in C#. OOP is what C# is all about; in this chapter, we're going to specialize on this topic. You may well be an accomplished OOP programmer already, in which case it's still a good idea to scan this chapter. OOP in C# has several differences from all other object-oriented languages.

If you're an OOP programmer, you know that object-oriented programming centers on creating types. The simple type `int` lets you declare integer variables, and in the same way, you can create your own classes, which contain not only data like the simple types, but methods as well. Just as you create integer variables with the `int` type, so you create objects from classes. An integer variable is an `instance` of the `int` type, just like an object is an instance of a class.

Classes are types, but are far more powerful than the simple types like `int` and `float`. Not only can you customize your data storage using classes, but you can also add methods to classes. That kind of compartmentalization—where data and methods are rolled up into a single class—is the entire reason that OOP was introduced in the first place. It enables the programmers to deal with larger programs. The process of wrapping related data and methods into a class (and so preventing them from cluttering up the rest of the program) to create a single entity is called *encapsulation*.

You create classes in C# with the `class` statement:

```
[attributes] [modifiers] class identifier [:base-list] { class-body }[;]
```

Here are the parts of this statement:

- *`attributes`* (Optional)—Attributes hold additional declarative information, as we'll see in Chapter 14, "Using Attributes and Reflection."
- *`modifiers`* (Optional)—The allowed modifiers are `new`, `static`, `virtual`, `abstract`, `override`, and a valid combination of the four access modifiers we'll see in this chapter.
- *`identifier`*—The class name.
- *`base-list`* (Optional)—A list that contains the base class and any implemented interfaces, separated by commas.
- *`class-body`*—Declarations of the class members.

FOR C++ PROGRAMMERS

The semicolon after a class declaration is optional in C#, unlike C++.

We've been using classes since the first page of this book, as in example ch01_01.cs:

```
class ch01_01
{
  static void Main()
  {
    System.Console.WriteLine("Hello from C#.");
  }
}
```

Here, the code is all contained in one class, the ch01_01 class. But there's nothing to stop you from creating other classes in the same file. For example, here, we've added another class, `Calculator`, with one method, `Addem`, which adds two integers and returns their sum:

```
class ch03_01
{
  static void Main()
  {
    .
    .
    .
  }
}
```

```
class Calculator
{
  public long Addem(int value1, int value2)
  {
    return value1 + value2;
  }
}
```

We've created a new class now—the `Calculator` class. To put that class to use, we simply have to create an object of that class.

Creating Objects

To create an object, also called an *instance*, of a class, you use the `new` keyword. We've seen how this process works; you can see how we create a new object of the `Calculator` class in ch03_01.cs, Listing 3.1, and use that object's `Addem` method to add 2 and 3. Note the parentheses after `Calculator` in the statement `Calculator obj = new Calculator();`. These parentheses are necessary when you use the `new` keyword. They let you pass data to a class's *constructor*, the special method that lets you initialize the data in a class. We'll go into depth about constructors later in this chapter.

LISTING 3.1 Creating a New Class (ch03_01.cs)

```
class ch03_01
{
  static void Main()
  {
    Calculator obj = new Calculator();
    System.Console.WriteLine("2 + 3 = {0}", obj.Addem(2, 3));
  }
}

class Calculator
{
  public long Addem(int value1, int value2)
  {
    return value1 + value2;
  }
}
```

Here's what you see when you run this example, ch03_01:

```
C:\>ch03_01
2 + 3 = 5
```

FOR C++ PROGRAMMERS

In C#, all objects are based on the .NET `System.Object` class, and so support the properties and methods of that class.

In C#, all objects are based on the `System.Object` class, which means that they already support the methods built into that class. You can see those methods in Table 3.1. Note in particular the `ToString` method, which returns a string representation of an object. You can customize this method to return a string for your own classes by overriding this method, as we'll see in the next chapter.

TABLE 3.1
Public Methods of System.Object Objects

METHOD	MEANS
`Equals`	Indicates whether two `Object` instances are equal.
`GetHashCode`	Serves as a hash function for a particular type.
`GetType`	Returns the `Type` of the current instance.
`ReferenceEquals`	Determines whether the specified `Object` instances are the same instance.
`ToString`	Returns a `String` that represents the current `Object`.

It's worth mentioning that you can also create simple variables using `new` as well, because even those simple types are based on the `System.Object` class and have their own constructors. Here's an example:

```
int int1 = new int();
int int2 = new int(5);   //Initialize int2 to 5
```

The first statement here creates a new `int` variable named `int1`. Remember that you can't use initialized variables in C#; in this case, `int1` is automatically initialized to 0. In other words, the first statement above is logically identical to:

```
int int1 = 0;
```

You can see the default values of the various value types created when you use the new operator in Table 3.2. Note that, although you can't set reference types to a default value of 0 as you see for value types in Table 3.2, their default value is `null` (which means they do not reference an object on the heap).

TABLE 3.2
Default Values of Value Types

VALUE TYPE	DEFAULT VALUE
`bool`	false
`byte`	0
`char`	'\0'
`decimal`	0.0M
`double`	0.0D
`enum`	The value of the expression `(E)0` (where `E` is the enum identifier)
`float`	0.0F
`int`	0
`long`	0L
`sbyte`	0
`short`	0
`struct`	The value produced by setting all value-type fields to their default values and all reference-type fields to `null`
`uint`	0
`ulong`	0
`ushort`	0

Using Access Modifiers

As mentioned at the beginning of the chapter, *encapsulation*, the capability to hide data and methods to stop them from cluttering up the rest of your code, is one of the biggest advantages of OOP. Encapsulating data and methods not only ensures that they don't clutter up the rest of your code, it also ensures that the rest of your code doesn't interfere with them. You can use access modifiers to set the allowed access to not only classes, but also to all members of those classes. Here are the available access modifiers:

- The `public` keyword gives a type or type member public access, the most permissive access level. There are no restrictions on accessing public members.
- The `protected` keyword gives a type or type member protected access, which means it's accessible from within the class in which it is declared, and from within any class derived from the class that declared this member. As discussed in the next chapter, a protected member of a base class is accessible in a derived class only if the access takes place through the derived class type.
- The `internal` keyword gives a type or type member internal access, which is accessible only within files in the same assembly. It is an error to reference a member with internal access outside the assembly within which it was defined. The C++ analog is `friend`.

- The `private` keyword gives a type or type member private access, which is the least permissive access level. Private members are accessible only within the body of the class or the struct in which they are declared. It is a compile-time error to reference a private member outside the class or the struct in which it is declared.

FOR C++ PROGRAMMERS

The C++ counterpart for the C# `internal` access modifier, which restricts members to access within the same assembly, is `friend`.

The following five accessibility levels can be specified using the access modifiers: `public`, `protected`, `internal`, `internal protected`, and `private`.

Note that if you don't specify an access modifier for a type or type member, the default access modifier is `private`. For example, you might have noticed that we explicitly declared the `Addem` method `public` in the Calculator class (see Listing 3.1). We did that so we could access `Addem` using an object of the `Calculator` class in the program's main class, `ch03_01`. If we had another method, `Subtractem`, that we declared `private` in the `Calculator` class, `Subtractem` would be private to the `Calculator` class, which means that we can't access it using objects of that class. For example, this code won't compile:

```
class ch03_01
{
  static void Main()
  {
    Calculator obj = new Calculator();
    System.Console.WriteLine("2 + 3 = {0}", obj.Addem(2, 3));
    //This won't work!!
    System.Console.WriteLine("3 - 2 = {0}", obj.Subtractem(3, 2));
  }
}

class Calculator
{
  public long Addem(int value1, int value2)
  {
    return value1 + value2;
  }

  private long Subtractem(int value1, int value2)
  {
    return value1 - value2;
  }
}
```

Here's the error you'd see if you tried to compile this code:

```
ch03_01.cs(7,49): error CS0122: 'Calculator.Subtractem(int, int)'
is inaccessible due to its protection level
```

We'll see the `protected` keyword, which you use primarily with class inheritance, in the next chapter, and discuss the `internal` keyword in more depth in Chapter 13, "Understanding C# Assemblies and Security," when we discuss assemblies.

It's now time to get into some in-depth OOP as we create class members, including fields, methods, and properties.

FOR C++ PROGRAMMERS

C# doesn't support this kind of C++ class definition, where you can use one access modifier for multiple members:

```
class Customer
{
  private:
    string firstName;
    string lastName;

  public:
    string GetName() {}
    void SetName(string firstName,
string lastName) {}
    .
    .
    .
}
```

Creating Fields and Using Initializers

The first type of class member we'll take a look at is the *field*, also called a *data member*. A field is just a class-level variable, outside any method. If you make your field `public`, it's accessible using an object of your class; for example, take a look at the `Messager` class here, which has a field named `Message` that holds the message that a method named `DisplayMessage` will display:

```
class Messager
{
  public string message;

  public void DisplayMessage()
  {
    System.Console.WriteLine(message);
  }
}
```

DON'T MAKE FIELDS PUBLIC

Although this example shows how to make a field public, it's usually not good programming practice to give external code direct access to the data in your objects. Instead, it's better to use accessor methods or properties, coming up in the next two topics.

Because the `message` field is `public`, you can access it using objects of this class. For example, you can assign a string to the `message` field of a `messager` object and then call its `DisplayMessage` method to display that message, as you see in ch03_02.cs, Listing 3.2.

LISTING 3.2 Creating Fields (ch03_02.cs)

```
class ch03_02
{
  static void Main()
  {
    Messager obj = new Messager();
    obj.message = "Hello from C#.";
    obj.DisplayMessage();
  }
}

class Messager
{
  public string message = "Default message.";

  public void DisplayMessage()
  {
    System.Console.WriteLine(message);
  }
}
```

Here are the results when you run this example. As you can see, we can store data in our new object's public field `message`, which the `DisplayMessage` method of that object can use:

```
C:\>ch03_02
Hello from C#.
```

You can also initialize fields with *intializers*, which are just like the initial values you assign to variables. For example, you can use an initializer to assign the string `"Default message."` to the `Message` field like this:

```
class Messager
{
  public string message = "Default message.";

  public void DisplayMessage()
  {
    System.Console.WriteLine(message);
  }
}
```

If you don't use an initializer, fields are automatically initialized to the values in Table 3.2 for simple values, and to `null` for reference types.

You can also make fields read-only using the `readonly` keyword:

```
public readonly string message = "Default message.";
```

When a field declaration includes a `readonly` keyword, assignments to that field can occur only as part of the declaration, with an initializer, or in a constructor in the same class.

Creating Methods

We were introduced to creating methods in Chapter 2, "Basic C# Programming," but there are a few more considerations to add. First, as we saw in ch03_01.cs, Listing 3.1, methods can be declared with access modifiers, and that's important when you're creating objects. Declaring a method `public` makes it accessible outside an object, for example, whereas declaring it `private` locks it away inside the object.

Now that we're thinking in terms of objects and methods, it's also important to discuss the `this` keyword. This keyword works the same in C# as it does in C++, and it refers to the current object in which your code is executing. For example, say you're writing code to set up a button that appears in a window and you want to pass the whole button to a method named `ColorMe`. You can do that by passing the `this` keyword to that method:

```
public class Button
{
  public void SetUp()
  {
    ColorMe(this);
  }
}
```

Another use for the `this` keyword is to refer to a field in the current object. In this example, the method `SetMessage` is passed a parameter named `message`, which it is supposed to store in the field of the same name in the current object. To avoid confusion, you can refer to the field named `message` in the current object as `this.message`, like this:

```
private string message;

public SetMessage(string message)
{
  this.message = message;
}
```

This example also illustrates another use for methods in classes: creating *accessor methods*. In this case, access to the private `message` field is restricted. From outside the current object, you have to call the `SetMessage` method to assign data to this field. Restricting access to an object's internal data is usually a wise thing to do, so much so that accessor methods have been formalized in C# into *properties*.

Creating Properties

Properties are much like fields as far as code outside your object is concerned, but internally, they use accessor methods to get and set their data, giving you the chance to add code that restricts what data is written to and read from a property.

FOR C++ PROGRAMMERS

Properties do not formally exist in C++.

For example, we have a class named `Customer` in which we want to support a property called `Name`, which holds the customer's name. To store that name, we'll use a private field called `name`:

```
class Customer
{
  private string name;
    .
    .
    .
}
```

To implement a property, you set up `get` and `set` accessor methods; the `get` method returns the property's value, and the `set` method sets it. Here's how to implement the `get` accessor method for this property, which simply returns the customer's name:

```
class Customer
{
  private string name;

  public string Name
  {
   get
   {
     return name;
   }
    .
    .
```

```
      .
  }
}
```

This example just returns the customer's name, but you can place whatever code you wanted here to process data as needed before returning that data (such as converting Fahrenheit temperatures into Centigrade), and that's what makes properties so powerful. In the `set` accessor method, you're passed the new value of the property in a parameter named `value`, which we'll store like this:

```
class Customer
{
  private string name;

  public string Name
  {
   get
   {
     return name;
   }
   set
   {
     name = value;
   }
  }
}
```

Now we can create objects of the `Customer` class, as well as set and get values using the `Name` property. You can see that at work in ch03_03.cs, Listing 3.3.

LISTING 3.3 Creating a Property (ch03_03.cs)

```
class ch03_03
{
  static void Main()
  {
    Customer customer = new Customer();
    customer.Name = "Nancy";
    System.Console.WriteLine("The customer's name is {0}",
      customer.Name);
  }
}
```

LISTING 3.3 Continued

```
class Customer
{
  private string name;

  public string Name
  {
   get
   {
     return name;
   }
   set
   {
     name = value;
   }
  }
}
```

Here's what you see when you run ch03_03.cs. We have been able to set and retrieve a value with the new property:

```
C:\>ch03_03
The customer's name is Nancy
```

Property declarations take one of the following forms:

```
[attributes] [modifiers] type identifier {accessor-declaration}
[attributes] [modifiers] type interface-type.identifier {accessor-declaration}
```

Here are the parts of this statement:

- *`attributes`* (Optional)—Hold additional declarative information, as we'll see in Chapter 14.
- *`modifiers`* (Optional)—The allowed modifiers are `new`, `static`, `virtual`, `abstract`, `override`, and a valid combination of the four access modifiers.
- *`type`*—The property type, which must be at least as accessible as the property itself.
- *`identifier`*—The property name.
- *`accessor-declaration`*—Declaration of the property accessors, which are used to read and write the property.
- *`interface-type`*—The interface in a fully qualified property name.

Note that unlike a class's fields, properties are not considered variables, which means it's not possible to pass a property as a `ref` or `out` parameter.

Read-only Properties

You can also create read-only properties if you omit the `set` accessor method. For example, to make the `Name` property a read-only property, you use this code:

```
class Customer
{
  private string name;

  public string Name
  {
    get
    {
      return name;
    }
  }
}
```

Creating Constructors

We've created objects using `new` like this: `Customer customer = new Customer();`. If you're an OOP programmer, you know those parentheses after `Customer` are there for a reason, because when you create an object from a class, you're using the class's *constructor*. A constructor is a special method that has the same name as the class and returns no value. It is used to initialize the data in the object you're creating. In C#, constructors are declared this way:

```
[attributes] [modifiers] identifier([formal-parameter-list])
[initializer] { constructor-body }
```

Here are the parts of this statement:

- *`attributes`* (Optional)—Hold additional declarative information, as we'll see in Chapter 14.
- *`modifiers`* (Optional)—The allowed modifiers are `new`, `static`, `virtual`, `abstract`, `override`, and a valid combination of the four access modifiers.
- *`identifier`*—The same as the class name.

- *`formal-parameter-list`* (Optional)—The optional parameters passed to the constructor. The parameters must be as accessible as the constructor itself.
- *`initializer`* (Optional)—Invoked before the execution of the constructor body. The *`initializer`* can be one of the following with an optional *`argument-list`*:

  ```
  : base (argument-list)
  : this (argument-list)
  ```

- *`constructor-body`*—The block that contains the statements that initialize the object.

For example, we can add a constructor to the `Customer` class so that you can initialize the customer's name when you create an object of that class. Listing 3.4 shows what that looks like. Note that the constructor has the same name as the class itself, and that we pass the constructor the name `"Paul"`.

LISTING 3.4 Creating a Constructor (ch03_04.cs)

```
class ch03_04
{
  static void Main()
  {
    Customer customer = new Customer("Paul");
    System.Console.WriteLine("The customer's name is {0}",
      customer.Name);
  }
}

class Customer
{
  private string name;

  public Customer(string name)
  {
    this.name = name;
  }

  public string Name
  {
   get
   {
     return name;
   }
  }
}
```

Here's what you see when you run this code. Note that the name we passed to the constructor was indeed stored in the `Customer` object and was used:

```
C:\>ch03_04
The customer's name is Paul
```

If you don't declare a constructor, a default version without parameters is created automatically. (Note that if you do create *any* kind of a constructor, even a private one—which won't allow objects to be made from a class—C# will not create a default constructor.) In this default constructor, all the fields in your class are set to their default values (see Table 3.2).

FOR C++ PROGRAMMERS

C# doesn't support member initialization lists in constructors that follow a colon like this: `Customer (string name) : internalName(name){}`. Note also that there is no C++ counterpart for calling other constructors with the `this` keyword.

You can also have one constructor call another using the `this` keyword, as here, where a parameterless constructor is calling the constructor we just created and passing it the name `"George"`:

```
public Customer() : this("George"){}
```

Creating Copy Constructors

There's another kind of constructor—the *copy constructor*. When you copy one object to another, C# will copy the reference to the first object to the new object, which means that you now have two references to the *same* object. To make an actual copy, you can use a copy constructor, which is just a standard constructor that takes an object of the current class as its single parameter. For example, here's what a copy constructor for the `Customer` class might look like. Note that we're copying the name field to the new object:

```
public Customer(Customer customer)
{
  this.name = customer.name;
}
```

Now you can use this constructor to create copies. The copy will be a separate object, not just a reference to the original object. You can see this at work in ch03_05.cs, Listing 3.5. In that code, we create an object named `customer` with the name `"Paul"`, and then copy it over to an object named `customerCopy`. Next, we change the name in `customer` to `"Sam"`, and, to make sure `customerCopy` doesn't refer to the same data as `customer`, display the name in `customerCopy`—which is still `"Paul"`.

LISTING 3.5 Creating a Copy Constructor (ch03_05.cs)

```
class ch03_05
{
  static void Main()
  {
    Customer customer = new Customer("Paul");
    Customer customerCopy = new Customer(customer);
    customer.Name = "Sam";
    System.Console.WriteLine("The new customer's name is {0}",
      customerCopy.Name);
  }
}

class Customer
{
  private string name;

  public Customer(string name)
  {
    this.name = name;
  }

  public Customer(Customer customer)
  {
    this.name = customer.name;
  }

  public string Name
  {
   get
   {
     return name;
   }
   set
   {
     name = value;
   }
  }
}
```

Here's what you see when you run ch03_05.cs. As you can see, `customerCopy` does not refer to the same object as `customer`:

```
C:\>ch03_05
The new customer's name is Paul
```

Creating Structs

C# also supports *structs*, which you create with the `struct` keyword. Structs in C# are like lightweight versions of classes. They're not reference types; they're value types, so when you pass them to methods, they're passed by value. They're like classes in many ways—they support constructors, for example (but not inheritance). They take up fewer resources in memory, so when you've got a small, frequently used class, give some thought to using a struct instead. Here's how you declare a `struct`:

```
[attributes] [modifiers] struct identifier [:interfaces] body [;]
```

Here are the parts of this statement:

- *`attributes`* (Optional)—Hold additional declarative information, as we'll see in Chapter 14.
- *`modifiers`* (Optional)—The allowed modifiers are `new`, `static`, `virtual`, `abstract`, `override`, and a valid combination of the four access modifiers.
- *`identifier`*—The `struct` name.
- *`interfaces`* (Optional)—A list that contains the interfaces implemented by the struct, all separated by commas.
- *`body`*—The struct body that contains member declarations.

FOR C++ PROGRAMMERS

In C++, structures are like classes, with only a few minor differences. In C#, structs are value types, not reference types, and cannot support inheritance, initializers, or destructors.

Because structs are value types, you can create them without using `new` (although you can also use new and pass arguments to a struct's constructor).

Consider the `Complex` struct, which holds complex numbers. Complex numbers have both real and imaginary parts (the imaginary part is multiplied by the square root of 1). For example, in the complex number 1 + 2i, the real part is 1 and the imaginary part is 2. In this struct, we'll implement the public fields `real` and `imaginary`, as well as a constructor and a method named `Magnitude`, to return the magnitude of this complex number (which will use the `System.Math.Sqrt` method to calculate a square root):

```
struct Complex
{
  public int real, imaginary;

  public Complex(int real, int imaginary)
  {
    this.real = real;
    this.imaginary = imaginary;
  }

  public double Magnitude()
  {
    return System.Math.Sqrt(real * real +
      imaginary * imaginary);
  }
}
```

Now you can create new complex numbers without using `new`, just as you would for any value type. You can see an example in ch03_06.cs, Listing 3.6, where we create the complex number 3 + 4i (where i is the square root of -1), and display this number's magnitude.

LISTING 3.6 Creating a struct (ch03_06.cs)

```
class ch03_06
{
  static void Main()
  {
    Complex complexNumber;
    complexNumber.real = 3;
    complexNumber.imaginary = 4;
    System.Console.WriteLine("Maginitude: {0}",
      complexNumber.Magnitude());
  }
}

struct Complex
{
  public int real, imaginary;

  public Complex(int real, int imaginary)
  {
    this.real = real;
    this.imaginary = imaginary;
```

LISTING 3.6 Continued

```
    }

    public double Magnitude()
    {
      return System.Math.Sqrt(real * real +
        imaginary * imaginary);
    }
  }
```

When you run this example, this is what you see:

```
C:\>ch03_06
Maginitude: 5
```

Creating Static Members

So far, the fields, methods, and properties we've been creating have all been members of objects. You can also create fields, methods, and properties that you use with classes directly, not with objects. These members are called *static members*, also called *class members* (not object members), and you use them with the class name, not the name of an object.

As you know, `Main` is declared static so that C# itself can call it without creating an object from your main class. That's the way it works with all static members—they're intended to be used with the class, not with an object. We'll take a look at creating static members now, starting with static fields.

FOR C++ PROGRAMMERS

Here's a big difference between C# and C++: in C#, you cannot access static members of a class using an object of that class, as you can in C++. You must use the class name to access static members, not an object.

Creating Static Fields

A static field is a class field, which means that its data is stored in one location in memory, no matter how many objects are created from its class. In fact, you can use a static field to keep track of how many objects have been created from a particular class. To do that, you might declare a static field, which we'll name `numberOfObjects` in the example, and increment that field each time the class's constructor is called (an explicit initializer is required for `static` fields, so we've assigned `numberOfObjects` the value 0 here):

```
public class CountedClass
{
  public static int numberOfObjects = 0;

  public CountedClass()
  {
    numberOfObjects++;
  }
}
```

Now each time a new object of this class is created, `numberOfObjects` is incremented. You can see how this works in ch03_07.cs, Listing 3.7, where we create three new objects and display the value of the class's `numberOfObjects` field each time.

LISTING 3.7 Creating a Static Field (ch03_07.cs)

```
class ch03_07
{
  static void Main()
  {
    System.Console.WriteLine("Number of objects: {0}",
      CountedClass.numberOfObjects);
    CountedClass object1 = new CountedClass();
    System.Console.WriteLine("Number of objects: {0}",
      CountedClass.numberOfObjects);
    CountedClass object2 = new CountedClass();
    System.Console.WriteLine("Number of objects: {0}",
      CountedClass.numberOfObjects);
    CountedClass object3 = new CountedClass();
    System.Console.WriteLine("Number of objects: {0}",
      CountedClass.numberOfObjects);
  }
}

public class CountedClass
{
  public static int numberOfObjects = 0;

  public CountedClass()
  {
    numberOfObjects++;
  }
}
```

Here's what you see when you run ch03_07.cs. As you can see, the class field `numberOfObjects` was incremented each time we created a new object:

```
C:\>ch03_07
Number of objects: 0
Number of objects: 1
Number of objects: 2
Number of objects: 3
```

Although for the sake of brevity in this example, we made `numberOfObjects` a public static field, it's usually not a good idea to make class fields public. Instead, you can use an accessor method, but if you do, make sure that it's declared `static`, as are the methods we'll see next.

Creating Static Methods

As we've discussed, when you have a `static` method, you don't need an object to call that method; you use the class name directly. You may have noticed, for example, that we used the `System.Math.Sqrt` method to find square roots and calculate the value of a complex number's magnitude in Listing 3.6; this is a `static` method of the `System.Math` class, and it's made `static` so you don't have to go to the trouble of creating a `System.Math` object before calling that method.

Let's see an example in which we can create our own `static` method. Although the `System.Math` class has a handy `Sqrt` method for calculating square roots, it doesn't have a method for calculating quad roots (fourth roots—the square root of a value's square root). We can correct that glaring omission by creating a new class, `Math2`, with a static method named `QuadRt`, which returns quad roots like this:

```
public class Math2
{
  public static double QuadRt(double value)
  {
    return System.Math.Sqrt(System.Math.Sqrt(value));
  }
}
```

Now you can use the `QuadRt` method with the `Math2` class directly, no object needed, as you see in ch03_08.cs, Listing 3.8.

LISTING 3.8 Creating a Static Method (ch03_08.cs)

```
class ch03_08
{
  static void Main()
  {
    System.Console.WriteLine("The quad root of 81 is {0}",
      Math2.QuadRt(81));
  }
}

public class Math2
{
  public static double QuadRt(double value)
  {
    return System.Math.Sqrt(System.Math.Sqrt(value));
  }
}
```

Here's what you see when you run ch03_08.cs:

```
C:\>ch03_08
The quad root of 81 is 3
```

Note that because static methods are not part of an object, you cannot use the `this` keyword in such methods. It's important to know that static methods cannot directly access non-static members (which means, for example, that you cannot call a non-static method from a static method). Instead, they must instantiate an object and use the members of that object to access non-static members (as we saw when creating methods called from `Main` in Chapter 2).

SHOP TALK

CLASSES WITH ONLY STATIC MEMBERS

As part of programming teams, I've sometimes had to create utility classes, like the `System.Math` class, which only have static members. In that case, it's best not to allow objects to be created from that class (for one thing, in C#, none of your static members could be called such an object anyway). In these cases, I gave the utility classes a `private` constructor, which stops the creation of a default constructor, making sure no objects can be created from the class. When you're creating code for general distribution, it's always wise to think of worst-case scenarios. If your code can be used the wrong way, someone will do it.

Creating Static Constructors

You can also make constructors static, like this:

```
public class CountedClass
{
  public static CountedClass()
  {
    .
    .
    .
  }
}
```

Static constructors are called before any objects are created from your class, so you can use them to initialize the data in standard constructors, if you like. C# makes no promise when a static constructor is called. All you know is that it'll be called sometime between the beginning of the program and before an object is created from your class.

Static constructors are also called before any static members in your class are referenced, so you can use them to initialize a class (not just an object). Note that static constructors do not take access modifiers or have parameters.

Creating Static Properties

Like fields, properties can also be static, which means you don't need an object to use them. As an example, we'll convert the static field `numberOfObjects` in the static fields example (see Listing 3.7) into a property of the same name. That's easy enough—all we have to do is to implement the new property like this (note that the value of the property is stored in a `private` field, which must be `static` so we can access it from the `static` property):

```
public class CountedClass
{
  private static int number = 0;

  public CountedClass()
  {
    number++;
  }

  public static int NumberOfObjects
  {
   get
   {
      return number;
    }
    set
    {
      number = value;
    }
  }
}
```

Now we can use this new static property as we used the static field earlier, as you see in ch03_09.cs, Listing 3.9.

LISTING 3.9 Creating a Static Property (ch03_09.cs)

```
class ch03_09
{
  static void Main()
  {
    System.Console.WriteLine("Number of objects: {0}",
      CountedClass.NumberOfObjects);
    CountedClass object1 = new CountedClass();
    System.Console.WriteLine("Number of objects: {0}",
      CountedClass.NumberOfObjects);
    CountedClass object2 = new CountedClass();
    System.Console.WriteLine("Number of objects: {0}",
      CountedClass.NumberOfObjects);
    CountedClass object3 = new CountedClass();
    System.Console.WriteLine("Number of objects: {0}",
      CountedClass.NumberOfObjects);
  }
}

public class CountedClass
{
  private static int number = 0;

  public CountedClass()
  {
    number++;
  }

  public static int NumberOfObjects
  {
```

LISTING 3.9 Continued

```
    get
    {
      return number;
    }
    set
    {
      number = value;
    }
  }
}
```

Creating Destructors and Handling Garbage Collection

The flip side of constructors are *destructors*, which are called when it's time to get rid of an object and perform cleanup, such as disconnecting from the Internet or closing files. Getting rid of no-longer-needed objects involves the C# *garbage collector*, which calls your destructor. The C# garbage collector is far more sophisticated than the ones available in most C++ implementations, as we'll see soon. Let's start with destructors first.

Creating a Destructor

You place the code you want to use to clean up an object when it's being deleted, if any, in a destructor. Destructors cannot be static, cannot be inherited, do not take parameters, and do not use access modifiers. They're declared much like constructors, except that their name is prefaced with a tilda (~). You can see an example in ch03_10.cs, which appears in Listing 3.10, and includes a destructor for the `Customer` class. When there are no more references to objects of this class in your code, the garbage collector is notified and will call the object's destructor (sooner or later—you can't predict when it will happen).

LISTING 3.10 Creating a Destructor (ch03_10.cs)

```
class ch03_10
{
  static void Main()
  {
    Customer customer = new Customer();
  }
}
```

LISTING 3.10 Continued

```
public class Customer
{
  public Customer()
  {
    System.Console.WriteLine("In Customer's constructor.");
  }

  ~Customer()
  {
    System.Console.WriteLine("In Customer's destructor.");
  }
}
```

Here's what you see when you run ch03_10.cs:

```
C:\>ch03_10
In Customer's constructor.
In Customer's destructor.
```

In C#, destructors are converted into a call to the `System.Object` class's `Finalize` method (which you cannot call directly in C#). In other words, this destructor:

```
~Customer()
{
  // Your code
}
```

is actually translated into this code by C# (the call to `base.Finalize` calls the `Finalize` method in the class the current class is based on; more on the `base` keyword in the next chapter):

```
protected override void Finalize()
{
  try
  {
    // Your code
  }
  finally
  {
   base.Finalize();
  }
}
```

However, in C#, you must use the standard destructor syntax as we've done in Listing 3.10, and not call or use the `Finalize` method directly.

Understanding Garbage Collection

Destructors are actually called by the C# *garbage collector*, which manages memory for you in C# code. When you create new objects with the `new` keyword in C++, you should also later use the `delete` keyword to delete those objects and free up their allocated memory. That's no longer necessary in C#—the garbage collector deletes objects no longer referenced in your code automatically (at a time of its choosing).

What that means in practical terms is that you don't have to worry about standard objects in your code as far as memory usage goes. On the other hand, when you use "unmanaged" resources like windows, files, or network connections, you should write a destructor that can be called by the garbage collector, and put the code you want to use to close or deallocate those resources in the destructor (because the garbage collector won't know how to do that itself).

FOR C++ PROGRAMMERS

In C#, you no longer have to use `delete` to explicitly delete objects after allocating them with `new`. The garbage collector will delete objects for you by itself. In fact, `delete` is not even a keyword in C#.

Implementing a Dispose Method

On some occasions, you might want to do more than simply put code in a destructor to clean up resources. You might want to disconnect from the Internet or close a file as soon as you no longer need those resources. For such cases, C# recommends that you create a `Dispose` method. Because that method is public, your code can call it directly to dispose of the object that works with important resources you want to close.

In a `Dispose` method, you clean up after an object yourself, without C#'s help. Dispose can be called both explicitly, and/or by the code in your destructor. Note that if you call the `Dispose` method yourself, you should suppress garbage collection on the object once it's gone, as we'll do here.

FOR C++ PROGRAMMERS

In C#, you can implement a `Dispose` method to explicitly deallocate resources.

Let's take a look at an example to see how this works. The `Dispose` method is formalized in C# as part of the `IDisposable` interface (we'll discuss interfaces in the next chapter). That has no real meaning for our code, except that we will indicate we're implementing the `IDisposable` interface. In our example, the `Dispose` method will clean up an object of a class we'll call `Worker`.

Note that if you call `Dispose` explicitly, you can access other objects from inside that method; however, you can't do so if `Dispose` is called from the destructor, because those objects might already have been destroyed. For that reason, we'll track whether `Dispose` is being called explicitly or by the destructor by passing `Dispose` a `bool` variable (set to `true` if we call `Dispose`, and `false` if the destructor calls `Dispose`). If that `bool` is true, we can access other objects in `Dispose`—otherwise, we can't.

Also, if the object is disposed of by calling the `Dispose` method, we don't want the garbage collector to try to dispose of it as well, which means we'll call the `System.GC.SuppressFinalize` method to make sure the garbage collector suppresses garbage collection for the object. You can see what this code looks like in ch03.11.cs, Listing 3.11.

LISTING 3.11 Implementing the Dispose Method (ch03_11.cs)

```
class ch03_11
{
  static void Main()
  {
    Worker worker = new Worker();
    worker.Dispose();
  }
}

public class Worker: System.IDisposable
{
  private bool alreadyDisposed = false;

  public Worker()
  {
   System.Console.WriteLine("In the constructor.");
  }

  public void Dispose(bool explicitCall)
  {
   if(!this.alreadyDisposed)
   {
     if(explicitCall)
     {
      System.Console.WriteLine("Not in the destructor, " +
        "so cleaning up other objects.");
      // Not in the destructor, so we can reference other objects.
      //OtherObject1.Dispose();
      //OtherObject2.Dispose();
```

LISTING 3.11 Continued

```
      }
      // Perform standard cleanup here...
      System.Console.WriteLine("Cleaning up.");
    }
    alreadyDisposed = true;
   }

   public void Dispose()
   {
    Dispose(true);
    System.GC.SuppressFinalize(this);
   }

   ~Worker()
   {
    System.Console.WriteLine("In the destructor now.");
    Dispose(false);
   }
}
```

Here's what you see when you run ch03_11.cs. Note that the object was only disposed of once, when we called `Dispose` ourselves, and not by the destructor, because the object had already been disposed of:

```
C:\>ch03_11
In the constructor.
Not in the destructor, so cleaning up other objects.
Cleaning up.
```

Here's another thing to know if you're going to use `Dispose`—you don't have to call this method explicitly if you don't want to. You can let C# call it for you if you add the `using` keyword (this is not the same as the `using` directive, which imports a namespace). After control leaves the code in the curly braces following the `using` keyword, `Dispose` will be called automatically. Here's how this keyword works:

```
using (expression | type identifier = initializer){}
```

RENAMING DISPOSE

Sometimes, calling the `Dispose` method `Dispose` isn't appropriate. On some occasions, for example, `Close` or `Deallocate` might be a better name. C# still recommends you stick with the standard `Dispose` name, however, so if you do write a `Close` or `Deallocate` method, you can have it call `Dispose`.

Here are the parts of this statement:

- `expression`—An expression you want to call `Dispose` on when control leaves the `using` statement.
- `type`—The type of identifier.
- `identifier`—The name, or identifier, of the type `type`.
- `initializer`—An expression that creates an object.

Here's an example, where `Dispose` will be called on the `Worker` object when control leaves the `using` statement:

```
class usingExample
{
  public static void Main()
  {
    using (Worker worker = new Worker())
    {
     // Use worker object in code here.
    }
    // C# will call Dispose on worker here.
  {
}
```

Forcing a Garbage Collection

Here's one last thing to know about C# garbage collection—although the C# documentation (and many C# books) says you can't know when garbage collection will take place, that's not actually true. In fact, you can force garbage collection to occur simply by calling the `System.GC.Collect` method.

This method is safe to call, but if you do, you should know that execution speed of your code might be somewhat compromised.

Overloading Methods

A major topic in OOP is overloading methods, which lets you define the same method multiple times so that you can call them with different argument lists (a method's argument list is called its *signature*). C# not only supports method overloading, but revels in it. Most of the methods you'll find built into C# have several overloaded forms to make life easier for you—for example, you can call `System.Console.WriteLine` with a `string`, a `float`, an `int`, and so on, all because it's been overloaded to handle those types of arguments (it has 18 overloaded forms in all).

It's easy to overload a method; just define it multiple times, each time with a unique signature. You can see an example in ch03_12.cs, Listing 3.12, where we're working with our `Math2` class again, adding an overloaded method named `Area` that can return the area of both squares and rectangles.

You can call `Area` with either one or two arguments. If you call this method with one argument, `a`, it'll assume you want the area of a square whose sides are `a` long. If you call this method with two arguments, `a` and `b`, the method will assume you want the area of a rectangle which is `a` long and `b` high. In other words, the code can determine what method to call based on the methods' signatures.

LISTING 3.12 Overloading a Method (ch03_12.cs)

```
class ch03_12
{
  static void Main()
  {
    System.Console.WriteLine("Here's the area of the square: {0}",
      Math2.Area(10));
    System.Console.WriteLine("Here's the area of the rectangle: {0}",
      Math2.Area(10, 5));
  }
}

public class Math2
{
  // This one's for squares
  public static double Area(double side)
  {
    return side * side;
  }

  // This one's for rectangles
  public static double Area(double length, double height)
  {
    return length * height;
  }
}
```

Here's what you see when you run ch03_12.cs. As you can see, we've overloaded the method successfully:

```
C:\>ch03_12
Here's the area of the square: 100
Here's the area of the rectangle: 50
```

Overloading Constructors

You can overload constructors just like any other method. For example, we could give the `Messager` class two constructors. (The `Messager` class appears in Listing 3.2.) One of these will take a string argument which holds the text the `Messager` object's `DisplayMessage` method should display, and the other constructor takes no arguments and simply stores a default string for use with `DisplayMessage`. Here's what it looks like:

```
public Messager()
{
  this.message = "Hello from C#.";
}

public Messager(string message)
{
  this.message = message;
}
```

You can see these two constructors at work in ch03_13.cs, Listing 3.13. In that code, we create `Messager` objects using both constructors, and then display the messages in those objects.

LISTING 3.13 Overloading a Constructor (ch03_13.cs)

```
class ch03_13
{
  static void Main()
  {
    Messager obj1 = new Messager();
    obj1.DisplayMessage();
    Messager obj2 = new Messager("No worries.");
    obj2.DisplayMessage();
  }
}
```

LISTING 3.13 Continued

```
class Messager
{
  public string message = "Default message.";

  public Messager()
  {
    this.message = "Hello from C#.";
  }

  public Messager(string message)
  {
    this.message = message;
  }

  public void DisplayMessage()
  {
    System.Console.WriteLine(message);
  }
}
```

Here's what you see when you run ch03_13.cs. As you can see, we've used both constructors, the one that takes no parameters, and the one that takes a single string parameter:

```
C:\>ch03_13
Hello from C#.
No worries.
```

Overloading Operators

If you're an OOP programmer, you know that you can also overload operators, not just methods. You do that by defining `static` methods using the `operator` keyword. Being able to overload operators like `+`, `-`, `*` and so on for your own classes and structs lets you use those classes and structs with those operators, just as if they were types built into C#. C# doesn't allow as many operators to be overloaded as C++ does. You can see the possibilities for C# in Table 3.3. Note the division into *unary* operators and *binary* operators—unary operators take one operand (like the negation operator, `-x`), and binary operators take two operands (like the addition operator, `x + y`).

TABLE 3.3
Overloading Possibilities for C# Operators

OPERATORS	OVERLOADING POSSIBILITIES
`+, -, !, ~, ++, --, true, false`	These unary operators can be overloaded.
`+, -, *, /, %, &, |, ^, <<, >>`	These binary operators can be overloaded.
`==, !=, <, >, <=, >=`	The comparison operators can be overloaded.
`&&, ||`	The conditional logical operators cannot be overloaded, but they are computed with `&` and `|`, which can be overloaded.
`[]`	The array indexing operator cannot be overloaded, but you can define indexers in C# (see Chapter 6, "Understanding Collections and Indexers").
`()`	The cast operator cannot be overloaded directly, but you can define your own conversion operators, as you'll do in this chapter.
`+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`	Assignment operators cannot be overloaded, but if you overload a binary operator, such as `+`, `+=` is also overloaded.
`=, ., ?:, ->, new, is, sizeof, typeof`	These operators cannot be overloaded.

Note also that, unlike C++, the = assignment operator cannot be overloaded in C#. An assignment always performs a simple bit-by-bit copy of a value into a variable. On the other hand, when you overload a binary operator like +, the corresponding compound assignment operator, +=, is automatically overloaded. Cast operations are overloaded by providing conversion methods, as we'll see in a page or two.

FOR C++ PROGRAMMERS

Unlike C++, you cannot overload the `=`, `()`, `[]`, `&&`, `||`, and `new` operators in C#.

You can overload operators for either classes or structs. To see how this works, we'll overload the `Complex` struct we built earlier in the chapter (see Listing 3.6). This struct holds complex numbers like 1 + 2i, where i is the square root of -1, and we'll see how to overload operators like + so that we can add two `Complex` objects, or the unary negation operator so that if `complex` holds 1 + 2i, `-complex` will yield -1 - 2i. All this takes place in ch03_14.cs, which appears in Listing 3.14. We'll take this code apart in the next few sections.

LISTING 3.14 Overloading Operators (ch03_14.cs)

```
class ch03_14
{
  public static void Main()
  {
    Complex complex1 = new Complex(1, 2);
    Complex complex2 = new Complex(3, 4);
```

LISTING 3.14 Continued

```
      System.Console.WriteLine("complex1 = {0}", complex1);
      System.Console.WriteLine("complex2 = {0}", complex2);

      Complex complex3 = -complex1;
      System.Console.WriteLine("-complex1 = {0}", complex3);

      System.Console.WriteLine("complex1 + complex2 = {0}",
        complex1 + complex2);

      if(complex1 == complex2){
        System.Console.WriteLine("complex1 equals complex2");
      } else {
        System.Console.WriteLine("complex1 does not equal complex2");
      }
    }
  }

  public struct Complex
  {
    public int real;
    public int imaginary;

    public Complex(int real, int imaginary)
    {
      this.real = real;
      this.imaginary = imaginary;
    }

    public override string ToString()
    {
      if (imaginary >= 0){
        return(System.String.Format("{0} + {1}i", real, imaginary));
      } else {
        return(System.String.Format("{0} - {1}i", real,
        System.Math.Abs(imaginary)));
      }
    }

    public static Complex operator-(Complex complex)
    {
```

LISTING 3.14 Continued

```
      return new Complex(-complex.real, -complex.imaginary);
    }

    public static Complex operator+(Complex complex1, Complex complex2)
    {
      return new Complex(complex1.real + complex2.real,
        complex1.imaginary + complex2.imaginary);
    }

    public static implicit operator Complex(int theInt)
    {
      return new Complex(theInt, 0);
    }

    public static explicit operator int(Complex complex)
    {
      return complex.real;
    }

    public static bool operator==(Complex complex1, Complex complex2)
    {
      if (complex1.real == complex2.real &&
        complex1.imaginary == complex2.imaginary)
      {
        return true;
      }
      return false;
    }

    public static bool operator!=(Complex complex1, Complex complex2)
    {
      return !(complex1 == complex2);
    }

    public override bool Equals(object obj)
    {
      if (!(obj is Complex))
      {
        return false;
      }
      return this == (Complex) obj;
    }
```

LISTING 3.14 Continued

```
  public override int GetHashCode()
  {
     return (int) System.Math.Sqrt(real * real +
       imaginary * imaginary);
  }
}
```

Creating the Complex Struct

We start ch03_14.cs by using the `Complex` struct we saw earlier in this chapter, which has a constructor you pass the real and imaginary parts to, and we'll add the `ToString` method. Any time C# needs a string representation of a complex number (as when you pass it to `System.Console.WriteLine`), it'll call the number's `ToString` method:

```
public struct Complex
{
  public int real;
  public int imaginary;

  public Complex(int real, int imaginary)
  {
   this.real = real;
   this.imaginary = imaginary;
  }

  public override string ToString()
  {
   if (imaginary >= 0){
     return(System.String.Format("{0} + {1}i", real, imaginary));
   } else {
     return(System.String.Format("{0} - {1}i", real,
     System.Math.Abs(imaginary)));
   }
  }
  .
  .
  .
}
```

USING SYSTEM.STRING.FORMAT

Note the use of `System.String.Format` in the previous code. This method works just like `System.Console.WriteLine`, except that it returns a string instead of displaying text to the console. That means we can use it to embed the real and imaginary parts of the complex number into a string.

Overloading a Unary Operator

The next step is to start overloading operators for `Complex` numbers. We'll start by overloading the unary negation operator, `-`. To do that, you add this method to the `Complex` struct, which uses the `operator` keyword and is passed a `Complex` number to negate:

```
public static Complex operator-(Complex complex)
{
  return new Complex(-complex.real, -complex.imaginary);
}
```

All we have to do here is to negate the real and imaginary parts of the complex number and return the result, as you see in this code. Now if you created a complex number, 1 + 2i, and negated it like this:

```
Complex complex1 = new Complex(1, 2);
System.Console.WriteLine(-complex1;
```

You'd see this result:

```
-1 - 2i
```

Overloading a Binary Operator

We've been able to overload the `-` unary operator for complex numbers by adding a static method to the `Complex` struct that uses the `operator` keyword and is passed the operand to negate. When you're overloading a binary operator, like the + addition operator, you are passed two operands; in the case of the + operator, those are the complex numbers you're supposed to add. Here's the method you'd add to the `Complex` struct to overload the + operator for complex numbers:

```
public static Complex operator+(Complex complex1, Complex complex2)
{
  return new Complex(complex1.real + complex2.real,
    complex1.imaginary + complex2.imaginary);
}
```

Now if you were to use this code to add 1 + 2i and 3 + 4i:

```
Complex complex1 = new Complex(1, 2);
Complex complex2 = new Complex(3, 4);

System.Console.WriteLine("complex1 + complex2 = {0}",
  complex1 + complex2);
```

you would see this result:

```
complex1 + complex2 = 4 + 6i
```

Overloading Conversion Operations

FOR C++ PROGRAMMERS

It's not possible to create nonstatic operators overloads in C#, so binary operators must take two operands.

You can also overload conversion operations. Conversions can be either implicit or explicit, and you use the `implicit` or `explicit` keywords in those cases. The name of the operator in this case is the target type you're converting to, and the parameter you're passed is of the type you're converting from. For example, here's how to convert from an integer value to a `Complex` number—note that we'll just assign the integer value to the real part of the resulting complex number. Because data will be lost, we'll make this an implicit conversion:

```
public static implicit operator Complex(int intValue)
{
  return new Complex(intValue, 0);
}
```

On the other hand, converting from a complex number to an `int` does imply some data loss, so we'll make this an explicit conversion:

```
public static explicit operator int(Complex complex)
{
  return complex.real;
}
```

Now when you cast from `Complex` to `int` explicitly, this method will be called.

Overloading Equality Operators

Overloading the == equality operator is like overloading any binary operator, with a few differences. For one, if you overload ==, C# will insist that you overload `!=` as well, so we'll do both operators here.

FOR C++ PROGRAMMERS

If you overload ==, C# will insist that you also overload !=.

When you overload the == operator, you're passed two Complex objects to compare; you return true if they're equal and false otherwise. Complex numbers are equal if their real and imaginary parts are equal, so here's how to overload the == operator for complex numbers:

```
public static bool operator==(Complex complex1, Complex complex2)
{
  if (complex1.real == complex2.real &&
    complex1.imaginary == complex2.imaginary)
  {
    return true;
  }
   return false;
}
```

And here's how to overload !=:

```
public static bool operator!=(Complex complex1, Complex complex2)
{
   return !(complex1 == complex2);
}
```

If you only overload == and !=, C# will give you a warning when you compile your code that you haven't overridden the Object.Equals(object o) method. This method is sometimes used by code instead of the == operator to check for equality (in Visual Basic .NET, for example, you can't overload operators, so code would use the Equals method). For example, to check if complex1 equals complex2, you could call complex1.Equals(complex2). C# wants us to *override* this method, replacing the default version in the Object class, not overload it, and we'll discuss overriding methods in the next chapter. All that means in this case is that we use the override keyword here. After using the is operator to ensure that the object passed to us is a Complex object, we just compare the current object to the one passed, and return the result of that comparison, like this:

```
public override bool Equals(object obj)
{
  if (!(obj is Complex))
  {
    return false;
  }
  return this == (Complex) obj;
}
```

But there's still more to do. If you've overloaded ==,!=, and `Equals`, C# will still give you another warning. You haven't overridden the `Object.GetHashCode` method. A hash method is used to quickly generate a hash code, which is an `int` that corresponds to the value of an object. Hash codes allow C# to store objects more efficiently in collections, as we'll discuss in Chapter 6. You don't have to override `GetHashCode`—you can simply ignore the warning. In this case, we'll return the magnitude of the complex number as its hash code:

```
public override int GetHashCode()
{
  return (int) System.Math.Sqrt(real * real +
    imaginary * imaginary);
}
```

Now, at last, you can compare two complex numbers using the == operator, like this, which compares 1 + 2i and 3 + 4i:

```
Complex complex1 = new Complex(1, 2);
Complex complex2 = new Complex(3, 4);

if(complex1 == complex2){
  System.Console.WriteLine("complex1 equals complex2");
} else {
  System.Console.WriteLine("complex1 does not equal complex2");
}
```

Here's what this code produces:

```
complex1 does not equal complex2
```

For the full story on operator overloading, run ch03_14.cs; this example implements all the operator overloads we've discussed and puts them to work using the code we've developed. Here's what you see when you run this example:

```
C:\>ch03_14
complex1 = 1 + 2i
complex2 = 3 + 4i
-complex1 = -1 - 2i
complex1 + complex2 = 4 + 6i
complex1 does not equal complex2
```

Creating Namespaces

The last OOP topic we'll look at in this chapter is how to create your own namespaces. As you know, namespaces let you divide programs up to avoid clashes between identifiers (even if you don't declare your own namespace, your code is given its own default namespace, the global namespace). To create your own namespace, you use the `namespace` keyword:

```
namespace name[.name1] ...] {
  type-declarations
}
```

Here are the parts of this statement:

- *`name`*, *`name1`*—A namespace name can be any legal identifier, and it can include periods.
- *`type-declarations`*—Within a namespace, you can declare one or more of the following types: another namespace, a `class`, `interface`, `struct`, `enum`, or `delegate`.

You can create as many namespaces in a file as you want; just enclose the code for each namespace in the code block following the `namespace` keyword. For example, here's how we can put the `Messager` class into the namespace `Output`:

```
namespace Output
{
  class Messager
  {
    public string message = "Default message.";

    public void DisplayMessage()
    {
      System.Console.WriteLine(message);
    }
  }
}
```

NAMING NAMESPACES

Microsoft's suggestion is to name namespaces this way: `CompanyName.ProjectName.Component.SubComponent`.

To access `Messager` outside the `Output` namespace, you qualify its name as `Output.Messsager`, just as you can qualify `Console.Writeline` as `System.Console.WriteLine`. (Alternatively, you can use a `using Output` directive, just as you use a `using System` directive.) You can see this at work in ch03_15.cs, Listing 3.15.

LISTING 3.15 Namespace Example (ch03_15.cs)

```
class ch03_15
{
  public static void Main()
  {
    Output.Messager obj = new Output.Messager();
    obj.message = "Hello from C#.";
    obj.DisplayMessage();
  }
}

namespace Output
{
  class Messager
  {
    public string message = "Default message.";

    public void DisplayMessage()
    {
      System.Console.WriteLine(message);
    }
  }
}
```

Here's what you see when you run ch03_15.cs:

```
C:\>ch03_15
Hello from C#.
```

Namespaces can also extend over multiple files—more on how that works when we discuss assemblies. We took a look at overloading in this chapter—how about overriding? As OOP programmers know, overriding is one of the most important aspects of class inheritance, and it's covered in the next chapter.

In Brief

This chapter covered the essentials of C# OOP, starting with creating classes and objects. We saw that access modifiers let you restrict access to the members of classes and structs, encapsulating your data and methods as needed. Here's an overview of the topics we covered:

- We can add fields, methods, and properties to classes, and we can initialize fields with initializers. We can use the `this` keyword in methods, and create not only standard properties, but also read-only and static properties. And we can create constructors and copy constructors to initialize the data in objects.
- Static members are declared with the `static` keyword, and are associated with the class itself, not with objects created from that class. Static fields can store data for the class, and static methods and properties can be called without first instantiating an object.
- Method overloading is implemented by providing alternative definitions of the same method with different signatures. Operator overloading uses the `operator` keyword to let you specify which operator to overload.
- When we overloaded the == operator, C# will issue a warning unless we also override the `Object.Equals` and `Object.GetHashCode` methods.