Top of Form



**Chapter 1**

**Introduction**

Up to this point, I have tried to present C# to you as a procedure-oriented language. That is, the programs that you have seen so far have focused mainly on the tasks that are to be accomplished. I have done this mostly because this is a very natural way to think about programming. However, C# is an object-oriented language. That means that we are able to focus on objects and how they can be used to solve our problems.

For example, imagine if you were asked to write a program that did payroll for a company. In a procedure-oriented program, you would concern yourself with calculating the gross pay, taxes withheld, and the net pay. Each of these calculations would probably be accomplished by writing a method.

However, if you were thinking of this problem in an object-oriented sense, you would focus on the objects of this problem. You would concern yourself with things like employee objects, time/date objects, and paycheck objects. As you will see, each of these objects has a so-called "state of being," and each provides a way to change the current state.

All of this may seem a little foreign at this point, but do not be concerned. By the end of this lesson, it will all make much more sense, and you will be on your way to being an excellent object-oriented programmer. Since you currently do not know what a class is, we will start this lesson by learning a little bit about what classes are. We will then move on to see how to create a class and then how to use these classes to accomplish the tasks of our problem.  
  
  
  
  
**Chapter 2**

**Reminder:** Whenever you see a " **¬** " character at the end of a line of code, this denotes a break in that code-line due to our HTML page size limitations. The remaining section of that code-line can be found on the next line below. After pasting your code into your text-editing program, simply select and delete that character, plus the hidden-break following it, to realign the code-line properly.

**Understanding and Writing Classes**

C# has two different types of classes. You have a lot of experience with the first type of class. Every program you have written so far has been an application program, which means that you wrote a class containing the Main() method. Recall that the program execution begins in the Main() method. The second type of class is one that is used as a blueprint for an object. These classes do not contain a Main() method. But what is meant by the term *blueprint*?

If you look around you right now, you will see lots of different objects. Let's take a closer look at a very simple object, a tennis ball. All tennis balls look very similar. Each ball can be described by a set of characteristics. That is, every tennis ball has a certain size, a certain texture, and a certain color. Now when the balls come out of the can, all three of those balls should be identical. Why is that? The answer is because they came straight from the factory, and we assume that the factory produces identical balls. How did the factory do that? Well, they had a design or a blueprint for what they wanted the ball to look like. Using object-oriented terms, the blueprint for the tennis balls was the *class*. The factory used that class to create three different balls, or *objects*.

Using the tennis ball analogy some more, let's think about the ball's characteristics. Most tennis balls are yellow, but have you ever seen an orange tennis ball? How did the factory do that? Easy. One day the plant manager came in and did something in the factory to make it so that instead of producing a yellow color, the factory produced balls that were orange. Notice the balls that were produced were still tennis balls. That is, the blueprint was still the same; it was just that one of the characteristics was different. When we write our blueprints, or classes, we will use class variables to keep track of the characteristics of our objects. This means that when we create our objects, we will be able to change the characteristics of our object by changing the value of its variables.

In addition to characteristics, an object can have a set of behaviors. Again using the tennis ball, you can throw it, bounce it, or you might even change its color with the help of a little paint. We will model this in our C# objects by writing methods. These methods will be used to do things with or to the objects that we have created.

Many of you now may be scratching your heads wondering why we would want to model a tennis ball in C#. And to be honest, I'm not sure why you would. I just picked a tennis ball to model because it is a familiar object to which we can all relate. Let me give you another example, this time using C# and a class that you might see as being very useful, a time.

How many times have you heard someone ask, "What time is it?" Probably more than you can count. But let's think about time as an object. What characteristics do you need to keep track of in order to define what time it is? If you think a little bit, you will realize that you need to keep track of the hour, the minute and the second. With these three numbers, you can identify the exact time. So, if you were to write a C# class to define the time, you would create three int variables: one for the hour, one for the minute, and one for the second. Let's take a look at the code that we would write for such an example.

public class Time

{

private int hour;

private int minute;

private int second;

}

Notice how this class starts the same way as the classes that we wrote containing the Main() method. After this line of code, you will notice that the three variables are declared. The declaration statement here is the same as we saw before except that we are using the keyword private before the declaration. This keyword is what is called an *access modifier*. That means that it specifies which parts of code will have access to this variable to set or get its value. For this class, we will use two different access modifiers, public and private. The *public modifier* says that any line of code anywhere can access that variable. On the other hand, the *private modifier* says that only code inside the class will have access. I will explain later why I gave these variables the private modifier.

In addition to storing information about the time, you would need to provide a set of behaviors for your objects. For example, in your program, it would be necessary to be able to get and set the hour, minute, and second. It would also be nice to be able to print the current values of the time object to the screen. We will provide these behaviors by writing methods. The example below shows only the methods for getting and setting the hour. Other methods would be written in a similar fashion.

using System;

public class Time

{

private int hour;

private int minute;

private int second;

public void SetHour(int h)

{

if(h < 0 || h > 23)

{

Console.Out.Write("Invalid hour.");

Console.Out.WriteLine(" Setting hour ¬

to 0.");

hour = 0;

}

else

hour = h;

}

public int GetHour( )

{

return hour;

}

}

In this code, you will notice how these methods look very similar to the methods you have seen so far. There are a couple of differences, though. First, notice that both methods use a variable named *hour*. You may be remembering that I made a big deal about declaring your variables before using them in code. You may have even tried to do something like this earlier and the code did not compile. However, this code does compile. The reason is because we declared the hour variable to be an *instance variable*. Instance variables are declared inside the class and outside of any methods. By declaring them at this location, we are telling C# that all methods inside the class have direct access to them. Therefore, the hour variable has been declared, and C# knows that your code is talking about the instance variable named *hour*.

Before we get to the second difference in these methods, I want to take a quick look back at my discussion of an object. Recall that I said that objects will have a state of being, which is kept track of by using variables. In addition to that state of being, our objects will have a set of behaviors that provide us a way of a getting or changing the current state. You can see from the above example that one of our characteristics of the state of being of our time object will be the hour. The GetHour() and SetHour() methods provide a way to access or change this state.

The second difference in these methods is again the access modifier. Notice how, this time, I gave my methods the public modifier. In general, you will declare your instance variables with the private modifier and your methods with the public modifier. The reason this is done is because you do not want another class directly changing the value of your instance variables, because they might do something wrong. Instead, you give them access to your private variables by giving them some public methods that they can use. You can then write code inside this method to check to be sure the value is valid. Using our hour variable as an example, think about if another class had direct access to the hour variable. There would be nothing stopping that code from setting the hour to 50, for example. Since this is not a valid hour, we restrict access to the hour variable and force the code outside the class to use our public SetHour() method which does the error checking.

**Chapter 3**

**Instantiating Classes**

Now that you know how to write a class, you need to know how to use one. Recall that a class is just a blueprint for an object. In order to do anything useful, you need to create an instance of the class. Again think of our tennis ball example. It would be awfully hard to play a good game of tennis if you used the blueprint for a tennis ball. Creating an instance of a class is called *instantiating* the class, or sometimes *declaring an object*. Declaring objects is a two-step process.

The first step of declaring an object is to create a variable. Creating a variable that stores an object is just like declaring any other type of variable. You simply use the data type and the variable name except that this time, the data type is the name of a class. For example, if you wanted to declare a Time object named *myTime*, you would type:

Time myTime;

This tells the compiler that the myTime variable will store a reference to a Time object. It is important to know that we do not actually have an object yet. The way I usually explain this is by thinking about int variables. When we declare an int variable, we immediately get a storage location because the compiler knows exactly how much space is needed to hold an int. However, objects are complex things. An object can have many different variables and, therefore, can take up a lot of memory. The computer may need to go very far into memory to get a block of continuous memory big enough to store our object. Therefore, our object variables do not actually store the values of the object. Instead, they store a reference to the spot in memory where the object will be stored.

So, how do we actually get an object? Well, that is step two of the process. For step two, we need to use the *new operator*. For example, to get a Time object and store its reference in the myTime variable:

myTime = new Time();

Of course, both of these steps can be done at the same time by combining statements:

Time myTime = new Time();

Looking at the text following the new keyword, you will see something that looks like a method call because it has parenthesis after it. But this word is the name of the class; how can that be? The answer is that this is a *method call*. It is a call to a special method called a *constructor*. We will learn about constructors later in this lesson.

Let's look at a set of code that will define a class and then create an object. Note that in order to save space here, I am not going to write a complete class. This code would be typed into a file named CreateTime.cs.

using System;

public class CreateTime

{

public static void Main()

{

Time myTime = new Time();

Time yourTime = new Time();

myTime.SetHour(5);

Console.Out.WriteLine("The hour is: " + ¬

myTime.GetHour());

Console.Out.WriteLine("The hour is: " + ¬

yourTime.GetHour());

}

}

public class Time

{

private int hour;

private int minute;

private int second;

public void SetHour(int h)

{

if(h < 0 || h > 23)

{

Console.Out.Write("Invalid hour.");

Console.Out.WriteLine(" Setting hour ¬

to 0.");

hour = 0;

}

else

hour = h;

}

public int GetHour()

{

return hour;

}

}

Type this code in and compile it. You will see that the compiler gives you a warning saying that the fields 'timeminute' and 'timesecond' are never used, which is correct. I added these variables so that you could use them when you work your assignment at the end of this lesson. For now, since this is only a warning by the compiler, it is okay to ignore it and run the program.

In this example, there are only five lines of code inside the Main method. The first declares the object variable and then instantiates the class. The second line of code declares and instantiates a second Time object. The third line is used to set the value of the hour variable in the myTime object. This is done by using the SetHour() method.

Notice how this method is called. First, I used the name of my object variable, then a dot (.), and finally the name of the method. This syntax is very important to using our objects. The dot operator tells the compiler that this method is associated with this object. This is how the computer knows which object's variables to change. Look again at the code and notice that I actually created two objects. That means that out in memory there are two hour variables, two minute variables, and two second variables. If we tried to call the SetHour() method without giving it an object, how would the computer know which hour variable to set to 5? It wouldn't be clear to the computer; therefore, we use the object name and the dot operator to let the computer know which hour variable to change.

The code in Main finishes up by displaying a message and the value of the hour variable in the myTime object to the screen, which is 5. The last line displays a message and the value of the hour variable in the yourTime object to the screen, which is 0. Once again, realize that we called the SetHour() method on only the myTime object to set the hour. However, this was never done for the yourTime object; therefore, the default value, 0, was displayed.

Now you should have an idea of how to write your class definitions and how to create and use objects. I want to turn your attention now to organization and file management. Although it is acceptable and sometimes very convenient to write two or more classes inside a single file, it is generally not a good practice. One of the reasons we are writing classes to define objects is so that our code can be reused. Also, since the Time class has nothing to do with the CreateTime class, there is really no reason to have them in the same file. For these reasons, we will break the previous file into two different files, one for each class.

First, we will work with the Time class. We can simply cut out the code for the Time class and put it in its own file. In addition to the original code, we will put one line at the top to create a namespace. You may recall from earlier in the course that a *namespace* is a way of organizing related classes in C#. The new Time.cs file becomes:

namespace TimeNamespace

{

using System;

public class Time

{

private int hour;

private int minute;

private int second;

public void SetHour(int h)

{

if(h < 0 || h > 23)

{

Console.Out.Write("Invalid hour.");

Console.Out.WriteLine(" Setting hour ¬

to 0.");

hour = 0;

}

else

hour = h;

}

public int GetHour()

{

return hour;

}

}

}

For the class with Main, we will give it the name *UsingTime*. Don't forget to save it as a *.cs* file. Since this file no longer contains the definition of Time, we will need to tell the compiler where it is located. This is done by typing a *using statement* at the top of the code. Our new file becomes:

using System;

using TimeNamespace;

public class UsingTime

{

public static void Main()

{

Time myTime = new Time();

Time yourTime = new Time();

myTime.SetHour(5);

Console.Out.WriteLine("The hour is: " + ¬

myTime.GetHour());

Console.Out.WriteLine("The hour is: " + ¬

yourTime.GetHour());

}

}

Now you have two new classes, and you will need to compile them. Unfortunately, by breaking the two classes apart, we have made compiling a little more difficult. If you attempt to compile the Time class as your did before with csc Time.cs, it will not compile. The reason for this is because the Time class does not have a Main method. Instead, you must compile this class as a *netmodule*. This is done by typing:

csc /t:module Time.cs

This command will create for you a file named *Time.netmodule*. Now you are ready to compile the UsingTime.cs file. Since this file depends on the new Time.netmodule file, we must explicitly tell the compiler this fact. Therefore, we will type:

csc UsingTime.cs /addmodule:Time.netmodule

|  |
| --- |
| Pen and notepad**Note**  It is important to place a blank space between the 'csc UsingTime.cs' command and the '/addmodule':Time.netmodule' in the command above. If you miss this, the compiler will give you an error. |

This creates our executable file called *UsingTime.exe*, and we are ready to run the program like normal with the command *UsingTime*.

**Chapter 4**

**Additional Class Features**

To this point, if you create an object, the only way to give it initial values is to instantiate your class and call on the setter methods for each variable. For our Time class, this means three method calls. However, for more sophisticated classes, this could involve many more method calls. It seems like there should be an easier way. And of course, there is an easier way. Earlier, I mentioned that the word following the new operator was a call to the constructor method. This method is a special method that looks very similar to other methods, but it has a few differences.

* The name of the constructor. The name of a constructor must be the same as the name of the class.
* How the constructor is called. A constructor cannot actually be called. Instead this method is run when the object is created.
* The purpose of a constructor. The constructor's purpose is to build the object or prepare the object to be used. This usually means giving values to the instance variables.
* The return type of a constructor. Since the constructor's job is to construct the object, there is never anything returned from a constructor. Also, since nothing is ever returned, we do not put a return type in the constructor's method header, not even void.

Using these ideas, we could write a constructor for our Time class as:

public Time(int h, int m, int s)

{

if(h < 0 || h > 23)

hour = 0;

else

hour = h;

if (m < 0 || m > 59)

minute = 0;

else

minute = m;

if (s < 0 || s > 59)

second = 0;

else

second = s;

}

Now, if we wanted to create an object called *myTime* that held the time 8:00:34, in Main() we could type:

Time myTime = new Time(8, 0, 34);

This saves us the trouble of calling on methods to set the hour, minute and second because this constructor will set these values for us. But how did we get away without a constructor before? Remember that we did not write a constructor in the previous chapter, yet the program compiled and ran. The answer is that since we did not write a constructor, C# provided one for us. The constructor that was provided for us is called a *default constructor*. A default constructor takes no parameters. In addition, the default constructor provided for us does not do anything.

Let's say you decide to go ahead and add the constructor listed above to your Time class. You will find that this code will compile just fine. However, if you try to compile the UsingTime file from the previous chapter, it will not work. Actually, the compiler will give you an error message saying that there is no constructor that takes zero arguments. That is, there is no default constructor. This may seem a little odd, because it was there before we wrote our own constructor. That is the exact reason it is no longer there. I always think of it this way: C# figures that if you do not provide a constructor, you must not know how to. Therefore, it writes a simple one for you. However, if you do provide a constructor, it figures that you know how and therefore does not provide anything for you.

The moral of the story is this: If you provide a single constructor for your class, you should also provide a default constructor to give your variables default values. For example, in our Time class, if our user does not specify the time, don't you think it might be a good idea to give the object the current time? This can be done with the following constructor:

public Time()

{

DateTime current = DateTime.Now;

hour = current.Hour;

minute = current.Minute;

second = current.Second;

}

Try adding this constructor in addition to the constructor shown earlier. Compile and run your program and see what is printed out. The program should display the first hour is 8 and the second hour is the current hour of the day.

At this point, it is important to bring up the idea of overloading methods again. Recall that an overloaded method is two or more methods with the same name but different numbers or types of arguments. As you have seen, constructors can be overloaded. In fact, constructors are often overloaded.

I also want to use the code in the previous constructor to talk again about classes. Notice the first line inside the constructor:

DateTime current = DateTime.Now;

This line may look a little confusing to you, so I want to explain it. First, you can see that this line is creating a DateTime object named *current*. The DateTime class is contained in the System namespace. Therefore, we are creating our DateTime object; however, here we are not using the new operator to create the object. Instead, we are assigning a DateTime object that already exists.

Let's break down the right-hand side of the equation. The text shows DateTime.Now. We know that DateTime is a class, so what is *Now*? Since it does not have parentheses after it, it cannot be a method. Therefore, it must be a variable. And since we are using this variable outside of the class definition, it must be a public variable. But you should remember that I said that in order to access a variable, you need to use the object name first. Here we have a class name. The reason for this is because *Now* is a different kind of variable. It is what is called a *static variable*. In addition to creating instance variables that belong to an instance of the class, or an object, you can also create static variables that belong to the entire class. These variables are declared just like instance variables, except that they use the keyword *static* before the declaration.

While we are on the subject of static variables, did you know that you can have static methods? Think about our Main method in the UsingTime class. In its header, you see the keyword *static*. That means that there is only one Main method for the UsingTime class. It also means that the Main method can be called using only the class name and that no object needs to be declared of type UsingTime. This is very important because the operating system is calling Main when your program is run. Since the operating system has no way of creating a variable, Main must be static. This course will not discuss static variables anymore other than to say that they do exist in some classes and I want you to know how to use them.

Getting back to our example, since *Now* is a static variable, we can use the class name. Interestingly enough, *Now* is a DateTime object. You could translate this line of code into English as: Create for me a DateTime object named *current* and store the DateTime object Now there. Now, let's explore the remaining lines of code in the constructor because it will help to reinforce the ideas of classes. The next line of code is:

hour = current.Hour;

This line is using the current object variable and calling on Hour. Again, since it has an object name in front of it and does not have parentheses after it, *Hour* must be an instance variable. In addition, since we are using this variable outside of the class, it must be a public variable.

As you can see, working with classes is a little more complex than the code we have been working with so far. However, if you can get the different concepts organized in your mind, your programs will actually become easier to write. The reason is because many common classes have already been written for you; you just need to link your programs to these classes and know how to use the code.

Getting back to the DateTime, this class provides for us a way to keep track of a time and a date. That means that the Time class that we are currently writing is essentially already written for us. Well, that is OK, because I think that the Time class is doing a good job of providing guidance for us on how to write and use a class.

I want to finish this lesson by taking a step back and talking about why we are writing these classes anyway. The main reasons are for organization and abstraction. By writing classes, we can now organize all of the code that keeps track of the time into one object. Our user does not need to keep track of three whole numbers that represent the time. Instead, they can just create one Time object that keeps track of those variables. In addition, you can probably see that it would be useful to have methods to compare two Time objects or print the value of the Time object in different formats, say in regular format or in military format. In addition, you might see that it would be useful to compare two Time objects or even get the difference between two Time objects. You can provide all of these behaviors, as well as many others, by writing public methods for your user to use.

Writing classes also provides us with *abstraction*, or detail hiding. Look at the DateTime class. Do you know how it got the current time? Do you care? All you need to know is how to get a Now object and how to access the Hour, Minute, and Second members. The same would be true for someone using your Time class. In general, you would compile your Time class and provide the machine language file to someone else. That way, they can use your code, but they cannot see how you actually implemented the methods.

**Chapter 5**

**Summary**

This chapter has given you a new way to look at programming. You can now change the way that you code by focusing on objects. Each object will contain a set of characteristics and behaviors. The characteristics are stored using instance variables, and the behaviors are provided with methods. The beauty of this is that now you can organize your code better while abstracting the details of that code away.

In the next lesson, you will learn how to use files in your programs. To this point, all of the input has come from the keyboard and all of the output has gone to the screen. This has been OK so far, because your programs have not dealt with much data. However, many programs that you write will need to work with a lot of data. You will not want to spend time entering all of this data into your programs over and over as input. Also, you will want to store all of the output that comes from your program. Files will provide a way to do this for us. The next lesson will also explain to you what an exception is and how you can keep these things from crashing your programs.  
  
  
  
**Supplementary Material**

|  |
| --- |
| [DateTime Members](http://msdn.microsoft.com/en-us/library/system.datetime_members(VS.85).aspx)  http://msdn.microsoft.com/en-us/library/system.datetime\_members(VS.85).aspx |
| This page is from the MSDN site. It lists all of the member functions for the DateTime class. The left panel contains links to DateTime member variables also. |
| [Introduction to Classes](http://www.csharp-station.com/Tutorials/Lesson07.aspx)  http://www.csharp-station.com/Tutorials/Lesson07.aspx |
| This page describes object-oriented programming in a different fashion. |

**FAQs**   
  
**Q:** When I try to compile my Time class, I keep getting the following error:  
  
error CS5001: Program 'c:\Lesson 10\Time.exe' does not have an entry point defined'  
  
What is wrong?  
  
**A:** Time is a class that does not contain Main. It is meant to be used to create objects. Therefore when you compile it, you must specify it to be a netmodule. You should be compiling with the following statement:  
  
csc /t:module Time.cs  
  
  
**Q:** When I try to compile my UsingTime, I keep getting the following error:  
  
UsingTime.cs(2, 7): error CS0246: The type or namespace name 'TimeNamespace' could not be found (are you missing a using directive or an assembly reference?)  
  
I have the correct 'using TimeNamespace;' statement at the top. What is wrong?  
  
**A:** How did you compile the UsingTime.cs file? Remember, in addition to putting your 'using' statement in the code, you also need to compile your program and add in the Time.netmodule file. You should be compiling with the following statement:  
  
csc UsingTime.cs /addmodule:Time.netmodule

**Assignment**   
  
  
Finish the Time class that was started in this lesson. Do this by completing the following methods:

* **SetMinute(int m)**
* **SetSecond(int s)**
* **GetMinute()**
* **GetSecond()**
* **AddSecond()** - This method should add 1 to the second variable stored in the current object. If the second becomes 60, be sure to set the second equal to 0 and increment the minute by one.
* **AddMinute()** - This method should add 1 to the minute variable stored in the current object. If the minute becomes 60, be sure to set the minute equal to 0 and increment the hour variable by one.
* **AddHour()** - This method should add 1 to the hour variable stored in the current object. If the hour becomes 24, be sure to set the hour equal to 0.
* **DisplayCivilian()** - This method should display the time stored in the object as civilian time.
* **DisplayMilitary()** - This method should display the time stored in the object as military time.
* **Equals(Time t)** - This method should compare the time stored in the current object to the time stored in the Time object *t* that is passed into the method.

[Click here for solution: **Time.zip**](https://api.ed2go.com/CourseBuilder/2.0/images/resources/prod/cpb-0/Time.zip)

Now change the code in the UsingTime class to test each of these methods.

[Click here for solution: **UsingTime.zip**](https://api.ed2go.com/CourseBuilder/2.0/images/resources/prod/cpb-0/UsingTime.zip)

|  |
| --- |
| Copyright © 1997 - 2012 Education To Go. All rights reserved. The material on this site cannot be reproduced or redistributed unless you have obtained prior written permission from Education To Go. Education To Go and ed2go are registered trademarks of Education To Go, a part of Cengage Learning. |

Bottom of Form