Top of Form



**Introduction to C# Programming: Lesson 5**

**Chapter 1**

**Introduction**

Grouping sections of code into value-returning methods enables you as the programmer to better organize your code and allows for more code reuse. These are two things that will make your programming experience much better. But what do you do if you have a section of code that you want to make into a method, but there is no result from the code? For example, imagine that you need to write code that will print out three lines that will be used as a heading for your output. You decide that you are going to put into practice your new skills at writing methods. But there is nothing to be returned from this code! You will need to put in what is called a void method.

This lesson will show you how to write and use a void method. In addition, you will learn how to pass the address of a variable to a method. This may not seem like an important thing now, but you will soon see how necessary it really is. Finally, this lesson will finish up with a discussion of method overloading, or two or more methods with the same name.  
  
  
  
  
**Chapter 2**

**Writing a Void Method**

As shown earlier, there are times when you want to split off a group of code into a method, but there is no value to be returned. At first, this may not seem like a very big problem, but remember how a method is written. Let's start by looking at a method that computes the volume of a cube.

public static double CalcVolumeCube(double¬

length1, double length2, double length3)

{

double volume;

volume = length1 \* length2 \* length3;

return volume;

}

This simple method takes in three double values, one for each side of a cube. The method then calculates the volume by multiplying the three values together. Finally, the method returns the calculated value to the statement that called the method.

Let's now relate this to a method that will not return any value. If there is no value to be returned, then what should go in the method header for the return type? You might guess that the programmer should just leave this area blank. After all, if there is no value to be returned, then there is no return type, and therefore, no word is necessary in the method header. While this does seem to make sense, it will not work. C# expects to see some word there specifying a return type. To let the compiler know that your method is not a value-returning method, you will use the keyword *void* in the method header.

For example, look at the following method header for a method that returns an integer:

public static int MyMethod()

By contrast, this method is a void method:

public static void MyMethod2()

Notice how the two method headers are identical, except for the return type. You can just think of the word *void* as a return type for *nothing*.

A second difference between value-returning and non-value-returning methods is the return statement. Again, you may guess that since there is nothing returned, there is no need for the return statement. This is correct. C# allows the programmer to leave out the return statement in a void method. However, C# also allows you to use the return statement by itself. That is, you can type return; in a void method. Either is perfectly acceptable, and sometimes you will prefer one over the other. To help show you the difference, below is a complete void method that prints out a copyright notice:

public static void PrintCopyright(string company)

{

Console.Out.WriteLine("This material is ¬

property of " + company);

}

For another example of a void method, you may think about a Main method. Has it dawned on you yet that you have already seen, written, and used a void method? Main is the most famous and most important void method of all, since it is used in every C# program.

The last thing that you will need to know about a void method is how to call it. Again, let's look back to how we called on a value-returning method. The C# statement just used the name of the method and listed any arguments inside the parentheses. Recall that with value-returning methods, we needed to do something with the value that was returned. There were three choices: We could print the result, store the result, or use the result in an equation. Since void methods do not return values, there is no need to do anything with a result. Therefore, when you call on a void method, you can simply type the name of the method and list any arguments inside the parentheses.

For review, the following is an example of a call to a value-returning method:

result = CalcResult(5, 10);

Notice how the value that is returned is stored in a variable named *result*. In contrast, the following is a call to a void method:

PrintSummary(result);

Again, you can see that just the name of the method is used, and since there is no value returned, there is nothing to be stored.

To better help you to put all of the concepts of void methods together, the following is a complete program that uses both a value-returning and a void method:

using System;

public class ManyMethods

{

public static void Main()

{

// Get two values

int value1 = GetNumber();

int value2 = GetNumber();

// Call on Division

Division(value1, value2);

}

public static int GetNumber()

{

string numString;

int number;

Console.Out.Write("Enter a whole number: ");

numString = Console.ReadLine();

number = Convert.ToInt32(numString);

return number;

}

public static void Division(int numerator,

int denominator)

{

int quotient = numerator / denominator;

int remainder = numerator % denominator;

Console.Out.WriteLine(numerator + ¬

" divided by " + denominator + " equals");

Console.Out.WriteLine(quotient + " with ¬

a remainder of " + remainder);

return;

}

}

This simple program makes use of integer division and the % operator to help the user to compute the division of two numbers. But what the user does not see is that there are three methods: Main, GetNumber, and Division. Two of these methods are void methods and one is value returning. Notice that the code in GetNumber is used twice. Since all values must be read in as strings and then converted into their appropriate type, this is great code to put inside its own method. Now, any time you want to read in an integer from the user, you just need to call on the method. It will do the prompting, reading, and converting.

After using the GetNumber method twice, the program then calls on the Division method by passing it the two numbers that were just read. This method then calculates the quotient and remainder from dividing the two numbers. These values are displayed on the screen. With nothing left to do, this method uses the return; statement to exit the method. Again, this statement is not required, but you may do this in your programs.

Notice also the differences between calling the value-returning GetNumber method and the void Division method. As I said earlier, since there is no result from a void method, there is nothing to print, store, or use in an equation.

Now that you have seen how to use the two kinds of methods, I want to show you something that is important to remember. The compiler will allow you to call on a value-returning method without printing, storing, or using the result in an equation. For example, change the Main method from the ManyMethods class above to:

public static void Main()

{

int value1 = 10;

int value2 = 3;

GetNumber();

GetNumber();

Division(value1, value2);

}

Compile and run the program. You will see that the results are always for the numbers 10 and 3. This may be confusing for your user, but you should understand why this is happening. The GetNumber method is called two different times. Each time the user is asked for a number, a number is read in, and then that number is returned. But what happened to it? Since the calling statement did not store, print, or use the result in an equation, the value is simply returned and then disappears. This is something that I find confuses beginning programmers. Remember that just because your program compiles, it does not mean that it is correct.  
  
  
  
  
**Chapter 3**

**Passing by Value and by Reference**

At this point, you have seen value-returning methods, void methods, methods that take parameters, and methods that do not take parameters. You might think that there is nothing else to learn about methods. But there is: passing variables by value and passing variables by reference.

To show you why this is important, I want to start with an example. Consider the following program:

using System;

public class ManyMethods

{

public static void Main()

{

int number = 1;

ChangeNumber(number);

Console.Out.WriteLine("The number in ¬

Main is " + number);

}

public static void ChangeNumber(int value)

{

value = value + 1;

Console.Out.WriteLine("The new value ¬

inside ChangeNumber is " + value);

}

}

Notice that the value of the variable number passed from Main is passed to the ChangeNumber method. That means that the number *1* is assigned to the local variable value. Since value is a local variable to the ChangeNumber method, anything that happens to value affects value and not number. This is very useful, because most of the time, you do not want the values of your variables to be changed by another method. It is often said that your variable was passed *by value*. That means that only the value of the variable was passed from the calling statement to the method. You could also say that a copy of the value is created, one with the same value but a different location in memory. By default, all variables in C# are passed by value.

But what if you do want the method to be able to change the value of your variable? In that case, you would need to pass the variable *by reference*. When you pass a variable by reference, you are actually passing the memory address for that variable. In this case, the program has two references to the same memory location. That means that any changes made using one reference are felt by the other reference as well. If the above program had passed the variable number by reference, then both output statements would have shown the value *2*.

There are two different keywords associated with passing by reference in C#, ref and out. This keyword is placed in the method header just before the declaration of the variable in the parameter list. Let's take a look at the following method header:

public static void AnotherMethod(int val1,   
ref int val2)

In this method, two different integer parameters are passed. The first parameter, val1, will receive an integer by value. That is, a copy of the value is made and stored in the memory location for val1. On the other hand, the second parameter, val2, is passed by reference. That means that val2 will hold the memory location of another integer variable. Therefore, any changes to val2 that are made in method AnotherMethod will be felt outside the method.

To change our method from before that adds one to the number that is passed in, all we have to do is change the method header to read:

public static void ChangeNumber(ref int value)

In addition to the method header, we also have to change the method call to read:

ChangeNumber(ref number);

That's it! This is a very simple change, but it is one that has serious consequences. Students often ask me, "How am I supposed to know when to pass by value and when to pass by reference?" The rule of thumb that I usually give them is this: Never pass by reference unless there is an important reason to do so.

One time that you would want to pass by reference is when you need to get two or more values back from a method. I didn't mention it earlier, but did you wonder if this was possible? Well, it is when you pass by reference. One common example used to demonstrate this point is a method that is used to swap the values of two variables:

public static void Swap(ref int one, ref int two)

{

int temp = one;

one = two;

two = temp;

}

Since these variables receive references to values, the effects of this code are felt outside the method call. In essence, two values were returned. I always urge my students to make methods that receive variables by reference void methods. It is not required by C#, but it is something that I find helps people remember that passing by reference is a dangerous thing to do.

You may recall that I mentioned that there are two different keywords for passing variables by reference. You have seen some examples of ref, but you can also use out. The only difference between the two keywords is that ref variables must come to the method with an initial value. Take the following code as an example:

using System;

public class ManyMethods

{

public static void Main()

{

int number;

ChangeNumber(ref number);

Console.Out.WriteLine("The number in Main ¬

is " + number);

}

public static void ChangeNumber(ref int value)

{

value = 10;

Console.Out.WriteLine("The new value inside ¬

ChangeNumber is " + value);

}

}

If you attempt to compile this program, you will get an error. The error will read something like, *use of unassigned local variable*. If you wanted to use this code above, you would need to either give the number variable an initial value or change the keyword ref to out.

One final point that I would like to make about passing by reference is the use of literal values. Suppose you had a method header like the following:

public static void RefMethod(ref int one)

You cannot call this method using the following statement:

RefMethod(5);

Although 5 is an integer value, you cannot pass it by reference. That is because you cannot change the value of 5; it will always be 5. Therefore, when you pass something by reference, the calling statement must use a variable and not a literal value.

**Chapter 4**

**Overloaded Methods**

At this point, we have learned that there are many ways to write a method. Your method can be value-returning, void, pass parameters by value, or pass them by reference. With all of these different options, you may think that you have everything covered. And for the most part, you are correct. However, I do have another scenario to present to you. In the previous section, I presented a method to you whose goal was to swap the values of two integers. This was a void method that took references to two different integer variables. Recall that the method header for this method was:

public static void Swap(ref int one,   
ref int two)

This method is very useful, but what happens if you try to pass it two double variables? The answer, as you might suspect, is that the code will not compile. Remember that the compiler requires that the variable types in the arguments of the method call match the parameters in the method header.

You might consider casting your double variables into int variables before passing them to the method. But after thinking about this, you will quickly realize that doing so will cause you to lose important data during the conversion.

The only choice left would be to write a new method. This method header might look like this:

public static void SwapDouble( ref double one,   
ref double two)

This would be perfectly legal C# code and would definitely make sense. But wouldn't it be nice to name this method Swap? There is nothing special about an int so that it should be called Swap and not SwapInt, right? Well, C# does allow for you to name them both Swap. This may seem like a contradiction to what was said earlier in the course. Remember when we first started talking about method names, I said that method names must be unique. Well, that is sort of true. You can have more than one method with the same name as long as the signature of the method is different. The method *signature* is defined as the name of the method and its parameter list.

By specifying a different signature, or a different number or type of parameter in the parameter list, you can create more than one method with the same name in the same class. This is called *method overloading*.

Using method overloading, we can now have two methods in the same class with the following headers. This is exactly what we need for our little problem with the Swap method. Note the method bodies have been left out for this example:

public static void Swap( ref int one,   
ref int two)

and

public static void Swap( ref double one,   
ref double two)

Suppose you place a statement in your program that says:

Swap(ref first, ref second);

The compiler determines which method is called based on the data type of the variables first and second. Often, when I think of the compiler looking for an overloaded method, I think about a child's toy that has an opening for a triangle, a square, and a circle. The goal is for the child to put the circular block through the circle, the triangle through the triangle, and the square through the square. Likewise, the compiler has the same job. It will take note of the number, type, and order of the arguments in the method call and then go searching through the class for a method signature that exactly matches the method call. Again, this makes programming more convenient for the programmer.

The final point I want to make about method overloading is about the body of the overloaded methods. Although the body of your overloaded methods will often be the same, they are not required to be so. The following code is an example of overloaded methods. Although this is not a particularly useful program as far as what it does, it demonstrates the ability to overload methods in C#.

using System;

public class Overloading

{

public static void Main()

{

MyMethod(1, 2);

MyMethod(1.0, 2.0);

}

public static void MyMethod(int a, int b)

{

Console.Out.WriteLine("This method takes ¬

two integers");

}

public static void MyMethod(double a, double b)

{

Console.Out.WriteLine("This method takes ¬

two doubles");

}

}

This Main method in this program has only two statements, and both are a call to the MyMethod method. The interesting part of this program is that the first method call passes two integers. When the compiler gets to this line of code, it searches the class for a method named *MyMethod* that takes two integers as parameters. When it finds this method, the code inside is run and a message is displayed telling the user that the method with two integers was called.

The second call to MyMethod passes two doubles. Just as it did before, the compiler searches the class for a method named *MyMethod* that takes two doubles. When it finds this method, the code is run and a message is displayed telling the user that the method with two doubles was called.  
  
  
  
  
**Chapter 5**

**Summary**

In this lesson, you took a very in-depth look at methods. You built on your knowledge of value-returning methods in Lesson 4 by exploring void methods. These methods were a group of code that performed some task, but they did not return a value. You also learned that by default all variables in C# are passed by value. However, C# also allows for the programmer to specify that variables are to be passed by reference. This gives the method access to the memory location of the variable. Therefore, when changes are made inside the method, the effects are felt outside.

Finally, this lesson ended with a discussion of method overloading. You can overload a method by giving the same name to two or more methods. The only difference between the methods is their parameter list.

With a better grasp of how to write methods, we will now turn our attention to the second of our three fundamental programming structures. Recall that so far all of our programs have used the sequence structure. That is, the program is run step by step in order. In the next two lessons, you will learn about the selection structure. Programs that determine the shortest route between two cities on a map or which items contain the keywords you are searching for at an auction site use the selection structure. The program's ability to make decisions is a powerful tool for you to learn and master.  
  
  
  
**Supplementary Material**

|  |
| --- |
| [Passing Variables](http://www.yoda.arachsys.com/csharp/parameters.html)  http://www.yoda.arachsys.com/csharp/parameters.html |
| This page discusses passing variables in C#. |
| [Method Overloading](http://www.csharpfriends.com/Articles/getArticle.aspx?articleID=105)  http://www.csharpfriends.com/Articles/getArticle.aspx?articleID=105 |
| More examples of method overloading. |

**FAQs**   
  
**Q:** Why do you tell us that it is so bad to pass our variables by reference?  
  
**A:** An analogy that I have heard about passing by value versus passing by reference regards a checking account. If your phone rang and your best friend asked you for the balance in your checking account, you might be a little concerned about giving this out. But in reality, there isn't much your friend can do with the balance.   
  
By contrast, if your friend called and asked for your checking account number, you should think hard before giving that out. That is because with your account number, it would be easier for your friend to gain full access over your account, possibly even withdrawal the funds there.  
  
The same idea works with passing data to methods. If you give the method the value, it really doesn't matter what they do to it, you still have the original. But if you pass the address, then that method can do whatever it wants to the value.

**Assignment**   
  
  
Write a program that will help to balance your checkbook. To help practice the ability to pass variables by reference, you should write the following methods:

* A method that reads in a double from the keyboard. Remember, all input is read in as a string, and then it must be converted to the appropriate data type. This method should be a void method that takes a reference to a double variable as a parameter.
* A void method that takes the beginning balance, the total deposits, the total withdrawals, and a reference to the variable containing the ending balance.

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