Top of Form



**Introduction to C# Programming: Lesson 8**

**Chapter 1**

**Introduction**

Early in the course, you learned that there are three basic control structures used in computer programming: sequence, selection, and repetition. You have learned that every program written uses the sequence structure, which is the execution of instructions, one by one, in the order that they are written. In the past two lessons, you have learned that many programs also contain the selection structure. The selection structure is used when you want the computer to make a decision and then execute a certain set of instructions based on the result of the decision. This leaves us with the final structure, repetition.

One of the biggest reasons that computers are so useful in our world today is because computers are very accurate. For example, a computer can add the numbers between one and one million and never make a mistake. You can have the computer add the next million numbers and it still won't make a mistake, plus it will not complain about having to work so hard or so long to solve such a problem.

Now we want to take this ability to do very routine, redundant, often boring tasks with such great accuracy and put it to work in our computer programs. This is done by using the repetition structure. The repetition structure is used to make the computer perform one or more operations until some condition is met.

This lesson will show you how to implement the repetition structure in three different ways: the while loop, the for loop and the do loop. Along the way, you will learn about counter and accumulator variables and how they are used with loops.  
  
  
  
  
**Chapter 2**

**while Loops**

Whenever I get to the loops section of the course, I am always tempted to give my class a pop quiz that would look something like this: Write a program that prints the first five numbers on the screen. Of course, at this point in the course, this should be a fairly simple task to accomplish. You would just need five output statements that look similar to:

Console.Out.WriteLine("x");

Remember, here the letter *x* would be replaced by the number to be printed. This is easy, right? But after a minute or so, I would stop the class and say, "Oh, did I say the first five? I meant to say the first five hundred." At this point, I would expect to hear hems and haws and see dirty looks. Now instead of writing five lines of code, you would need to write five hundred lines of code. Suppose I then said, "No, not the first five hundred, the first five thousand."

Certainly at this point in the course, you could write a program that would perform such a task. You might even decide that you could cut and paste and get a bunch of the numbers printed a little bit easier. However, this would still be quite a task. This program is much easier to write using a repetition structure. Let's look at the repetition structure some and then come back to this example.

Every repetition structure, or loop, consists of a loop body and loop condition. The statements that make up the inside of the loop are known as the *loop body*. The *loop condition* is a Boolean expression that is checked to determine the number of times the loop body is executed. This condition is similar to the condition used in the if selection structure. That is, this condition must evaluate to either true or false.

The syntax for the *while repetition structure* looks like the following:

*while (<condition>)  
  
statement;*

There are a couple things that I want to point out here. First, I want you to see that the loop condition is located in the while part of the structure. Notice how this statement comes before the loop body. This means that the condition is tested before the body is executed. For this reason, programmers call the while structure a *pretest loop*.

Second, notice that the body of the while loop consists of just one statement. However, just like we saw with the if statement before, we can use braces to make a block statement if we need to have more than one statement in our loop body. We will almost always need to use a block statement when writing a while loop. Let me show you why this is so with an example:

int a = 1;

while (a <= 2)

Console.Out.WriteLine("Hello");

This program segment will begin by creating an int variable called *a* and assigning it the value *1*. The next statement tests to see if the value 1 is less than or equal to 2. Since this is true, the program executes the code in the body of the loop and the word *Hello* is printed on the screen. Since this is the last line of the loop body, the computer goes back and tests the condition again. Again the computer determines that the value 1 is less than or equal to 2, and the loop body is executed again. At this point, you can see why we need to use a statement block for the loop body. We need something to change the value of the variable *a*.

The most important thing to remember when writing loops is to make sure that the loop will end. If you had typed the above code into a program, it would compile and run just fine. However, once the program execution entered the loop, it would never leave. That means that the word *Hello* would be printed to the screen over and over, and the program would never end. This is called an *infinite loop*. When you have this happen to you, notice I say "when" and not "if," you can stop the program execution by holding down the CONTROL key and pressing the letter C.

Now you see that in order to make the loop stop, you will need to update the variable *a* inside the body of the loop. However, if your loop body contained an update as its only statement, it wouldn't be a very useful loop. Therefore, we will make the loop body a block statement and place both the output statement and the update statement inside the body:

int a = 1;

while (a <= 2)

{

Console.Out.WriteLine("Hello");

a = a + 1;

}

Now the program will create the a variable and place the number 1 in memory. The condition is evaluated, and since 1 is less than or equal to 2, the loop body is entered. The first statement displays the word *Hello* on the screen, and the second statement adds one to the a variable. Since the end of the loop body has been reached, program execution continues with another evaluation of the loop condition. This time, the value 2 is compared to 2, and the condition evaluates to true. Again the loop body executes, the word is displayed on the screen, and 1 is added to the a variable. Since it is the end of the loop body, the condition is evaluated a third time. This time 3 is compared to 2, and the condition evaluates to false. When the loop condition evaluates to false, the loop body is skipped over and program execution continues with the next statement after the body.

Notice how the variable *a* was used in this example. It was used to count the number of times the loop was executed. For this reason, the variable *a* is referred to as a *counter variable*. When I write programs, I typically call my counter variables *count* or *counter*. This helps to make my code easier to read and understand.

A second point that I would like to make is about the counter variable's starting and ending values. This program displayed the word on the screen two times. To do this, I started my counter at 1, and condition was (a <=2). What would have happened if I had started my counter at 0 and the condition was (a <2)? The answer is that the word would have been displayed two times. The point is that you can start your counter at any value you want. Oftentimes you will want your loop to execute a specific number of times. For this example, we want the loop to execute two times. I usually tell my students to use one of the above examples and always use that same format. That is, always start your counter at 1 and have your condition be less than or equal to the number of times you want your loop run, which is also called a *loop iteration*. The other choice is to start your counter at 0 and make your condition be strictly less than the number of times you want your loop executed. If you jumble the two, you will be off by one iteration, and that is not a good thing.

To show you what I mean, type in the following complete program:

using System;

public class Hello

{

public static void Main()

{

String strNumber;

int number, count;

Console.Out.WriteLine("How many times do ¬

you want to say hello?");

strNumber = Console.ReadLine();

number = Convert.ToInt32(strNumber);

count = 1;

while (count <= number)

{

Console.Out.WriteLine("Hello");

count = count + 1;

}

}

}

Run this program. It should print out the message the correct number of times. Now take a couple minutes and try using different combinations of starting values and relational operators. It can get confusing. I often tell students that it is very easy to write code that looks like it does one thing, when it actually does something different. That is why I recommend that you pick one particular combination and stick with it.

**Chapter 3**

**for Loops**

Now that you have mastered while loops, I want to show you another form of the pretest loop. This second form is called the *for loop*. I usually find that students understand the while structure pretty well. However, the for structure sometimes gives students a problem. I believe this is because, although the two structures do the same thing, the syntax of the for structure is more like shorthand.

When the computer language was first written, someone noticed that every while loop did three essential things every time: initialize the counter variable, test the condition, and update the counter. In general, this would be structured as follows:

*<initialize variable>  
  
while ( <condition> )*

*{*

*<loop body>*

*<variable update>*

*}*

Programmers realized that this was using a lot of space in their programs, and it made it easy to forget one of the three essential parts. Therefore, the for structure was created. The syntax of the for structure is as follows:

*for ( <initialize> ; <condition> ; <update> )*

*statement;*

Notice how each of the three essential parts of the loop structure are contained in the for statement. Each of these parts is separated by a semicolon. Once again, the body of the loop consists of only one statement. If you want your loop body to contain more than one statement, you must use a statement block.

The reason I think students find this confusing at first is because they have a hard time remembering when each of the three statements is executed. As an example, I will rewrite the first while loop that we looked at previously:

for (a = 1; a <= 2; a = a + 1)

Console.Out.WriteLine("Hello");

When the computer gets to the for statement the first time, it assigns the value 1 to the variable *a*. Next, the computer will test the condition to see if it is true. Since 1 is less than 2, the loop body is executed. The word *Hello* is displayed on the screen, and since this is the end of the loop body, the update statement is executed. That means that 1 is added to the a variable. The program continues by testing the condition. Again it evaluates to true, and the loop body executes. After the word is displayed on the screen, the update statement is run and *a* becomes 3. Finally the condition is tested, and this time, it evaluates to false. When this occurs, the loop body is skipped and program execution continues with the next statement after the loop body.

Do you see what I mean about when each of the parts of the for statement are run being confusing? The initialize statement is only executed once. The condition is executed before each iteration of the loop, and the update is executed at the end of each iteration. But all of these statements are on the same line of code! Again, I do not think this is difficult to understand; it is just some shorthand that you will need to get used to.

Let's look at an example of a loop that does something other than just print out some words to the screen a bunch of times. The following method will ask the user for a certain number of integers, as specified in a parameter that is passed, and return the average of those numbers:

public static double calcAve(int number)

{

String strValue;

int total, value, count;

double ave;

total = 0;

for(count = 0; count < number; count++)

{

Console.Out.Write("Enter a value: ");

strValue = Console.ReadLine();

value = Convert.ToInt32(strValue);

total = total + value;

}

ave = (double)total / count;

return ave;

}

Before we talk about the loop itself, I want to point out the ++ operator in the for loop above. C# allows us to use this operator to replace a statement like, count = count + 1. I find myself using ++ a lot when I'm writing for loops, just because it makes the line of code so much shorter.

This method uses the for loop to control the number of values read in. The loop uses a counter variable named *count*. Also, this loop has another variable called *total* that is used to hold the running sum of all of the numbers that have been entered as the loop progresses. Similar to our counter variable, this type of variable has a special name, an *accumulator variable*. While it is not important to know the definitions of these two special kinds of variables in order to write a program, knowledge of their special functions can help you to design your programs and write your code.

Notice how the two different variable types work in different ways. Counter variables are used to count the number of times a loop is executed. This means that they are usually updated, or incremented, by a constant value. On the other hand, accumulators hold a running total and are usually updated by a value that changes.

The example above used both the counter and the accumulator to calculate the average. Of course, I could have done my calculations with other variables, but I did it this way to make my point about the different kinds of variables. I also did this to show you that, while you do have a choice in starting your counter at 0 or starting it at 1, sometimes one way is a little more convenient than the other. In this example, I started my counter at 0. This was important because, by starting at 0, it forced me to make my condition strictly less than the number of times the loop was to be executed. This condition also means that when the loop is finished executing, the counter variable will contain the number of times the loop body ran. I can then use this variable directly as the denominator in my equation. Again, I realize that I could have used another variable in the denominator, but I wanted to do this to make this specific point.  
  
  
  
  
**Chapter 4**

**do Loops**

Thus far, we have focused on pretest loops. Pretest loops are very useful because there may be times when we don't want the loop body to execute at all. This can happen for a pretest loop when the condition is initially false. Take the following program as an example:

Console.Out.WriteLine("How many numbers do you

want to count?");

strNumber = Console.ReadLine();

number = Convert.ToInt32(strNumber);

for(count = 1; count <= number; count++)

Console.Out.WriteLine(count);

If your user does not want to count any numbers, then no message will be displayed on the screen.

On the other hand, there may be times when you know that the loop will need to be run at least one time. Often, this is the case if you are going to display a menu of choices to the screen. You know when the user starts the program that they will need to see a list of choices at least one time. In this case, you could use what is called a posttest loop. A *posttest loop* is a loop in which the body of the loop is executed once and then the condition is tested. C# allows for the do loop as its posttest loop. The syntax of the do loop is as follows:

*do  
statement;  
while( );*

Just as with the while and the for loops, the loop body consists of one statement. If you want to have more statements, you must put them inside a block statement. The following example shows a code segment that prints out the numbers from 0 to 5 to the screen:

int count = 0;

do

{

Console.Out.WriteLine(count);

count = count + 1;

}while(count <= 5);

So now you are probably starting to think, "Why are there three ways to loop?" or maybe, "How am I supposed to pick which loop structure to use?" The answer to this question can sometimes get confusing because the three looping structures are equivalent. That means that any code written using a do can also be written using a while. However, you will find as you gain more experience writing code, one way will make sense over the others.

For now, use these general rules. First, if you know that the loop body must be executed at least one time, use a do loop because it is the only posttest loop. Second, consider if you know exactly how many times your loop will be executed. This is called a *counted loop*. Every loop example that you have seen so far has been a counted loop. Even the example of the method that calculated the average was a counted loop. It is true that when we were writing our code, we did not know how many numbers the user would enter, but the number that the user entered was stored in a variable. Since we can always get the value of the variable, we did actually know the exact number of times the loop would be run.

It may help to show you an example in which we do not know how many times the loop will be run. The following code will print a number and then ask the user if they want to see the next value.

int count = 0;

String strAgain;

char again = 'Y';

while(again == 'Y')

{

count = count + 1;

Console.Out.WriteLine(count);

Console.Out.WriteLine();

Console.Out.Write("Would you like to see

the next value? ");

strAgain = Console.ReadLine();

again = Convert.ToChar(strAgain);

}

As you can see, we do not know how many times this loop will be run. The number of times is totally dependent on the answer the user provides. This is known as a *conditional loop*. When you need to write a conditional loop, use the while structure. On the other hand, if the loop is a counted loop, it will probably be easier to write it as a for loop. Remember, these are not hard rules. You can use either loop in either case, but the rules do provide some guidance on how to choose between structures.

This example also uses a special kind of value called a sentinel. A *sentinel value* is a value that signals the computer to stop executing the loop. Sentinels are used in conditional loops. For example, consider a program that asks the user to enter a list of test scores to be averaged, and the user can enter as many as they wish. Since the programmer does not know how many the user will enter, they would write a conditional loop. They would also need to have a sentinel value that the user could give to signal the end of the data. A good sentinel value in this case might be -1, since there is no way you can get a -1 on a test.

Although the counting example was used to show you a conditional loop, it also shows my point about which type of loop to choose. Personally, I would never use a while loop to solve this problem. Notice how my code must give the again variable a value in order to enter the loop? This forces the loop body to be executed once. In this case, it would be better to use our posttest loop, the do loop. Don't be nervous about all of these choices that you have been given. The more examples you see and the more programs you write, the more comfortable you will get deciding on which structure to use.

One final point that I would like to make concerning all of these looping structures is regarding when to exit a loop. The only time you can leave a loop is after the condition is evaluated. That means that if something happens in the middle of the loop to make the condition false, the remainder of the loop body will still be executed. Here is an example to show this point:

int count = 0;

while(count < 1)

{

Console.Out.WriteLine("count = " + count);

count = count + 1;

Console.Out.WriteLine("Now count = " + count);

}

In this example, the count variable starts at the value *0*. Since 0 is less than 1, the loop body is entered. The first output statement shows that *count* is equal to 0. Next, 1 is added to *count*, making it equal to 1. At this point, the condition is no longer true, because 1 is not less than 1. However, we have not yet reached the end of the body of the loop, so the condition is not being tested. The second output statement is run, and it shows that *count* is now equal to 1. Now we reach the end of the loop, the condition is tested, and it evaluates to false.  
  
  
  
  
**Chapter 5**

**Summary**

In this lesson, you learned about the many different forms of the repetition structure. You learned pretest loops tested a condition first before entering the body of the loop. For this reason, these loops are called pretest loops. There are two different pretest loops: while and for. Although either can be used, generally while loops are used for conditional loops, and for loops are used for counted loops. You also learned that conditional loops use a sentinel value to signal to the computer to stop iterating. This lesson also showed you the posttest loop structure called *do*. The do loop is used when you know that the loop body must be executed at least one time.

You will find that these looping structures will help to make your programs much more useful, especially for redundant calculations. In addition, you will see how handy they are in the next lesson when you learn about arrays. Arrays are special variables that are a group of memory that holds related values. Each spot in memory can be referenced using the same name. While this may sound confusing right now, you will see how useful these variables, combined with your knowledge of loops, actually are.  
  
  
  
**Supplementary Material**

|  |
| --- |
| [Explanation of C# Loops with Examples](http://www.softsteel.co.uk/tutorials/csharp/lesson9.php)  http://www.softsteel.co.uk/tutorials/csharp/lesson9.php |
| This Web page gives a nice description of each type of loop, complete with a short example of each. |

**FAQs**   
  
**Q:** You didn't show anything about nested loops. Is this legal in C#?  
  
**A:** Yes, just like you can nest one if inside of another if, you can also nest one loop inside of another loop.

**Assignment**   
  
  
Write a program that uses a loop to ask the user for a list of numbers. The program should stop looping when the user enters *999*. When the user enters that value, your program should display the sum of all the values that were entered. Do not include the value 999 in the sum.

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