Top of Form



**Introduction to C# Programming: Lesson 3**

**Chapter 1**

**Introduction**

In the previous lesson, you started getting used to the C# compiler and the general structure of a C# program. You even wrote your first C# program to print text to the screen. This program confirmed your ability to program in C#, but let's be honest, the program wasn't exactly something that you will be able to sell for a million dollars. Printing text on the screen is an important part of every program, but the ability of a program to do calculations and store results will greatly enhance its usefulness.

In this lesson, you will learn about the different data types available in C#. Then you will learn how to declare and use a variable. To make your programs more useful, you will then learn how to do numerical calculations, convert from one data type to another, and read input from the user.  
  
  
  
  
**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.

**Data Types and Variables**

In the previous lesson, you wrote a program that printed a string of words to the screen. This set of words is known as a *literal constant*. A literal constant is something that never changes. However, you may want to write a program that uses a value that can change as the program runs. To do this, you would need a *variable*. A variable is a location in memory that is named. This location can hold different values as the program runs. For example, imagine that you are writing a program that will keep track of the number of miles that you drive in your car each week. One week you may only drive 100 miles, but the next you may drive 1,000. This variable will be able to hold these different values when the program is run at different times or even during the same execution of the program. Hopefully, you see that this is a very powerful tool.

When you decide to use a variable in your C# program, you will first need to determine what kind of value it will store. In general, you will probably want to store a single character, a string of characters, a whole number, or a number with a decimal point. Each of these values is stored in a different format and takes up a different amount of memory. For this reason, if you want to use a variable to store this information, you need to declare to C# the variable's *data type*. The following table shows the most common data types, their names, the amount of space that they require, and the range of possible values.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 3.1. Different C# data types** | | | |
| **Type** | **Bytes** | **Description** | **Range** |
| int | 4 | whole number | -2,147,483,648 to  2,147,483,647 |
| double | 8 | floating point number | 5x10^-324 to 1.7x10^308 |
| char | 2 | a single Unicode character | --- |
| string | --- | a string of zero or more characters | --- |

Before we go any further, I just want to make a quick note about the range of values column. The reason for this range is due to the amount of memory and the format of the data type. Also, you will notice that there is no range given for the char data type. A char variable stores characters as Unicode characters. You may remember the discussion of ASCII earlier in this course. ASCII allows up to 128 characters to be stored. This is plenty of characters for the English language, but not enough for other languages, like Chinese. To accommodate these extra characters, C# uses Unicode, which allows for up to 65,536 characters to be stored.

Strings are a different sort of data type. As you might guess, a string with 200 characters takes up more memory than a string of two characters. This is why there is no number of bytes listed for the string data type. Also, there is no range of values for a string. This is because it is not clear how to say one string is higher than another. Strings can be compared to one another alphabetically. This will be discussed later in the course.

While these four data types are the ones you will use most in this course, there are others. There is a data type that allows you to store whole numbers using the same amount of memory as an int, except that the values can only be positive. These numbers are said to be unsigned, thus the name of the data type is *uint*. You can also store whole numbers using more memory, thus giving you a larger range of values. This type is known as *long*. There is also a type called *ulong*. You can also store whole numbers in a smaller amount of memory than an int. These types are *short*, *ushort*, *byte*, and *sbyte*. C# also provides a data type to store decimal numbers that are smaller than doubles. These variables will use half as much memory, and the type is called a *float*. There is a data type that is intended to be used with monetary values called *decimal*. The final data type that I will discuss is called *bool*. A bool variable takes up only one bit of memory and therefore can only store one of two values, true or false. As I said before, all of these data types are important, but in this course, you will mainly use the four types from the table above.

Now that you know the different types that are possible for your variable, you need to know how to declare a variable. A *variable declaration* is a way of telling the compiler the data type that the variable will store and the name of the variable. For example, if you wanted to create a variable that will store your age, you might type:

int age;

Notice that you will type the data type first, followed by the name of your variable. Recall from the previous lesson that every statement in C# ends with a semicolon. This line of code would translate into *go out to memory and give me a spot that is big enough to hold an int and call that spot age*. It is customary in C# to give your variables names that tell what they represent. It is my strong belief that if you choose good variable names, your code will be easier to write, easier to read, and easier to debug. Please make your life as easy as possible and get used to choosing good variable names.

Well, you now have a spot in memory where you can put a whole number. But there is no value there right now. To have a value placed in that memory location, you will need to use the *assignment operator*, =. So, if you wanted to put the value *30* into your variable named *age*, you would type:

age = 30;

This may look very simple and natural to you, and it should. However, I want you to get used to reading this line of code from right to left. That is, I would translate this statement into English as *put the value 30 into the variable named age*. If you get used to this now, it will help to make some things easier later on in the course.

Well, I'm not sure about you, but I tend to be a little on the lazy side from time to time. When I get to this point of talking about variables, I usually start thinking to myself, isn't there a way to do those two steps in one step? After all, I would think that most of the variables that I create will need to have an initial value. And the answer is yes! You can both declare and *initialize*, give the variable an initial value, all in the same step. This would be done by typing:

int age = 30;

This line of code translates to *go out to memory, get me a spot big enough for an int, call it age, and put the value 30 there*.

If you want to declare variables of type float, char, or string, you will do it in the same fashion. That is, you will always declare by typing the data type and then the variable name.

While declaring variables may be the same between different data types, the assignment is a little bit different for chars and strings. The assignment operator and order of words will be the same. However, characters must be surrounded with single quotation marks (' ') and strings must be surrounded with double quotation marks (" "). For example, if you want to create a string variable named *message* that contains the text *I love C#*, you would type:

string message = "I love C#";

And if you wanted to create a char variable named *grade* that contains the character *A*, you would type:

char grade = 'A';

**Chapter 3**

**Using Variables**

The one thing that you will want to do most often with a variable is display the value that it is storing. As you learned in the previous lesson, to display something to the screen, you use the WriteLine() method. When you wanted something literally printed to the screen, you put that text inside quotation marks and all of that inside the parentheses in the method call. In a similar way, if you want to print the value that some variable is storing, you will put the name of the variable *without* quotation marks inside the parentheses. Look at the following code for a Main() method:

public static void Main()

{

double cost = 12.95;

System.Console.Out.Write("The cost of that item is ");

System.Console.Out.WriteLine(cost);

}

This code would have the following as output:

The cost of that item is 12.95

Before you go any further in this course I want to tell you that from now on, I will almost always just show you *only* the Main method. This is done to save space. Recall from the previous lessons that every method must be contained inside a class. To have this Main method contained inside a class, you would only need to add a class header and an opening curly brace ({) to open the body of the class. Next, you would type the code above, and finally, you would type a closing curly brace (}) to close the body of the class.

With all of this in mind, you may now be looking at the output and wondering why everything is displayed on the same line. The reason is because the method used to display the text uses the Write() method. This method displays something to the screen, just like the WriteLine() method. However, this method does not advance the cursor to the next line after it is done displaying the text.

OK, so now you might be looking at this code thinking, "Every time I want to display literal text and the values of variables, I need to use two different statements? That seems like a lot of work. Isn't there an easier way?" And I am happy to say that yes, there is a way. To do this, you need to use a *format string*, a string of characters that contains one or more placeholders. A *placeholder* is a number inside a pair of curly braces. This placeholder will be replaced by the value of the variable that is listed later in the statement. You will always start your numbering at zero. To show you what I mean, I will rewrite the above code using a format string:

public static void Main()

{

double cost = 12.95;

System.Console.Out.WriteLine("The cost of that item is ¬

{0}", cost);

}

If you wanted to print out the values of more than one variable inside of a line of text, you would use more than one placeholder. For example, look at the following code and see if you can guess what will be displayed:

public static void Main()

{

double cost = 12.95;

string name = "picture frame";

System.Console.Out.WriteLine("The item named {0} costs ¬

{1}", name, cost);

}

Did you figure that the output would be:

The item named picture frame costs 12.95

If you did, you were correct. This may take some getting used to, but I know you can get it with some practice!

As you continue to write programs, you will start to find that it is necessary to do some mathematical calculations. Computer programs are great for doing math, because they can do it very quickly and very accurately. The following table shows the five major mathematical operators:

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 3.2. C# mathematical operators** | | | |
| **Operator** | **Description** | **Example** | **Result** |
| + | Addition | 18 + 4 | 22 |
| - | Subtraction | 27 - 3 | 24 |
| \* | Multiplication | 4 \* 8 | 32 |
| / | Division | 31 / 2 | 15  (Note: Since the two operands are ints,  the result must also be an int.  The fractional part is removed.) |
| % | Modulus or Remainder | 31 % 2 | 1  (Note: This is the remainder when 31 is divided by 2.) |

When you do more than one mathematical operation in a single statement, the operations will be performed based on the operator precedence or order of operations. This order is:

1. Parentheses
2. Multiplication, Division, Modulus
3. Addition, Subtraction

For example, the expression 1 + 2 \* 3 would result in the value 7. This is because 2 \* 3 is done first, yielding 6. Finally, 1 is added to 6 to get 7.

You may use these operations either as part of an assignment statement or inside the Write() or WriteLine() method. Can you figure out what the following code will display?

public static void Main()

{

int number1 = 4;

int number2 = 6;

int sum = number1 + number2;

System.Console.Out.WriteLine("The sum of {0} and {1} ¬

is {2}", number1, number2 , sum);

System.Console.Out.WriteLine("The product of {0} and ¬

{1} is {2}", number1, number2, number1 \* ¬

number2);

}

The output would be:

The sum of 4 and 6 is 10  
The product of 4 and 6 is 24

Did you get it right? Try to type this Main method into a class named *Lesson3* and see if you can get it running. You may want to play around with the code a little bit and see if you can get it to print out different combinations of literal text and values of variables. You might also want to try to use the different operators to see if your math is still as good as it used to be.  
  
  
  
  
**Chapter 4**

**Data Conversion and Input**

So far this lesson, a big deal has been made about the data type for a variable. You have learned that your variables must have a declared data type and that that variable can only hold data of the specified type. You have also learned that there is a set of mathematical operators that you can use on your numerical values. During all of this time, something may have crossed your mind. What if you try to perform some operation on values of different types. Maybe you want to add an int and a double, or multiply a double and an int. Is this possible? The answer is yes.

C# gives you the ability to perform operations on numbers that do not have the same type. However, the result can only be stored as one type. So which type is used? In the previous examples, and in all cases when an int and a double are used in the same expression, the result will be a double.

To help you remember this, you should think about what is being stored. Since a double can contain a fractional part, it would be more precise than an int. Therefore, C#, in an attempt to help you, will use this data type because of its increased precision. This is called an *implicit conversion*, C#'s automatic conversion from one data type to another. For example:

double result = 1.5 + 1;

will result in the value *2.5* being stored in result. But what if you wanted the result to be an int variable so that the result was stored as an int? You might try:

int result = 1.5 + 1;

This line of code will not compile. Instead, the compiler will give you the message, "cannot implicitly convert type 'double' to 'int'." This is the compiler's way of telling you that your answer is a double. It is actually trying to do you a favor, because if it allowed this statement to be performed, you would lose some of the information, the 0.5 to be specific.

Sometimes you will not care about that fractional part and will actually want C# to store the value as an int. This can be done by using a *cast*. Casting is the explicit conversion from one data type to another. To perform a cast, you will put the data type that you wish your result to have inside parentheses followed by the expression that you want to be cast. Using the previous example, the code would be:

int result = (int) 1.5 + 1;

This says to the compiler that the following expression will result in something other than an int; I want you to convert it to an int for me. Note that the value of the variable result here would be 2 because 1.5 as an int is 1. Remember C# truncates, or removes the part after the decimal point. And we all know that 1 + 1 = 2. You should note that when you use a cast, only the very next term is converted. That means that:

(int) 6.7 + 1 = 6 + 1 = 7

while  
6.7 + (int) 1 = 6.7 + 1 = 7.7

and  
(int) (6.7 + 1) = (int) 7.7 = 7

Hopefully by now you can see that variables are very useful to our programs. The variables you create will allow you to store different values while your program is running. In addition to simply storing these values, you can also do numeric computations and data conversions. You have seen a couple of short programs in this lesson that do all of these things: create variables, store values, do calculations, and display results. You can now use this knowledge and write programs that can do some very complex calculations. There is a problem, though. As of right now, you are using your program as a fancy calculator. Let me give you an example.

Let's say that you know someone who is an elementary school teacher. This teacher knows that you are learning how to program and asks you to write a program that will allow the students to practice their addition skills. Of course, you have the knowledge to do this. You would just write a program and use a couple of variables to store two numbers, calculate their sum, and display the result. This would be a very nice program. However, there is a problem. Your program would only be able to test the same two numbers each time the program is run. If you wanted your teacher friend to test different numbers, then they would have to know how to program with C#.

The easy way to solve this problem is to allow the user to pick which numbers are going to be added. This means that your program will get input from the user. To get input from the user, you can use the System.Console.ReadLine() method. This method will accept all of the characters that the user types in on the keyboard until they press the ENTER key. The ReadLine() method will then take all of those characters and return them to the programmer as a string. It is then up to the programmer to decide what to do with that string. As a programmer in this course, you will almost always store that string in a variable. Consider the following Main() method:

public static void Main()

{

System.Console.Out.Write("Enter your name: ");

string name= System.Console.ReadLine();

System.Console.Out.WriteLine("Hello, " + name);

}

Notice how the first line of method Main displays a message asking the user for their name. The second line then does a lot of work. It creates a space in memory for a string called *name* and calls on the ReadLine() method. The ReadLine() method then halts execution of the program to allow the user to type in some information. When the user presses the ENTER key, the ReadLine() method then returns that set of characters, and they are stored in the name variable. The final line of the program then displays some text and the value of the name variable.

OK, so let's go back to the example with our teacher friend. With the knowledge that you have now, you decide that your program will ask the user for two values and then compute the sum. Your Main() method might look like this:

public static void Main()

{

// Get input from the user

System.Console.Out.Write("Enter the first number: ");

string num1 = System.Console.ReadLine();

System.Console.Out.Write("Enter the second number: ");

string num2 = System.Console.ReadLine();

// Calculate the sum

double sum = num1 + num2;

// Display the result

System.Console.Out.WriteLine("The sum is " + sum);

}

But there is a problem with this code. Do you see it? If you look closely, you will see that the sum is a double variable, and num1 and num2 are string variables. This code will not compile because as you saw earlier in this lesson, you cannot store string data in a numerical variable.

Let's say that you decide to make the sum a string variable. If you do this, the program will indeed compile, but you will get an interesting result. When you run the program, if you type in the value *3* for the first number and *6* for the second number, the result from the program will be 36. Now we all know that 3 + 6 is 9, right? So what happened?

Adding two strings together doesn't really make much sense. Instead, the computer did what is called *concatenation*, which just means it put the second string onto the end of the first string. Concatenation will come in handy later, but for now, just realize that this is not what we want this program to do right now.

So let's try making the num1 and num2 variables into double variables. But, by definition, the ReadLine() method will return a string. Therefore, we will need to convert the string into a double. You saw earlier how casting converted numbers between types, but strings are special things in C#. In fact, you cannot cast a string into a number. Instead, you need to use a method for conversion. C# provides for us a class called *Convert*. You will learn about classes later, but for now, just think of a class as way to organize a number of methods. Class Convert contains methods to convert strings into any other data type. Some of the methods that are important for this course are listed in the table below:

|  |  |
| --- | --- |
| **Table 3.3. C# conversion methods** | |
| **Method Name** | **Description** |
| ToChar(*some string*) | Converts the specified string into a character |
| ToDouble(*some string*) | Converts the specified string into a double |
| ToInt32(*some string*) | Converts the specified string into an int<> |

To use any of these methods, you need to put the name of the class first, then a dot (.) and the name of the method that you want to use. For example:

int age = System.Convert.ToInt32(ageAsString);

will attempt to convert the string variable called *ageAsString* into an integer and store that value in an int variable named *age*.

By applying this idea to our problem for our teacher friend, the program now becomes:

public static void Main()

{

// Get input from the user

System.Console.Out.Write("Enter the first number: ");

string num1String = System.Console.ReadLine();

double num1 = System.Convert.ToDouble(num1String);

System.Console.Out.Write("Enter the second number: ");

string num2String = System.Console.ReadLine();

double num2 = System.Convert.ToDouble(num2String);

// Calculate the sum

double sum = num1 + num2;

// Display the result

System.Console.Out.WriteLine("The sum is " + sum);

}

One other thing that I want to point out about the code above: Notice how the last line of code does not use a placeholder. Instead, a plus sign (+) is used. This is another use of concatenation, putting a number onto the end of a string.

You may want to try to open Notepad or another text editor and type in this code. If you do this, please be sure to put this method inside a class. That is, you will need to have a class header and the opening curly braces ({). Then type the code above and finally type a closing curly brace (}) to close the body of the class. Compile and run the program.

**Chapter 5**

**Summary**

This lesson has opened a whole new world of programming to you. When you first started this lesson, you knew how to write a C# program that could display text on the screen. This was a wonderful thing, but you probably found out that there wasn't very much that you could make your programs do. This lesson described some different data types and showed you how to declare variables and assign a value to them. You next learned how to use mathematical operators to have your programs do simple, or very complex, mathematical problems. Finally, this lesson showed you how to interact with your user.

Armed with this new knowledge about C#, you can now write very useful programs. These programs can be written to be very friendly, even ask the user for their name and then politely ask the user for values using their name. It also allows your programs to be run many times using different sets of data each time. As you worked through this lesson, you may have thought about writing a program that calculates your taxes, computes the gas mileage for your car, or balances your checkbook. All of these things are very possible with what you have learned so far.

Now you have the basic knowledge to write very useful programs. However, before we go any further in the course, I want to introduce you more formally to methods. You have already used several: Write(), WriteLine(), ReadLine(), and others. But in the next lesson, I will show you how to write your own methods. This may not seem like a significant thing, but it is a very useful thing to do. By putting your code into methods, your code will become more organized and easier to read, write, and use. This is also a way to get you used to object-oriented programming, which is something I want you to accomplish by the end of the course.  
  
  
  
**Supplementary Material**

|  |
| --- |
| [All C# Variable Types](http://www.softsteel.co.uk/tutorials/csharp/lesson4.php)  http://www.softsteel.co.uk/tutorials/csharp/lesson4.php |
| This site provides a complete list of all possible data types and their complete range of values. |

**FAQs**   
  
**Q:** Converting a string to character uses ToChar(), converting to a double uses ToDouble(), why don't we use ToInt() to convert to an int?  
  
**A:** As you learned in the lesson, there are many different data types used to store whole numbers and floating point numbers. Since there are many data types for whole numbers, there are many conversion methods for converting strings to whole numbers. The ToInt32() method will attempt to convert the string to a 32 bit signed integer, which is how the data type int is stored in memory.  
  
**Q:**How can I fix it so that when I print out my cost, it shows two decimal places instead of one. That is 123.40, instead of 123.4.   
  
**A:**There are many ways to format output in C#. One way is to use the placeholder method that is discussed in Lesson 03, Chapter 03 and include a formatting code along with the placeholder number. For example, if you want to format the value of the cost variable as currency, you can use the currency code "C", as in:   
  
System.Console.Out.WriteLine("The price is {0:C}", cost);   
  
There are many different formatting codes, which can be found at the MSDN.com site at:   
  
<http://msdn.microsoft.com/en-us/library/dwhawy9k.aspx>  
  
**Q:**: I accidentally tried the following code (calling WriteLine without using 'Out') and it worked:  
  
System.Console.WriteLine("Hi");   
  
So why do we need the 'Out' if it works anyway?  
  
**A:**Technically speaking, we aren't required to use the 'Out' when we call WriteLine. In fact, you probably also noticed that we didn't use the 'In' with ReadLine... but we could have. Try it for yourself!  
  
These are just two of the places where the developers of the C# language gave us a little bit of flexibility. I guess they figured that we'd probably be doing these things an awful lot and just created these little aliases for us to use. So with that in mind, feel free to use the 'Out' or the 'In', or neither. It's completely up to you.

**Assignment**   
  
  
A friend of yours has a job installing carpet. To better serve their customers, your friend asks you to write a program that will calculate the cost of carpeting a client's floor. Write a program that will ask the user for the length and width of the room in feet. Next, ask the user for the cost of the carpet per square foot in dollars and cents. Using this information, calculate and display in a friendly, descriptive sentence, the area of the room (area = length \* width) and the cost of carpeting that room (cost = area \* price per square foot).

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