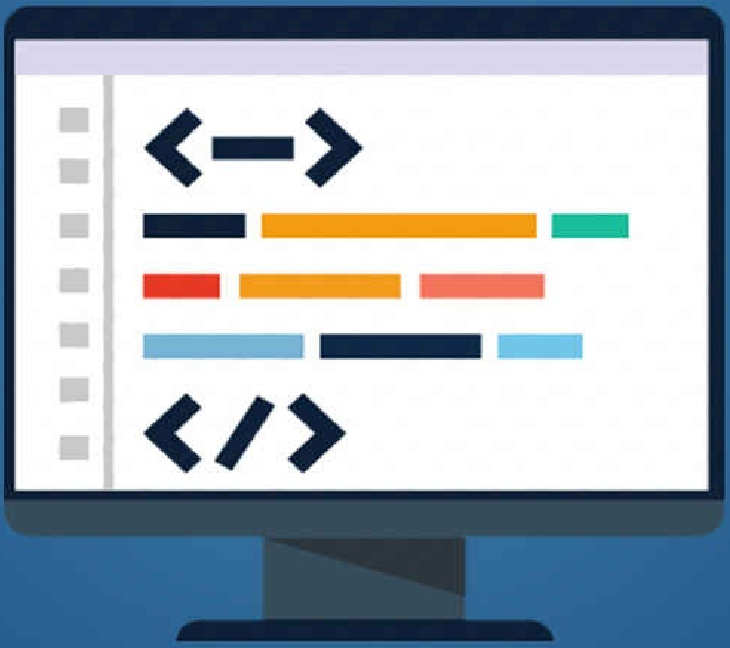
C#

CRASH COURSE



Learn The Basics Of C# Programming

LIFE-STYLE EBOOKS

C# CRASH COURSE

Beginner ’ s Course To Learn The Basics Of C# Programming Language

Introduction

I want to thank you and congratulate you for downloading the book, *"C# Crash Course—Beginner's Course To Learn the Basics of C# Programming Language"*

This book contains proven steps and strategies on how to master the basics of C# in just one week.

This eBook will teach you the fundamental ideas and concepts related to the C# programming language. It will explain programming lessons using clear words and practical examples. Actual C# codes and screenshots are also provided to help you understand the lessons. Additionally, this book will arm you with the skills, knowledge, and techniques necessary for becoming an excellent C# programmer.

After reading this material, you'll become familiar with the C# language and the command console. You'll know how to send information to program users and how to obtain their inputs. You'll know to compile your programs and how to format your codes. You'll know how classes, variables, functions, methods, and data types work. Simply put, you'll be a skilled C# programmer, and you can be so in as little as 7 days!

Thanks again for downloading this book, I hope you enjoy it!

Table of Contents

Chapter 1: The Core Concepts of C#

According to computer experts, the best way to learn a programming language is by writing and analyzing codes. Thus, you'll write an actual computer program before studying the language itself. This approach will help you to understand how C# works. Also, you'll become familiar with the syntax and structures of this language.

Your First Program

Here's the code that you need to type:

```
class HelloCSharp
{
   static void Main(string[] args)
   {
      System.Console.WriteLine("Hello C#!");
   }
}
```

This program prints "Hello, C#!" onto your computer's default output device. It's still a "raw" program so you can't run it yet. For now, let's just analyze its structure. The next section will teach you how to compile and run programs using a command prompt and a programming environment.

How Does It Work?

This sample program consists of three parts:

1. The class definition;
2. The method definition; and
3. The method's contents.

Let's discuss each part in detail:

The Class Definition – The initial line of the program defines a class named HelloCSharp. The most basic class definition has two sections: (1) the word "class" and (2) the name of the new class. In this example, HelloCSharp serves as the class's name. You'll find the contents of this class inside the pair of braces (i.e. { }).

The Method Definition – This part is located at the third line of the program. Here, you defined a method known as Main(). All C# programs use the main() method as their starting point. Here's the syntax that you need to use:

```
static void Main(string[] args)
```

As you can see, Main() needs to be void and static. The arguments (i.e. the part inside the parentheses) are completely optional. That means you can simplify the Main() function into:

```
static void Main()
```

Important Note: Your program won't run if you'll enter the Main() method incorrectly.

The Method's Contents – You'll find the contents of any method after its name. In general, a pair of braces encloses those contents. The fifth line of the sample program uses an object (i.e. System.Console) and a method (i.e. WriteLine()) to display a message on your computer's console. As mentioned earlier, the message is "Hello, C#!"

Important Note: When working on Main(), you may arrange C# expressions and statements according to your preferences. You won't get any error message even if you arrange those expressions and/or statements randomly. However, your completed program will run those entries according to their position inside the method's body.

The Rules You Need To Remember When Writing Codes

C# – A Case-Sensitive Computer Language

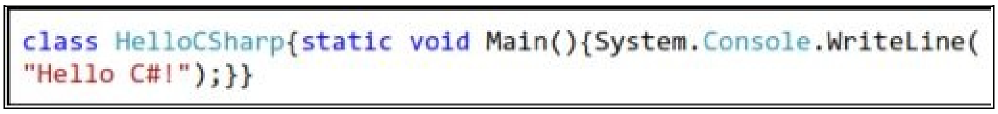
This language is case-sensitive. That means you need to be careful with letter capitalization when typing C# codes. Basically, C# treats "power", "Power", and POWER as three different objects.

Important Note: You need to remember this rule. It applies to all of the elements of any C# program (e.g. variables, constants, classes, etc.).

Use the Proper Format

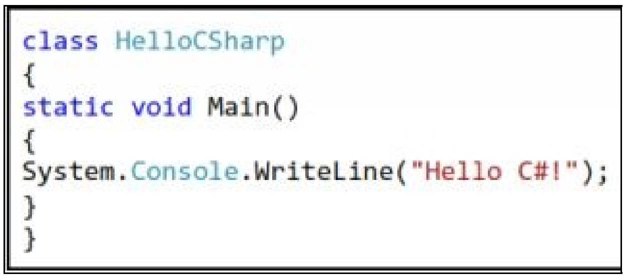
You need to add tabs, spaces, and newline characters to your code to improve its readability. These characters, which are ignored by the C# compiler, give an excellent structure to your source codes.

For instance, you may remove the newline characters from the HelloCSharp code:



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

As an alternative, you may remove the tab characters from it:



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

The last two examples work fine in terms of compilation and program execution. However, they are difficult to analyze, modify, understand, and maintain.

Important Note: Make sure that all of your programs have formatted codes. This way, you can guarantee the readability of your codes.

How to Format Codes Properly

Here are the rules that you need to follow when formatting your codes:

- You may indent methods within the class definition. C# allows you to use any number of tab characters (i.e. the character you'll get after hitting the Tab key of your keyboard) when formatting source codes.

- You should indent the method's contents within its own definition.
- Place the opening brace (i.e. {) right under the class or method to which it belongs. Also, it would be best if it will be the only character on its line.
- Make sure that the closing brace (i.e. }) is vertically aligned with the opening brace.
- The names of your class should begin with an uppercase letter.
- The names of your variables should start with a lowercase letter.
- The names of your methods should begin with an uppercase letter.
- Indent codes that are written inside another code.

Class Names

In C#, each program has one or more class definitions. Programmers often store each class definition inside a separate file that corresponds to the class's name. Also, the extension of these files should be ".cs". You can compile different classes into a single file without experiencing any technical problems. However, you'll have a hard time navigating your codes.

More Information about C#

The C# programming language is advanced, modern, multipurpose, and object-oriented. It has some similarities with other languages such as C, C++, and Java. The main difference is that C# offers easier programming and simplified syntaxes.

A C# program has one or more .cs files, which hold data types and class definitions. You need to compile those .cs files to get executable codes (i.e. files that end with .dll or .exe). For instance, if you will compile the HelloCSharp program, you'll get an executable file named "HelloCSharp.exe".

The Things You Need to Create C# Programs

Before you can write programs using C#, you need two important things—a text editor (e.g. Notepad) and the .NET framework. The text editor will help you in writing and editing codes. The .NET framework, on the other hand, will help you in compiling and executing your programs.

What is .NET?

Basically, .NET is a framework designed for the development and execution of computer programs. Most of the modern Windows systems have .NET as a built-in framework. Thus, you won't have to install any software onto your computer if you are using Microsoft's modern operating systems.

If you are using an old Windows OS, you need to download the .NET framework first. You may go to this site and get the most recent version of .NET.

Important Note: Make sure that your computer has .NET before reading the rest of this book. Otherwise, you'll have problems compiling and executing the sample programs that you'll see later.

Text Editors

You should use a text editor to write, edit, and save source codes. You can use any text editor installed on your computer. If you don't want to download additional programs, you may simply use "Notepad" (i.e. the pre-installed text editor of Windows computers).

Compiling and Executing Programs

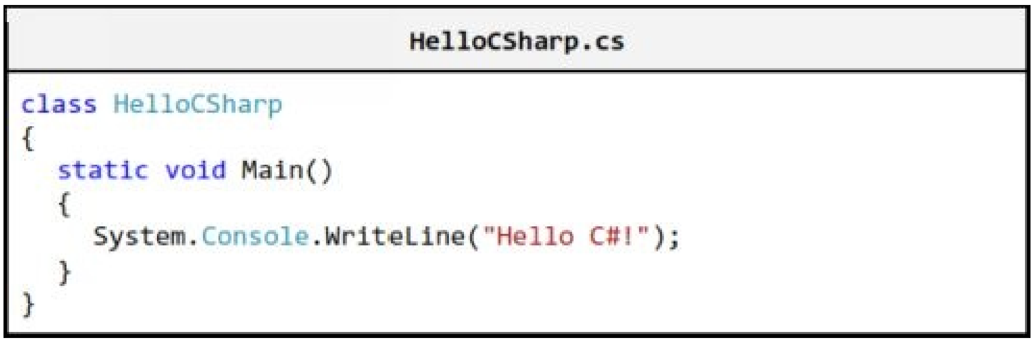
Now, it's time to compile and run the HelloCSharp program you've seen earlier. Here are the things you need to do:

1. Generate a ".cs" file and name it as "HelloCSharp".
2. Write the source code inside that .cs file.
3. Compile the code to get a .exe file.
4. Run the resulting program.

Turn on your computer and let's start writing some codes.

Important Note: This eBook assumes that you are using a Windows computer.

Here's the code of the HelloCSharp program:

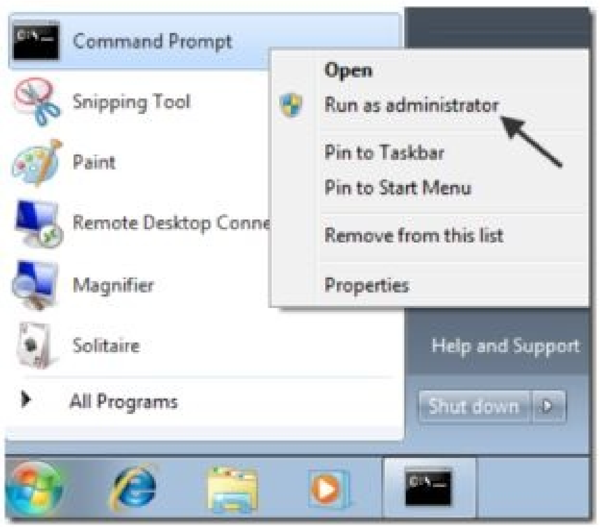


HelloCSharp.cs

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

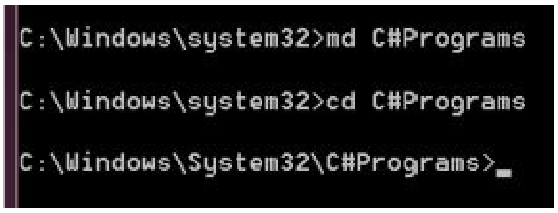
Writing a C# Program Using the Command Prompt

First, you need to launch the "Command Prompt" (also known as "Command Console"). If you are using Windows 7 or Windows 8, you can click on the Start button and run a search for "cmd". Check the search results and look for the program named "Command Prompt". Right-click on the program and choose "Run as administrator" from the menu. Here's a screenshot:



Important Note: If you will launch the Command Prompt by double-clicking on it, some of its important features might be unavailable.

Next, you need to generate a file directory so you can store the resulting program. Type "md" followed by the directory's name. Then, enter "cd" to get inside the new directory. Here's a screenshot:

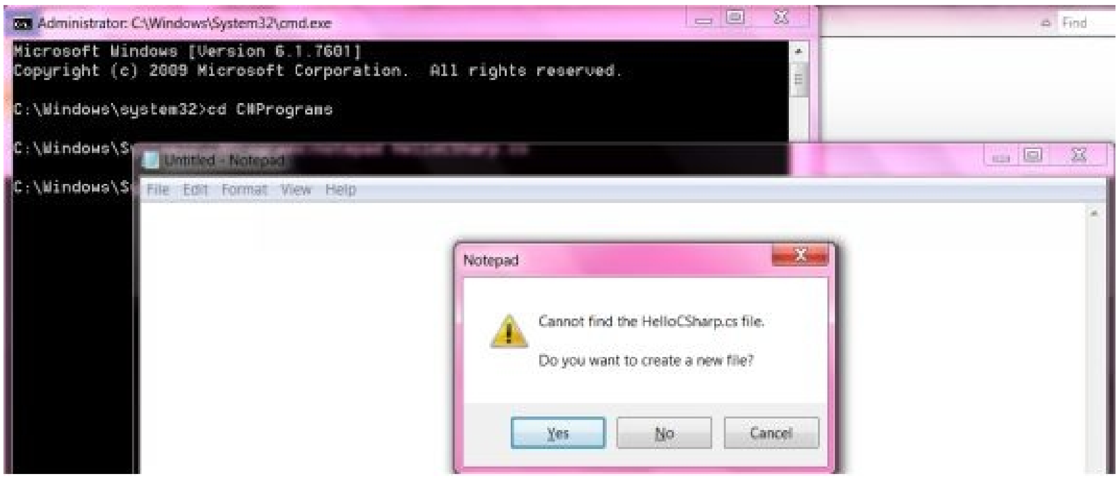


```
C:\Windows\system32>md C#Programs

C:\Windows\system32>cd C#Programs

C:\Windows\System32\C#Programs>_
```

For this example, the name of the directory is "C#Programs". The "cd" command changed the active directory to "C:\Windows\System32\C#Programs". Now, you're ready to create the .cs file. Go back to the Command Prompt and issue the following command:

```
notepad HelloCSharp.cs
```

After running that command, your screen will show you this:



```
Administrator: C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Windows\system32>cd C#Programs

C:\Windows\S

C:\Windows\S
```

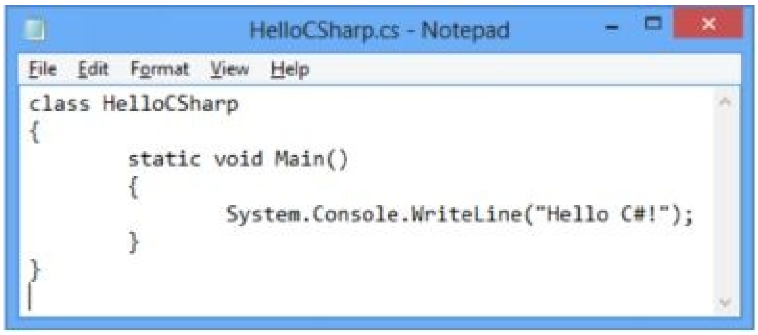
Untitled - Notepad

Notepad

Cannot find the HelloCSharp.cs file.

Do you want to create a new file?

Your computer will tell you that HelloCSharp.cs doesn't exist. Also, it will ask you whether you like to create a new file. Click on the "Yes" button and copy the program's code onto the Notepad application. The screenshot below will help you with this step:

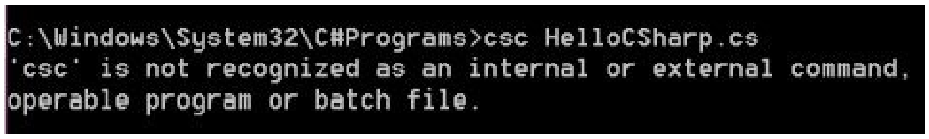


```
class HelloCSharp
{
        static void Main()
        {
                System.Console.WriteLine("Hello C#!");
        }
}
```

Press CTRL+S to save the file. Then close the text editor by hitting the red X or pressing ALT+F4. Now, you'll see the program's code inside the "C#Programs" directory.

Compiling the Code

At this point, you can compile the source code using "csc" (i.e. the built-in compiler of the C# language). Enter "csc HelloCSharp.cs" into the Command Prompt. Since it's your first time using the compiler, you might get an error message similar to this one:



```
C:\Windows\System32\C#Programs>csc HelloCSharp.cs
'csc' is not recognized as an internal or external command,
operable program or batch file.
```

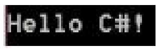
To solve this problem, you just need to specify the location of the C# compiler. Here's the required directory for Windows 7 computers: *C:\Windows\Microsoft.NET\Framework\v4.0.30319*

Important Note: The numbers at the end of the directory might change depending on the OS version you're using.

Thus, you should issue the following command:

C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc HelloCSharp.cs

The C# compiler will convert your .cs file into an executable program. To execute your first program, enter its name into the Command Prompt. You will get the following message:



```
Hello C#!
```

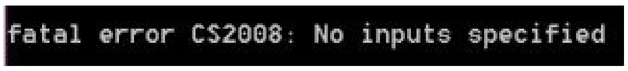
How to Change the System Path of a Windows Computer

Surely you don't want to type the compiler's complete directory each time you create a program. This part of the book will teach you how to set your computer so that it can automatically detect the compiler's location.

Here are things you need to do:

1. Access your computer's Control Panel and click on "System".
2. You'll see a new window. Search for "Advanced System Settings" and click on it.
3. Click on "Environment Variables" and look for a variable named "Path".
4. Click on the "Edit…" button and specify the compiler's directory. Again, that is: *C:\Windows\Microsoft.NET\Framework\v4.0.30319*

5. Hit “OK” to save the changes.
6. Now, close the current Command Prompt and launch a new one. Then, enter “csc”.
7. The screen should give you the following error:



```
fatal error CS2008: No inputs specified
```

8. That means your computer can automatically trigger the compiler. Once you specify a source code, your computer will generate .exe files for you.

Chapter 2: The Data Types, Variables, and Literals of C#

This chapter will focus on the data types and variables used in the C# language. By reading this material, you’ll know what these objects are and how you can use them in writing your own programs.

Variables – An Introduction

Typical programs utilize different values during their execution. For instance, you can develop a program that solves mathematical problems. Obviously, you need to enter different values while using this kind of program. Simply put, programmers don’t know which values will be entered by the user. Thus, they need to make sure that the resulting programs will accept any value that the user may provide.

When the user gives a value, you may temporarily store it in the machine’s RAM (i.e. random access memory). All of the values inside this memory may change. Because of this, programmers refer to such values as “variables”.

The Data Types

A data type is a set (or range) of values that share similar attributes. For example, the “byte” data type refers to the values within the “0 to 255” number range.

Attributes

Data types have the following attributes:

- Name (e.g. int, char, bool, etc.)
- Size (e.g. 1 byte)
- Default Value (e.g. 1, 2, 3, etc.)

The Different Types of Data in C#

C# supports the following data types:

- Boolean
- Characters
- Objects
- Floating-Point
- Decimals
- Integers
- Strings

Since these data types are pre-installed into the C# language, programmers refer to them as “primitive types”. Let’s discuss each data type in detail:

Boolean

You need to use “bool” (i.e. a C# keyword) to declare this data type. Boolean values can only be “true” or “false”. In the C# programming language, Boolean values are automatically set as “false”. Programmers use this data type to store the result/s of logical statements.

Characters

This data type can store single characters. You need to use “char” (i.e. another C# keyword) to declare this type. Write your “char” values within a pair of apostrophes (e.g. “I”).

Objects

Programmers regard objects as the most special type of data in the C# language. Basically, objects serves as the parent of other data types within .NET. This data type, which requires the “object” keyword during its declaration, may accept any value from other data types. You may think of objects as addresses that point to certain parts of your computer’s memory.

Real Type – Floating Points

In the C# language, “real type” refers to the actual numbers you’ve learned in mathematics. A floating-point real data type can either be “float” or “double”.

1. Float – Programmers call this subtype “single-precision real integer”. The default value of a float number is 0.0F (the “f” at the end is not case-sensitive). The letter “f” indicates that the value is a “float”.
2. Double – Programmers use the term “double-precision real integer” when referring to this subtype. Its default value is 0.0D (i.e. the letter “d” at the end is not case-sensitive.). C# automatically tags real numbers as “double” so you don’t really need to add “d” at the end of the value.

Real Type – Decimals

This programming language supports “decimal arithmetic”, a mathematical approach that uses decimal values to represent numbers. The main advantage of this approach is that it preserves its accuracy regardless of the values it works on.

C# uses the “decimal” data type to represent this kind of real number. This data type is 128 bits long and is accurate up to the 29th decimal value. Because of its excellent accuracy, these are used by programmers for financial computations (e.g. payments, taxes, interest, etc.).

Integers

This data type consists of 8 subtypes, which are:

SBYTE – An sbyte is a signed integer that is 8 bits long. That means it can contain up to 256 values (i.e. 2^8). Additionally, sbytes can be either negative or positive.

BYTE – This subtype contains unsigned integers that are 8 bits long. It is almost identical to sbyte. The only difference is that byte can never be negative.

SHORT – These integers are signed and 16-bits long.

USHORT – These are signed integers whose length is 16 bits.

INT – This is probably the most popular data type in C#. Programmers use this data type because it is perfectly compatible with 32-bit processors and big enough for typical computations. It is a signed integer whose length is 32 bits.

UINT – This 32-bit integer is unsigned. Thus, it can only be positive.

LONG – This integer is 64 bits long. It can be either positive or negative.

ULONG – This is similar to “long” integers. The only difference is that “ulong” can only assume positive values.

Strings

A string is a set or sequence of characters. C# requires you to use “string” (i.e. a keyword) when declaring values that belong to this data type. You have to write each string between a pair of quotation marks. This language allows you to perform various operations on strings (e.g. concatenation, character replacement, character search, etc.).

The Variables

You can use a variable to store, retrieve, and modify changeable data. As a programmer, you need to use variables in storing and processing information.

The Characteristics of a Variable

A variable has the following characteristics:

- Name (also known as “Identifier”)
- Data Type
- Value (i.e. the information you want to store).

Variables are sections of computer memory that have a name. They store values and allow a computer program to access their contents. You may place a variable inside the stack (i.e. the working memory of your computer program) or within the program’s dynamic memory.

Characters, Booleans, and Integers are known as value types since they keep their data within the program’s stack. On the other hand, strings, arrays, and objects are known as “reference types” because they don’t hold the values inside them. Instead, they serve as markers that point to the part of computer memory where the data is stored.

How to Name a Variable

You need to name your variables before using them. The variable’s name serves as an identifier: it helps the program in locating and specifying the variable. C# allows you to choose variable names freely. However, just like any other language, C# has some rules regarding variable names. These rules are as follows:

- You may use an underscore (i.e. “_”), letters, and numbers to create a variable name.
- You can’t use a number to start the name. Thus, FirstSample is valid while 1stSample is not.
- You can’t use C# keywords in naming your variables. This simple rule prevents compilation and runtime errors in your programs. If you really want to use keywords for your variables, you may begin the name using the “@” symbol (e.g. @bool, @int, @char, etc.).

How to Declare a Variable

You should declare a variable before using it. Here are the things you need to do when declaring a variable:

1. Specify the variable’s data type (e.g. char)
2. Specify the variable’s name (e.g. sample)
3. Set the variable’s initial value. This step is completely optional.

The format for variable declarations is:

data_type name = initial_value;

Here are some samples of valid variable declarations:

int year = 2000;

string = yourname;

char = test;

Important Note: You need to end your each of your C# statements using a semicolon. If you'll forget to add even a single semicolon character, your programs will generate undesirable and/or unexpected results.

How to Assign a Value

Just like any other computer language, C# allows you to assign a value to a variable. This process, known as "value assignment", requires you to use the equals sign. Programmers refer to this symbol as the "assignment operator". Write the variable's name on the left side of the operator. Then, specify the value you want to assign on the other side. Check the following examples:

test = x;

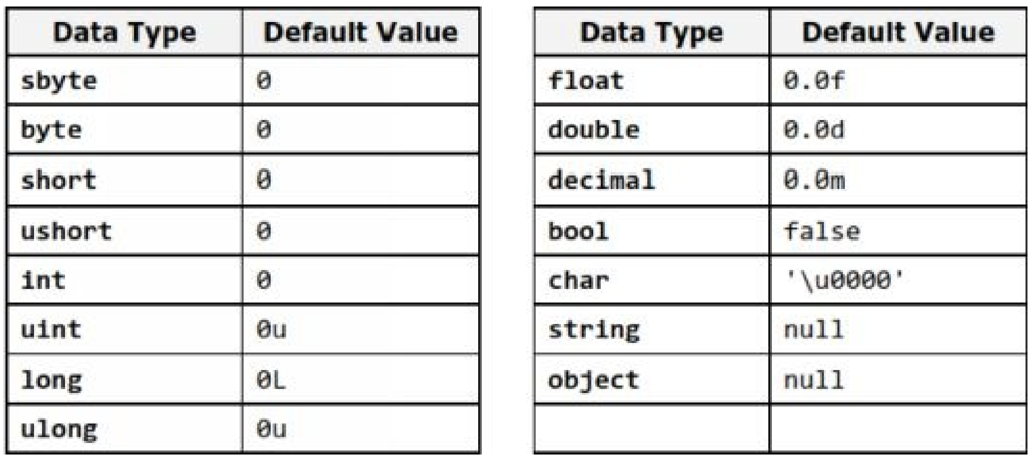
yourname = "John Doe";

How to Initialize a Variable

In programming, "initialization" is the process of assigning a value to a new variable. Thus, you'll set this "initial value" while declaring a variable.

The Default Value of a Variable

If you won't assign any value for your variable, C# will perform an automated initialization (i.e. it will assign a default value to the empty variable). The following tables will show you the default value of each data type:



Data Type	Default Value
sbyte	0
byte	0
short	0
ushort	0
int	0
uint	0u
long	0L
ulong	0u

Data Type	Default Value
float	0.0f
double	0.0d
decimal	0.0m
bool	false
char	'\u0000'
string	null
object	null

The code given below will show you how to declare variables and assign values.

// This is an example

byte years = 10;

```
short days = 31;

decimal pi = 3.14;

bool isittasty = true;

char sample = x;

string animal = “dog”;
```

Important Note: Lines that begin with two forward slashes (i.e. //) are “comments”. In programming, comments are descriptive text added to the code to improve its readability. Language compilers ignore these lines. Thus, comments won’t affect the functionality of your computer programs.

Values and References

C# programmers divide data types into two categories: value types and reference types.

A value type exists inside the execution stack of a program. Additionally, it holds the value stored by the programmer. The Value Type category consists of the Booleans, Characters, and Integers. The memory assigned to these data types becomes available once the program finishes its code execution. For instance, the variables within the Main() method of your program will be emptied once the program is closed.

A reference type, on the other hand, stores an address (i.e. a reference) in the program’s stack. This address points to the location of the needed value. Basically, a reference type assigns dynamic memory during its declaration. It also frees up some memory once the program no longer needs it.

The Literals

Literals are the values specified in a program’s source code. The example given below will illustrate this idea:

```
bool isitdelicious = false;

short income = 22000;

char firstletter = ‘c’;

byte ten = 10;

int prize = 250000;
```

In this example, the literals are false, 22000, ‘c’, 10 and 250000. These are values that you’ll add directly to the program’s code.

The Different Kinds of Literals

C# supports the following kinds of literals:

- Real
- Integer
- String

- Character
- Boolean

Let’s discuss each of these literals in detail:

Real

A real literal has a sign (i.e. + or -), a suffix, a prefix, and a decimal point. You should use this kind of literal for double, float, and decimal values. C# allows you to write real literals in their exponent form. Here are the suffixes that you can apply on real literals:

- F – This suffix represents float values.
- D – Use this suffix to indicate double values.
- M – This letter indicates decimal values.
- E – Use this letter for exponential values. For instance, “e4” means you need to multiply the integer by 10^4.

Integer

An integer literal consists of a prefix, a suffix, a sign (i.e. + or -), and a set of digits. You can use prefixes to add numbers in your code as decimals or hexadecimals. Here are the prefixes and suffixes that you can use in C#:

- 0X (e.g. 0XA6F2) – Use this prefix to indicate a hexadecimal value.
- U (e.g. 119U) – This suffix represents unsigned int (uint) or unsigned long (ulong) values.
- L (e.g. 360L) – Use this suffix for “long” values.

Important Note: C# will tag integer literals as “int” if you won’t use any suffix. Also, the letters (i.e. L, U and X) are not case-sensitive.

String

Use this literal for string values. Basically, a string literal is a set of characters placed within a pair of double quotes (e.g. “sample”).

Important Note: In the C# language, you may use the “@” symbol to create quoted (or verbatim) strings. Programmers use these strings for naming file paths.

Character

A character literal is a single character enclosed by a pair of single quotes (e.g. ‘z’) Use character literals to set values that belong to the “char” type. Additionally, you may write a character literal as:

- a single character (e.g. ‘X’)
- a code for a special character (e.g. ‘\u0067’)
- an escape code (see below)

Escape Codes – These are special character sequences that add characters to your source code. In general, the characters involved here cannot be added straight to the program's codes.

C# prevents you from adding some characters straight to your programs. Some of these characters are: tabs, newlines, backslashes, quotation marks, etc. The following list will show you the most popular escape codes in C#:

- \ – Newline Character
- \t – Tab Character
- \' – Single Quotation Mark
- \" – Double Quotation Mark

Boolean

There are only two types of Boolean literals: true and false. You can only assign one of these values when working with a "bool" (i.e. Boolean) variable. The code snippet given below will show you how to declare a Boolean literal:

```
bool answer = false;
```

Chapter 3: The Operators of the C# Language

This chapter will focus on the operators of C#. By reading this material, you’ll know how these operators work and how you can use them in your source codes.

Operators – The Basics

Operators help you in processing objects and data types. They accept inputs and operands to produce a result. The C# language uses special characters (e.g. “.”, “+”, “^”, etc.) as operators. Also, C# codes can take up to three different operands.

The Different Types of Operators

C# programmers divide operators into the following types:

Assignment Operator

In C#, you need to use the equals sign (i.e. =) to assign values. Here’s the syntax that you need to use when assigning a value:

```
name_of_operand = expression, literal, or second_operand
```

Here are some examples:

```
int y = 99;
```

```
string hello = “Good morning.”;
```

```
char sample = ‘z’;
```

Important Note: C# allows you to cascade (i.e. use several times within a single expression) the assignment operator. That means you won’t have to enter multiple equals signs if you are creating variables of the same data type. The examples given below will illustrate this rule:

```
int d = 1, e = 2, f = 3;
```

```
string hi = “Good Morning.”, bye = “Farewell.”, thanks = “Thank you.”;
```

Logical Operators

A logical (also known as “Boolean”) operator accepts and returns Boolean values (i.e. true or false). Here are the four logical operators of C#:

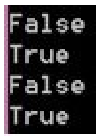
- “&&” – Programmers refer to this operator as “Logical AND”. It will give you “true” if both of the operands are true.
- “^” – This is known as the “Exclusive OR” operator. It will give you true if one of the operands is true.
- “||” – This is the “Logical OR” operator. You will get true if at least one of the operators is true.
- “!” – This operator, known as the “Logical Negation” operator, reverses the value of an operand.

The following code will show you how to use these operators:

```
class Test
```

```
{
    static void Main()
    {
    bool x = true, y = false;

    System.Console.WriteLine(x && y);
    System.Console.WriteLine(x || y);
    System.Console.WriteLine(!x);
    System.Console.WriteLine(x ^ y);

// This is an example.

    }
}
```

Once you have compiled and run this code, you will see the following results:



```
False
True
False
True
```

Arithmetic Operators

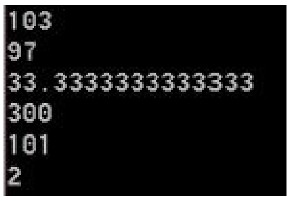
These operators help you in performing math operations. This category consists of the following operators:

- "+" – Use this operator to perform addition using two values (e.g. *int x = 1 + 1*)
- "-" – This operator allows you to perform subtraction in your codes. It deducts the value of the right operand from that of the left operand (e.g. *int x = 2 – 1*)
- "*" – With this operator, you can multiply the values of two operands (e.g. *int r = 6 * 6*)
- "/" – Use this operator to divide number values in your C# codes. Basically, "/" divides the value of the left-hand operand by that of the right-hand operand.
- "++" – This is known as the "increment operator". It increases the value of an operand by one. Unlike the ones discussed above, this operator works on a single operand. Also, you may write it before or after the operand. (e.g. 99++, 1++, ++5, etc.)
- "—" – This operator is the exact opposite of the increment operator. You can use it to decrease an operand's value by one. You may write it before or after the operand you want to modify.

The code given below will show you how these operators work:

```
class Test
{
    static void Main()
    {
    double d = 100, e = 3;

    System.Console.WriteLine(d + e);
    System.Console.WriteLine(d - e);
    System.Console.WriteLine(d / e);
    System.Console.WriteLine(d * e);
    System.Console.WriteLine(++d);
    System.Console.WriteLine(—e);

// This is an example.

    }
}
```

Once you compile and execute that code, you'll see this on your command prompt:



```
103
97
33.3333333333333
300
101
2
```

Binary Operators

These operators work on binary numbers. In the IT world, all of the information is expressed as a sequence of zeros and ones. That mean you need to master binary operators if you want to be a successful programmer.

A binary operator works exactly like a logical one. Actually, you may think that binary and logical operators use different inputs but conduct the same processes. A logical operator uses Boolean values to produce results. A binary operator, on the other hand, works on numbers presented as binary digits. Here are the binary operators you'll encounter while using C#:

- “&” – This is called “Binary AND”. It indicates the position/s where both of the operands have “1”.
- “^” – Programmers refer to this operator as “Binary Exclusive OR”. It will place “1” in each position where the values are different.
- “|” – This is the “Binary OR” operator. It indicates the position/s where any of the operands has “1”.
- “~” – This acts as the negation operator for binary values. Just like “!”, it reverses the current value of a variable.
- “<<” – This is the “Binary Left Shift” operator. It moves the bits of a binary number to the left. This movement depends on the value that you’ll assign (e.g. 4 << 1, 4 << 3, 4 << 2, etc.).

Comparison Operators

These operators allow you to compare the values inside two operands. Each of these operators gives a Boolean value as the final output. Here are the comparison operators of the C# language:

- “==” – This is the Equality operator. It checks whether the two operands are equal (e.g. *x == y*)
- “>” – This operator checks whether the left-hand operand’s value is greater than that of the right-hand operand (e.g. *x > y*).
- “<” – This operator compares the values of the two operands. It will give you “true” if the left-hand operand’s value is less than that of the right-hand operand (e.g. *y < x*).
- “>=” – With this operator, you can determine whether the value of the left-hand operand is greater than or equal to that of the right-hand operand.
- “<=” – This operator will give you true if the value of the left-hand operand is less than or equal to that of the right-hand operand.
- “!=” – Use this operator to check the equality of two operands. It will give you true if the values are not equal (e.g. *x != y*).

Conditional Operator

C# users refer to “?” as the conditional operator. Basically, this operator uses the result of a Boolean expression to determine which statement should be processed. This is called “ternary operator” because it works on three operands. Here, the first value should be Boolean, while the remaining values need to belong to the same data type (e.g. characters, strings, numbers, etc.).

The syntax of this operator is:

first_operand ? second_operand : third_operand;

Here’s how it works: If “first_operand” is true, the program will work on “second_operand”. If “first_operand” is false, however, the program will work on “third_operand”.

Chapter 4: Data Type Conversion

In general, an operator works on arguments that belong to the same type. However, the C# language offers a wide range of data types that you can use for certain situations. To conduct any process on variables that belong to different types, you should convert one of those variables first. In C#, data type conversion can be implicit or explicit.

Each C# expression has a data type. This data type results from the literals, variables, values, and structures used inside the expression. Basically, you might use an expression whose type is incompatible for the situation. When this happens, you’ll get one of these results:

- The program will give you a compile-time error.
- The program will perform an automated conversion. That means the program will get the right expression type.

Converting an “s” type to “t” type allows you to treat “s” as “t” while running your program. Sometimes, this process requires the programmer to validate the type conversion. Check the examples below:

- Converting “objects” to “strings” will need verification during the program’s runtime. This verification ensures that you really want to use the values as strings.
- Converting “string” values to “object” values doesn’t need any validation. That’s because the “string” type is a branch of the “object” type. You can convert strings to their “parent” class without losing data or getting any error.
- You won’t have to perform verification while converting int values to long values.

The “long” data type covers all of the possible values of the “int” type. That means the data can be transformed without any error.

- Converting “double” values to “long” values involves validation. You might experience loss of data, depending on the values you’re working on.

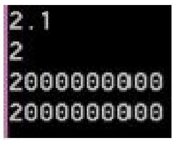
Important Note: C# has certain restrictions when it comes to changing data types. Here are the possible type conversions supported by this language:

Explicit Conversion

Use this conversion if there’s a chance of information loss. For instance, you’ll experience information loss while converting floating-point values to integer values (i.e. the fractional section will disappear). You might also lose some data while converting a wide-range type to a narrow-range type (e.g. long-to-int conversion, double-to-float conversion, etc.). To complete this process, you need to use the “type” operator. Check the examples below:

```
class Example
{
    static void Main()
    {
            double yourDouble = 2.1d;
            System.Console.WriteLine(yourDouble);

            long yourLong = (long)yourDouble;
            System.Console.WriteLine(yourLong);

            yourDouble = 2e9d;
            System.Console.WriteLine(yourDouble);

            int yourInt = (int)yourDouble;
            System.Console.WriteLine(yourInt);

    }
}
```

If compiled and executed correctly, that program will give you these results:


```
2.1
2
2000000000
2000000000
```

Implicit Conversion

You can only perform this conversion if data loss is impossible. This conversion is referred to as implicit because it doesn't require any operator. The C# compiler will perform implicit conversion whenever you assign narrow-range values to variables that have a wide-range.

String Conversion

C# allows you to convert any data type to "string". The compiler will perform this conversion process automatically if you will use "+" and at least one non-string argument. Here, the non-string argument will become a string and the "+" operator will produce a new value.

Chapter 5: The Console

This chapter will teach you how to use the console in providing and collecting data. You'll learn how to use this tool in performing I/O (i.e. Input/Output) operations in the C# language.

Console – The Basics

A console is a tool that you can use to interact with the programs available in your computer. Because of this, consoles are considered as an important part of any operating system.

In C#, user-to-program interactions involve standard output (e.g. screen) and/or standard input (e.g. keyboard strokes). When combined, these things are called I/O operations. The text within the console provides useful data and is a set of characters from a computer program.

The computer's OS (i.e. operating system) links I/O devices to every console program installed on the computer. Most systems tag screens and keyboards as default output and input devices. However, you can easily redirect these channels to certain files or devices.

The Interaction between a Program and Its User

Most computer programs interact with their user/s. Users need to provide inputs or instructions to the program they are interacting with. There are a lot of interaction methods available these days: console-based, browser-based, internet-based, graphical, etc. Now, the people who use consoles for program interactions are continuously decreasing. That's because the alternative options (e.g. browser-based) are better in terms of convenience and user-friendliness.

Using the Console

Sometimes, the console serves as the most important tool for user-program interactions. For instance, you need to use the console when creating simple computer programs. These programs require you to focus on the problems that must be fixed. Beautiful GUIs (i.e. graphical user interface) and attractive outputs won't help you with this task.

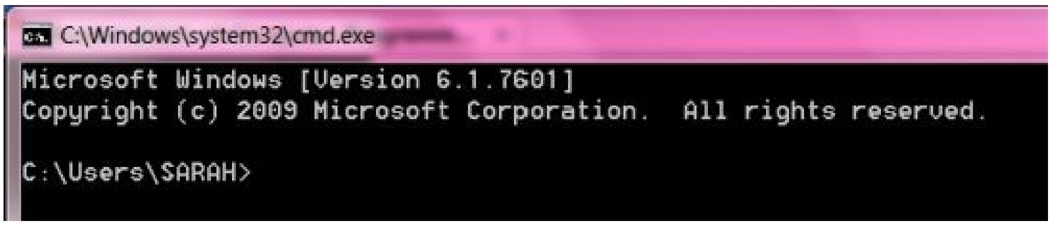
You can also use the console to try a small code block before adding it to a complex program. Since the console is a simple tool, you can easily isolate the code block without pressing any button or seeing different screens. The console will give you the results that you need in just a few seconds.

Accessing the Console

The process of accessing the console depends on the OS you're currently using. For instance, here are the steps you need to take if you are using a Windows computer:

1. Click on the Start button.
2. Click "Programs" and choose "Accessories".
3. Click on the option that says "Command Prompt".

The console will display your home directory, which is represented by your username. In the example below, the username is SARAH:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\SARAH>
```

Important Note: If you're using Windows Vista, Windows 7, or Windows 8, you can launch the console by running a search for "cmd".

Additional Information About the Console

The console is a black window that displays text-based information. It has a blinking cursor and shows character strings. As you can see, the cursor moves each time you type a character.

A Windows program can be desktop-based, browser-based, internet-based, or console-based. A console-based program utilizes the console for input and output processes. Desktop-based programs, on the other hand, rely on their GUI for user-program interactions. This book focuses on console-based applications so you'll be entering data using your keyboard and you will see the results within the command prompt.

Certain console-based applications require the users to provide text, numbers, characters, etc. That means you'll be using your keyboard often while working with a console-based program.

The Basic Commands

In this section, you'll learn about the basic commands that you issue using the command prompt. You should study this material carefully if you want to be a great C# programmer.

The Console Commands

- cd name_of_directory – This command allows you to change the active directory. For instance, if you want to go to the Users folder inside C:, you need to issue the following command:

  cd C:\Users

- dir – Use this command to view the files inside the active directory.
- rmdir name_of_directory – This command allows you to delete a directory. For example, *rmdir Sample* deletes the directory named "Sample".
- mkdir name_of_directory – With this command, you can create a new directory inside the active directory. For instance, if you're currently in the Users directory, *mkdir John* will create a folder named "John" inside "Users".
- type name_of_file – Use this command to view the content of an existing file. For instance, you may issue "*type HelloCSharp.cs*" to see the contents of that .cs file.

Standard I/O

The Standard I/O system is a mechanism developed for UNIX computers. This system involves special devices to accept input and produce output.

If a program is waiting for the user to perform an action or provide information, the console will show a cursor. This means the system is in "waiting mode".

Input and Output

Operating systems need to define the standard I/O mechanisms for user-program interactions. When launching a console-based application, the code at the program's initialization block needs to work with I/O mechanisms assigned by the OS. This code allows the user to interact with the console-based program involved.

Because of this code, the program can read the user's input from "Console.In" (i.e. C#'s standard stream for inputs), send information to "Console.Out" (i.e. C#'s standard stream for outputs), and report problematic situations through "Console.Error" (i.e. C#'s standard stream for error messages).

Important Note: You'll learn about C# streams later in this book. This chapter will concentrate on the concepts and ideas related to C#'s I/O mechanisms.

The I/O Devices

Some programs accept non-keyboard inputs. For example, you can create a C# program that takes inputs from text files, barcode readers, microphones, and other data sources. A program can also send its output to the screen, to an external file, to a printer, etc.

System.Console

The class named System.Console has various methods and attributes that you can use to read, display, or format text-based information. Three of the most popular attributes of the System.Console class are related to entering and showing data. These attributes are: Console.In, Console.Out, and Console.Error.

These attributes help you in displaying data, reading information, and reporting error messages. C# allows you to use these attributes directly. However, you need to keep in mind that System.Console contains other methods and attributes that can help you with I/O operations.

Important Note: The C# language allows you to modify the default streams of your programs. To do that, you need to use the following methods: Console.SetOut (for the Output stream), Console.SetIn (for the Input Stream), and Console.SetError (for the Error stream).

The Console Input

Programming languages have a system for entering information to the computer's console. The C# language, in particular, uses an object called "Console.In" to control its input stream.

Actually, you don't have to use Console.In directly. The "Console" class offers two methods (i.e. Console.ReadLine and Console.Read) that work with Console.In. You can use these methods to read data from your command prompt.

The Console.ReadLine Method

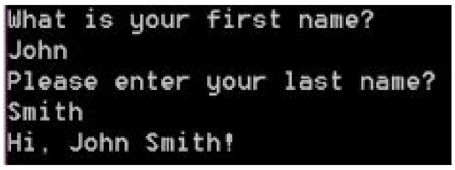
This method offers simplicity and convenience for people who want to read data from their console. When you call this method, your program will stop and wait for the user's input. Then, the user may enter a character or a string and hit "Enter". By pressing the Enter key, the user informs the program that they have entered the needed input. The Console.ReadLine method will read the user's input, hence the name.

The code given below will show you how Console.ReadLine works:

```
class Test
{
    static void Main()
    {
            System.Console.WriteLine("What is your first name? ");
            string yourfirstName = System.Console.ReadLine();

            System.Console.WriteLine("Please enter your last name? ");
            string yourlastName = System.Console.ReadLine();
```

```
		System.Console.WriteLine(“Hi, {0} {1}!”, yourfirstName, yourlastName);
}

}
```

If you’ll enter “John” and “Smith” into the command prompt, you’ll get the following message:



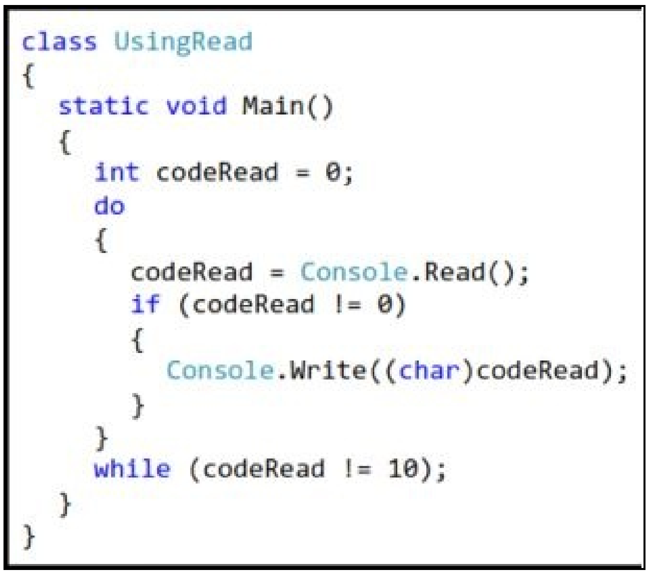
```
What is your first name?
John
Please enter your last name?
Smith
Hi, John Smith!
```

__The Console.Read Method__

There are significant differences between Console.Read and Console.ReadLine. First, Console.Read reads a single character only while Console.ReadLine reads the entire expression. Second, Console.Read returns a code rather than the accepted input. If you want to utilize the output as an ordinary character, you should do one of these things:

- Convert the output into a character
- Use the Convert.ToChar method on the result

Important Note: Constant.Read will only read a character if the user hits “Enter”. When the user presses this key, the program will transfer the input to the standard string’s buffer and invoke the Console.Read method. Check the example given below:



```
class UsingRead
{
   static void Main()
   {
      int codeRead = 0;
      do
      {
         codeRead = Console.Read();
         if (codeRead != 0)
         {
            Console.Write((char)codeRead);
         }
      }
      while (codeRead != 10);
   }
}
```

How to Read a Number

C# programs don't read numbers directly. They need to take the inputs as strings (i.e. through the Console.ReadLine method) and convert them into numbers. The process of converting strings into other data types is known as parsing. All of the primitive data types in the C# language support this process.

The Console Output

This part of the book will focus on two methods: Console.WriteLine and Console.Write. Both of these methods can print all of the data types in C# language. That means you can use them to display messages on the user's console.

The following code snippets will teach you how these methods work:

Console.WriteLine("I want to learn the C# programming language.");

// This line will print the string inside the parentheses.

Console.WriteLine(100);

// This line will show "100" on the user's command prompt.

Console.WriteLine(3.14);

// This line will print the double value 3.14 on the user's console.

You can use the Console.WriteLine method to show different kinds of data on the user's screen. That's because the Console class has a built-in version of each data type.

The only difference between Console.WriteLine and Console.Write is that the former prints the value within the parentheses and adds a newline character on the console whereas the latter simply shows the information inside the parentheses. Thus, using Console.WriteLine in your codes is like hitting the "Enter" key after showing the message (i.e. The cursor will go to the next line.).

String Concatenation

The C# language prevents programmers from using operators on strings values. Obviously, you can't perform arithmetic operations on actual words (e.g. fish – dog, power * humans, land / people, etc.). However, C# allows you to concatenate (i.e. combine) multiple string objects to generate a new one. You can do this by using the "+" symbol on your strings. The example given below will illustrate this idea.

```
string gadget = "smartphone";
string example = "I use my " + gadget + "to access " + "the internet.";
Console.WriteLine(example);
```

If you'll execute that code, the console will display this:

```
I use my smartphone to access the internet.
```

Chapter 6: The Conditional Statements of C#

This chapter will teach you how to use conditional statements in your C# codes. Basically, a conditional statement is a code block that allows you to perform various actions based on the assigned condition/s. Read this material carefully as it contains valuable information that can help you master C# in just 7 days.

The Operators

In this section, you'll learn more information about the comparison operators of C#. These operators are extremely important: they can help you in setting the conditions for your own conditional statements.

The Comparison Operators

C# has different operators that you can use to compare values of the same type. These operators are:

- <
- >
- <=
- >=
- !=
- ==

You may use the operators given above to compare C# expressions (e.g. two integers, two variables, two constants, etc.). While using any of these operators, the result will always be a Boolean. Analyze the following examples:

int height = 170;

Console.WriteLine(height < 180); // This will give you “true”.

string   gender = “Male”;

Console.WriteLine(gender == “Female”); // This will give you “false”.

double valueofPi = 3.14;

Console.WriteLine(valueofPi > 100); // This will give you “true”.

The Logical Operators

C# also supports some logical operators. Programmers use these operators to create Boolean (also called “logical”) statements. Let’s discuss each logical operator in detail:

The AND and OR Operators

You can only use the logical AND (i.e. &&) and logical OR (i.e. ||) operators on expressions that involve Boolean values. The logical AND operator will give you “true” if both of the operands are true. For example:

bool sample = (1 < 2) && (10 == 10);

Since 1 is less than 2 and 10 is equal to 10, the expression will give you true. Since the logical AND operator provides quick results without performing unnecessary analyses, programmers refer to it as a “short-circuit operator”. It checks the left section of the statement first. If the evaluation is “false”, the operation will immediately stop. The expression can no longer be true since one of the operands is false.

The logical OR (i.e. ||) operator, on the other hand, will give you true if at least one of your operands evaluates to true. For instance:

bool sample = (1 < 2) || (10 > 100);

This example will give you true since the left-hand operand is true. The logical OR operator is also a “short-circuit operator”, since it can give a result even without checking the entire C# statement. In the current example, logical OR will no longer analyze the second operand since the first one is already true. Thus, the condition has already been met.

The Exclusive OR and Negation Operators

The exclusive OR (i.e. ^) is a full-circuit operator since it needs to check both operands before giving the result. It will give you true ONLY IF one of the operands evaluates to true. Thus, you’ll get false if both of the operands are true. Check the following example:

bool sample = (1 < 2) ^ (2 == 2);

Since the two expressions are true, exclusive OR will give you false.

The negation operator (i.e. !), however, simply reverses the Boolean value to which it is attached. For instance:

bool sample = !(1 < 2);

One is less than two. However, since “!” is attached to the Boolean expression, the result will be reversed. That means this expression evaluates to “false”.

The Statements

Now that you know how to use the comparison and logical operators, it’s time to discuss the creation of conditional statements. These statements are powerful tools that can help you achieve programming logic.

The If Statement

The syntax of an if statement is:

```
if (the conditional expression)
{
    The if statement’s body;
}
```

As you can see, this syntax has three different parts:

1. The “if” clause – This part indicates the start of the “if” statement.

2. The Boolean expression – This part can hold a logical expression or a variable that holds Boolean data. Unlike C, C# doesn’t accept integer expressions for its conditional statements.

3. The body of the statement – You need to enclose this part using a pair of curly braces. C# allows you to place multiple statements inside this section.

The program will run the statement’s body if the Boolean expression evaluates to true. If the evaluation is false, the program will ignore the statement’s body and pass the control to the succeeding code blocks. The example given below will show you how “if” statements work:

```
Console.WriteLine(“Please enter a number.”);
int number1 = int.Parse(Console.ReadLine());
int number2 = 0;
if (number1 > number2)

{

    Console.WriteLine("C# is an excellent programming language.");
}
```

This code requires the user to provide a number. If the user's input is higher than zero, the program will print "C# is an excellent programming language." in the command prompt.

Important Note: You may omit the curly braces if the body of your conditional statement is just a single line. Check the example below:

```
int x = 10;
if (x > 0)
    System.Console.WriteLine("This variable holds a positive number.");
```

The If-Else Statement

Just like other programming languages, C# supports "if-else" conditional statements. When writing an if-else statement, you should use the following syntax:

```
if (the Boolean variable or expression)
{
    The conditional statement's body;
}
else
{
    The else statement's body;
}
```

The syntax given above consists of the following parts:

- The “if” clause – This part begins the conditional statement.

- The Boolean expression or variable – The value of this expression determines the codes that the program will use.

- The conditional statement’s body – This can be a single statement or a group of statements. If you are using multiple statements, enclose them inside a pair of curly braces.

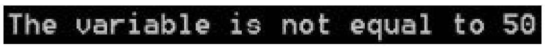
- The “else” keyword – This is a C# keyword that triggers the “else” clause.

- The else clause’s body – This is identical to the conditional statement’s body (see above).

When working on an if-else statement, the computer program analyzes the value within the parentheses. The value needs to be Boolean (i.e. either true or false). If the value is “true”, the program will execute the conditional statement’s body and ignore the “else” clause. If the value is false, however, the program will run the else statement’s body. Analyze the following example:

```
int a = 100;
if (a != 50)
    {
    Console.WriteLine(“The variable is not equal to 50”);
    }
else
    {
    Console.WriteLine(“The variable is equal to 50”);
    }
}
```

You can interpret this code as: if a != 50, the console will print: “The variable is not equal to 50”. On the other hand, if a = 50, the console will show: “The variable is equal to 50”. Since the variable named a holds 100, your command prompt will give you this:



```
The variable is not equal to 50
```

Nested Statements

In some cases, you need to write an "if" statement inside another "if" statement. Programmers refer to this process as "nesting". That means you are "nesting" a conditional statement if you are writing it inside another conditional statement.

Important Note: C# allows you to nest if-else statements. In this situation, each "else" clause points to the "if" clause that comes before it. This simple rule can help you analyze nested conditional statements, which can be extremely complex.

According to expert programmers, you should not create more than three levels of nested statements. If you really need to use four or more conditional structures, export them to a different method (You'll learn about methods later.).

The code given below will show you how nested "if" statements work:

```
int a = 10;
int b = 20;

if (a > b)
{
    Console.WriteLine("a is greater than b");
}
else
{
    if (a < b)
    {
            Console.WriteLine("a is less than b");
    }
    else
    {
            Console.WriteLine(“a is equal to b”);
    }
}
```

The Switch-Case Statement

Programmers use a switch-case statement in situations where the possible answers are numerous. Here’s the syntax of a switch-case statement:

```
switch (selector)
{
    case first_integer:
            statements;
            break;
    case second_integer:
            statements;
            break;

            default:
                        statements;
                        break;
}
```

Basically, selectors are expressions that return usable values (i.e. values that you can compare or process). The “switch” keyword compares the selector’s value against the “cases”. If the value of the selector matches that of a case, the statements assigned to that case will run. If there’s no match, however, the program will execute the default statement/s (if any). That means the program will analyze the selector first before running any statement.

Important Note: Make sure that the cases have unique values. If two or more cases share the same value, your program will produce undesirable results.

As you can see, each case has a “break” clause. The “break” clause terminates the body of the case to which it belongs. C# requires you to add this clause to all of your switch statements.

You don’t have to place the “default” part at the end of your switch-case statement. However, this position is recommended if you want to keep your codes readable.

Chapter 7: The Loop Statements

This chapter will teach you how to use loop statements. As a programmer, you’ll use a loop statement to run certain code blocks repeatedly. Loops play an important part in C# programming. Hence, you should study this material carefully as well if you want to master this language in 7 days.

Loops – The Basics

While creating programs, you often need to execute code sequences multiple times. Typing the same blocks of code several times can be extremely boring. Thus, you need to find a quick and simple way to repeat codes. Fortunately, the C# language supports loop statements.

A loop is a tool that runs code fragments repeatedly. You can use it to run codes for a certain number of times or as long as a given condition is fulfilled. C# offers different types of loop statements. Let’s analyze each type in detail:

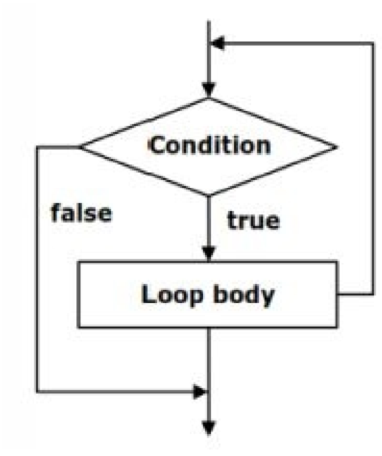
The While Loop

This is the simplest loop statement in C#. Its syntax is:

```
while (the_assigned_condition)
{
    the_loop's_body;
}
```

In this syntax, “the_assigned_condition” is an expression that produces a Boolean value. This condition dictates how many times the “body” will run. The “body”, on the other hand, is the statement (or group of statements) that you want to execute.

The diagram given below will illustrate how while loops work:



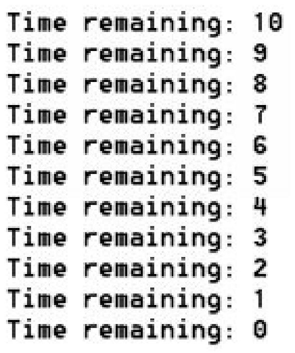
When working on a “while” loop, a C# program checks the value of the Boolean expression. If the value is “true”, the program will run all of the statements within the loop’s body. Then, the program will check the Boolean expression to see its value. If the value is still true, the program will rerun the loop’s body and go back to the first step (i.e. check the expression’s value). This process will go on until the Boolean expression evaluates to false. Once this happens, the program will process the code blocks right after the loop.

Important Note: Your C# program won’t run the while loop’s body if the Boolean expression is false. Thus, if the B00lean value is false when you launched the program, your in-loop statements will never be executed.

The following example will illustrate how while loops work:

```
int timer = 10;

while (timer >= 0)

{
System.Console.WriteLine(“Time remaining: ” + timer);

timer—;
}

```

If you will compile and execute this code, your command prompt will print this data:



```
Time remaining: 10
Time remaining: 9
Time remaining: 8
Time remaining: 7
Time remaining: 6
Time remaining: 5
Time remaining: 4
Time remaining: 3
Time remaining: 2
Time remaining: 1
Time remaining: 0
```

The Break Operator

You can use “break” (a C# operator) to exit a loop prematurely. Programmers use this operator if they don’t want to wait for the loop’s natural termination. Basically, a loop will end once it encounters a break operator. This situation forces the program to jump to the code fragments right after the loop.

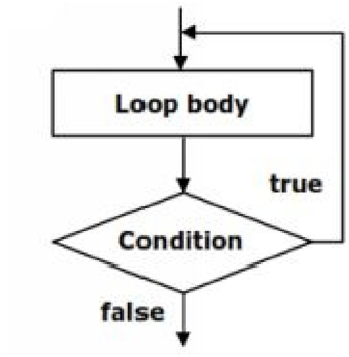
In C#, you can only use this operator inside the loop’s body. That means “break” will only work while the loop is running.

The Do-While Loop

A do-while loop is almost identical to a while loop. The only difference between these loops is that the former analyzes the Boolean value after running the loop's body. Thus, you can rest assured that your codes will run at least once even if the Boolean expression is false. Here is the syntax of the do-while loop:

```
do
{
    statements;
} while (Boolean expression)
```

Do-while loops work according to this pattern:



The program will execute all of the statements within the loop's body. Then, it will check the condition (i.e. the Boolean expression). If the condition evaluates to true, the program will execute the loop's body again and perform the "value inspection". This process will go on until the condition becomes false. Thus, your codes might run forever if your assigned condition always evaluates to true.

The For Loop

In general, "for" loops are more complex than do-while and while loops. However, for loops can help you perform difficult tasks using fewer C# codes. The syntax of a "for" loop is:

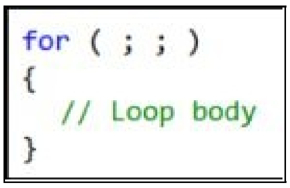
```
for (initializer; Boolean_expression; update)

{
    the_loop's_body;
}
```

A “for” loop has an initializer (i.e. the initial value of the counter), a Boolean expression, a C# statement that updates the counter, and the loop’s body.

The “counter” (i.e. the initial value assigned to the loop) serves as a “for” loop’s most distinctive feature. In most cases, the counter’s value increases until it reaches the final value (e.g. 1 to 10). Also, programmers usually know how many times the “for” loop will iterate.

You may include one or more variables in your “for” loops. These variables may move in descending or ascending order. When writing programs, you may combine ascending and descending variables in a single “for” loop. In addition, an ascending variable can go from 1 to 1024 because “for” loops support arithmetic operations (e.g. addition, multiplication, etc.).

Important Note: All of the parts of a “for” loop are optional. You can create an infinite “for” loop by leaving blank spaces on the syntax. Here’s an example:



```
for ( ; ; )
{
   // Loop body
}
```

At this point, you’re ready to learn more about the different sections of the “for” loop.

The Initialization

A “for” loop can possess an initializing fragment:

```
for (double sample = 1; …;   …)

{

   // You can use the “sample” variable here.
}
   // You can’t use the variable here.
```

C# programs execute this fragment once, right before running the “for” loop. Often, programmers use an initializing fragment to create a counter (also known as “loop variable”) and assign its initial value. This counter is available and usable when inside the loop’s body. C# allows you to declare multiple variables using a single initializing fragment.

The Condition

Obviously, a “for” loop needs a conditional expression. Here’s a sample:

```
for (double sample = 1; sample < 5; …)

{
   // This is the loop’s body.
}
```

Computer programs evaluate the conditional expression before running the loop’s body. If the result is “true”, the body of the loop will run; otherwise, the involved program will jump to the statements written after the current loop.

The Update Statement

This is the final part of a “for” loop. Basically, an update statement updates the loop’s counter. Let’s use the code snippet given above:

```
for (double sample = 10; sample > 1; sample—)
{
   // This is the loop’s body.
}
```

The program executes this part after running the loop’s body. Keep in mind that this statement updates the counter’s value.

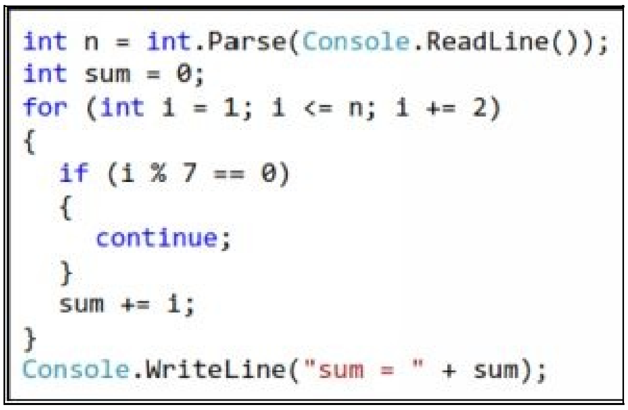
The Loop’s Body

This part consists of C# statements. It can utilize the variables you declared in the loop’s initializing fragment. Check the example below:

```
for (double sample = 10; sample > 1; sample—)
{
    System.Console.WriteLine(sample);
}
```

The Continue Operator

In some cases, you need to stop the active iteration without ending the loop itself. You can accomplish this task using the C# operator called “continue”. The example given below will illustrate how this operator works:



```
int n = int.Parse(Console.ReadLine());
int sum = 0;
for (int i = 1; i <= n; i += 2)
{
   if (i % 7 == 0)
   {
      continue;
   }
   sum += i;
}
Console.WriteLine("sum = " + sum);
```

This code computes the total of all the odd numbers within the range (1 to n), which produce remainders when divided by 7.

The “Foreach” Loop

Just recently, C# introduced the concept of “foreach” loops (i.e. extended “for” loops). This loop concept is also available in other programming languages such as C, VB, C++, PHP, etc. With this programming tool, you can run all of the elements of a list, array, or other groups of values. A “foreach” loop takes all of the existing elements even in non-indexed data groups.

When writing this kind of loop, use the following syntax:

```
foreach (type name_of_variable in name_of_group)
{
    the_statements;
}
```

As you can see, this loop is much simpler than a typical “for” loop. A “foreach” loop can help you scan all of the objects inside a collection quickly. It’s no surprise that countless programmers use this loop in writing their codes.

The Nested For Loop

C# allows you to place a “for” loop inside another “for” loop. The loop at the innermost part of the code runs the most number of times. The one at the outermost section, on the other hand, gets the least number of repetitions. You should use the following syntax when “nesting” for loops:

```
for (initializing_fragment, condition, update_statement)
{
    for (initializing_fragment, condition, update_statement)
              {
              statements
              }
}
```

Here, the program runs the initial “for” loop, executes its body, and triggers the nested loop. The program will check the second loop’s condition and execute the codes inside it until the evaluation becomes false. Then, the program will make the necessary adjustments on the loops’ counters. The program will repeat the entire process until all of the conditions evaluate to false.

Chapter 8: The Methods of the C# Language

This chapter will focus on the C# methods. Here, you'll learn how to declare and utilize methods in your own programs. You need to memorize the lessons contained in this chapter if you want to be an effective C# programmer in just 7 days.

Methods – The Basics

For most programmers, methods are core aspects of any program. Methods can solve problems, accept user inputs (known as parameters), and produce results.

Methods complete their tasks by representing the data conversions done by the program. Additionally, methods are the areas where the actual processes are completed. This is the main reason why C# programmers consider methods as the basic units of any computer program.

The Benefits Offered by Methods

In this section, you’ll discover the major benefits offered by C# methods. After reading this material, you’ll know why you should use these tools in writing your programs.

Better Structure and Code Readability

Programming experts claim that you should use methods while writing computer programs. Methods, even the simplest ones, can improve the structure and readability of your C# codes.

Here’s an important fact that you need to know: programmers spend 20% of their time on writing and checking their computer program. The remaining 80% is spent on maintaining and improving the software. Obviously, you’ll have an easy time working on your program if your codes are readable and well-structured.

Prevention of Redundant Codes

Methods can help you avoid redundant codes in your programs. Redundant or duplicated codes often produce undesirable results in computer applications.

Better Code Repetition

This benefit is closely related to the previous one. If your program needs to use a certain code fragment multiple times, it’s an excellent idea to transfer the said fragment into a C# method. You can invoke methods multiple times, which means you can repeat important codes without retyping them.

Declaring, Implementing, and Invoking C# Methods

At this point, you need to know the processes that you can perform on an existing method. These processes are:

- Declaration – In this process, you will link a method to a program. This process allows you to call the method in any part of your program.

- Implementation – This process involves entering codes to complete a certain task. The codes involved here exist inside the method you're using.

- Invocation – This is the process of calling a declared method. Here, you'll use the method to solve a problem or perform an action.

Method Declarations

In C#, you need to declare methods inside a class. Additionally, you're not allowed to "nest" methods (i.e. write a method inside the body of another method). The perfect example for this is Main(), the method you've used multiple times. The code given below will illustrate this:

```
class DeclaringMethods
{
   static void Main()
{
   System.Console.WriteLine("Hi C#!");

}
}
```

The Syntax

Use the following syntax when declaring a method:

static data_type_of_the_result name_of_method (list_of_parameters)

Let’s analyze this syntax using the Main() method (i.e. *static void Main()*). Main() uses “void” as its return type since it doesn’t generate any result. The word “Main” serves as its name. The parentheses, on the other hand, act as containers for the parameters that the programmer will provide.

Important Note: You have to follow this syntax when writing a C# program. Don’t change the placement of the method’s parts.

C# doesn’t require you to include parameters in your declarations. That means you can leave the parentheses empty (e.g. Main()).

The Method’s Name

You need to indicate the method’s name during declarations, invocations, or implementations. Here are the rules that you need to remember when naming your methods:

- Make sure that the initial letter is in uppercase.
- Begin each word with an uppercase letter (e.g. SampleMethod, NewMethod, MainLine, etc.).
- Use verbs and nouns as names of your methods.

Important Note: The rules discussed above are completely optional. Follow these rules if you want to have excellent structure and readability in your C# codes.

Method Implementations

Declaring a method isn’t enough. You also need to implement your methods if you want them to run inside your programs. Programmers use the term “body” when referring to the implementation of a method.

The Body

You’ll find the method’s body inside a pair of curly braces. This body consists of commands, expressions, and statements that you want to execute. Thus, the body is an important part of any method.

Important Note: Keep in mind that you cannot nest methods in C#. Don’t write methods inside other methods.

Method Invocations

Basically, this is the process of running the statements within the method’s body. Invoking a method is easy and simple: you just have to indicate its name, add a pair of curly braces, and terminate the line using a semicolon. Here’s the syntax that you should use:

```
name_of_method();
```

The sample code given below will show you how to invoke a method:

```
class Animals
{     static void Main()
{
    Console.WriteLine(“I love dogs.”);
}
}
```

If you’ll compile and execute that code, your command prompt will print the following message:

```
I love dogs.
```

The Places Where You Can Invoke a Method

C# allows you invoke methods in the following areas:

- Inside the program’s main method (i.e. Main()).
- Inside another method.
- Inside the method’s own body. This technique is called “recursion”.

Conclusion

Thank you again for downloading this book!

I hope this book was able to help you become a skilled C# programmer in just a few days.

The next step is to continue writing your own programs using C#.

Finally, if you enjoyed this book, then I ’ d like to ask you for a favor, would you be kind enough to leave a review for this book on Amazon? It ’ d be greatly appreciated!

Thank you and good luck!