

## Lesson 1

### Introduction to Programming

#### Learning Objectives

Students will learn about:

- Basics of Computer Programming
- Decision structures
- Repetition structures
- Exception handling

#### OBJECTIVE1: Basics of Computer Programming

##### What is a Computer PROGRAM?

A computer program is a series of organized instructions that directs a computer to perform tasks.

Without programs, computers are useless. A program is like a recipe. It contains a list of variables (called ingredients) and a list of statements (called directions) that tell the computer what to do with the variables.

##### PROGRAMMING

Programming is a creation of a set of commands or instructions which directs a computer in carrying out a task.

##### PROGRAMMING LANGUAGE

A programming language is a set of words, symbols and codes that enables humans to communicate with computers. It has Syntax and semantics rules that programmer follows to write the programs.

Some of programming languages examples are:

BASIC (Beginner's All Purpose Symbolic Instruction Code), Pascal, C, C++, Smalltalk, Java, C#, Vb.Net

##### Types of Programming Language

A low-level programming language is a programming language that provides little or no abstraction from computer's microprocessor.

A high-level programming language is a programming language that is more abstract, easier to use, human readable and more portable across platforms.

##### GENERATIONS OF PROGRAMMING LANGUAGE

### FIRST GENERATION OF PROGRAMMING LANGUAGE

The first generation of programming language, or 1GL, is machine language. Machine language is a set of instructions and data that a computer's central processing unit can execute directly. Machine language statements are written in binary code, and each statement corresponds to one machine action.

Computers use a special code of their own to express the digital information they process. It's called the binary code because it consists of only two symbols—0s and 1s.

Why 0s and 1s? Because those are the only two numbers you need to express the flow of electricity through a transistor. It's either on or it's off. On is 1, off is 0. Everything you say to a computer has to be put in terms of these two numbers.

### SECOND GENERATION PROGRAMMING LANGUAGE

The second generation programming language, or 2GL, is assembly language. Assembly language is the human-readable notation for the machine language used to control specific computer operations. An assembly language programmer writes instructions using symbolic instruction codes that are meaningful abbreviations or mnemonics. An assembler is a program that translates assembly language into machine language.

### THIRD GENERATION PROGRAMMING LANGUAGE

The third generation of programming language, 3GL, or procedural language uses a series of English-like words that are closer to human language, to write instructions.

High-level programming languages make complex programming simpler and easier to read, write and maintain. Programs written in a high-level programming language must be translated into machine language by a compiler or interpreter. PASCAL, FORTRAN, BASIC, COBOL, C and C++ are examples of third generation programming languages.

### FOURTH GENERATION PROGRAMMING LANGUAGE

The fourth generation programming language or non-procedural language, often abbreviated as 4GL, enables users to access data in a database. A very high-level programming language is often referred to as goal-oriented programming language because it is usually limited to a very specific application and it might use syntax that is never used in other programming languages. Basic, c, SQL, NOMAD and FOCUS are examples of fourth generation programming languages.

### FIFTH GENERATION PROGRAMMING LANGUAGE

The fifth generation programming language or visual programming language is also known as natural language. This generation Provides a visual or graphical interface, called a visual programming environment, for creating source codes. Fifth generation programming allows people to interact with computers without needing any specialized knowledge. People can talk to computers and the voice recognition systems can convert spoken sounds into written words. Prolog and Mercury are the best

## Lesson1: Introduction to Programming

known fifth-generation languages.

### EXT: OPEN PROGRAMMING LANGUAGE

The Open Programming Language (OPL) is an embedded programming language found in portable devices that run the Symbian Operating System. For example mobile telephones and PDAs. OPL is an interpreted language that is analogous to BASIC.

## PROGRAMING APPROACHES

### STRUCTURED PROGRAMMING

Structured programming often uses a top-down design model where developers map out the overall program structure into separate subsections from top to bottom. In the top-down design model, programs are drawn as rectangles. A top-down design means that the whole program is broken down into smaller sections that are known as modules. A program may have a module or several modules. Structured programming is beneficial for organizing and coding computer programs which employ a hierarchy of modules. This means that control is passed downwards only through the hierarchy.

Examples of structured programming languages include Ada, Pascal and Fortran.

### OBJECT-ORIENTED PROGRAMMING

The object-oriented approach refers to a special type of programming approach that combines data with functions to create objects. In an object-oriented program, the object has relationships with one another.

One of the earliest OOP languages is Smalltalk.

Java, Visual Basic and C++ are examples of popular OOP languages.

### DIFFERENCE BETWEEN STRUCTURED AND OBJECT ORIENTED PROGRAMMING

- Structured programming often uses a top-down design model.
- The object-oriented programming approach uses objects.

## TRANSLATOR

Have you ever wondered how your computer runs your favorite software? Your favorite software is a program that consists of several instructions that perform its operation. A programmer will write a source code which consists of the instructions needed to run a program but it needs to be translated to machine language code. Using binary code to program a computer is terse and extremely difficult to accomplish for any non-trivial task. Thus, to simplify programming, scientists and

## Lesson1: Introduction to Programming

computer engineers have built several levels of abstractions between computers and their human operators. These abstractions include software (such as operating systems, compilers, and various runtime systems) that takes responsibility for translating a human-readable program into a machine readable program.

All software packages or programs are written in high-level languages, for example, C#, Visual Basic and Java. The translation of high level languages to machine language is performed by a translator(compiler or interpreter with assembler )will translates the source code into machine language which is made of a sequence of bits (eg. 01100011). The computer will load the machine code and run the program.

### ASSEMBLER

An assembler is a computer program for translating assembly language — essentially, a mnemonic representation of machine language — into machine language. For example in intel 80836, the assembly language for the 'no operation' command is NOP and its machine code representation is 10010000.

### INTERPRETER

The interpreter will read each codes(Line of code) converts it to machine code and executes it line by line until the end of the program. Examples of interpreter-based language are BASIC, Logo and Smalltalk.

### COMPILER

The source code (in text format) will be converted into machine code which is a file consisting of binary machine code that can be executed on a computer. If the compiler encounters any errors, it records them in the program-listing file. When a user wants to run the program, the object program is loaded into the memory of the computer and the program instructions begin executing. A compiled code generally runs faster than programs based on interpreted language. Several programming languages like C++, Pascal and COBOL used compilers as their translators.

## Introducing Algorithms

An algorithm is a set of ordered and finite steps to solve a given problem.

The term *algorithm* refers to a method for solving problems. Algorithms can be described in English, but such descriptions are often misinterpreted because of the inherent complexity and ambiguity in a natural language. Hence, algorithms are frequently written in simple and more precise formats, such as flowcharts, decision trees, and decision tables, which represent an algorithm as a diagram, table, or graph. These techniques are often employed prior to writing programs in order to gain a better understanding of the solution.

## Lesson1: Introduction to Programming

These algorithm-development tools might help you in expressing a solution in an easy-to-use way, but they can't be directly understood by a computer. In order for a computer to understand your algorithm, you'll need to write a computer program in a more formal way by using a programming language like C#.

This section of the lesson focuses on two techniques for presenting your algorithms—namely, flowcharts and decision tables—that are more precise than a natural language but less formal and easier to use than a computer language

An algorithm has following characteristics:

1. Each and every instruction should be precise and unambiguous i.e. each and every instruction should be clear and should have only one meaning.
2. Each instruction should be performed in finite time.
3. One or more instructions should not be repeated infinitely. It means that the algorithm must terminate ultimately.
4. After the instructions are executed, the user should get the required results.

### INTRODUCING FLOWCHARTS

A *flowchart* is a graphical representation of an algorithm. A flowchart is usually drawn using Standard symbols. It represents the entire process in graphical.






- ◆ from start to finish,
- ◆ showing inputs, outputs,
- ◆ pathways and circuits,
- ◆ action or decision points,
- ◆ and ultimately, completion

Flowcharts are usually drawn using some standard symbols; however, some special symbols can also be developed when required

Table 1.1 shows Flowchart Standard Symbols

#### Table1.1

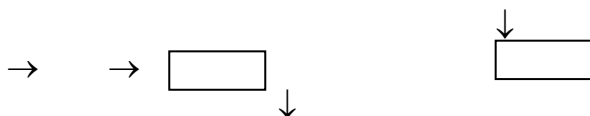
## Lesson1: Introduction to Programming

Symbol	Name	Function
	Start/end	An oval represents a start or end point.
	Arrows	A line is a connector that shows relationships between the representative shapes.
	Input/Output	A parallelogram represents input or output.
	Process	A rectangle represents a process.
	Decision	A diamond indicates a decision.

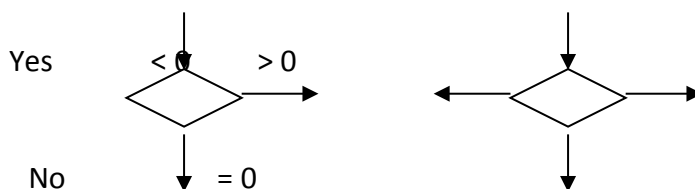
### GUIDELINES FOR DRAWING A FLOWCHART

The following are some guidelines in flowcharting:

1. In drawing a proper flowchart, all necessary requirements should be listed out in logical order.
2. The flowchart should be clear, neat and easy to follow. There should not be any room for ambiguity in understanding the flowchart.
3. The usual direction of the flow of a procedure or system is from left to right or top to bottom.
4. Only one flow line should come out from a process symbol.



5. Only one flow line should enter a decision symbol, but two or three flow lines, one for each possible answer, should leave the decision symbol.

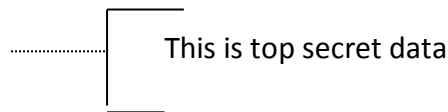


6. Only one flow line is used in conjunction with terminal symbol.





7. Write within standard symbols briefly. As necessary, you can use the annotation symbol to describe data or computational steps more clearly.

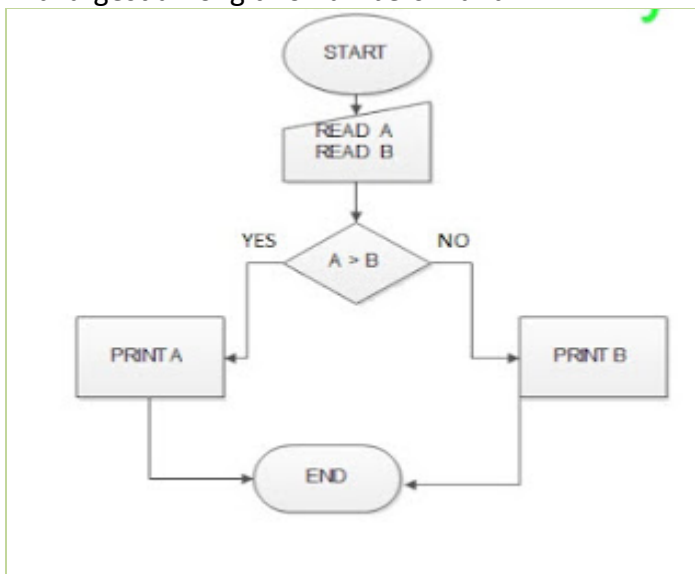


8. If the flowchart becomes complex, it is better to use connector symbols to reduce the number of flow lines. Avoid the intersection of flow lines if you want to make it more effective and better way of communication.
9. Ensure that the flowchart has a logical *start* and *finish*.
10. It is useful to test the validity of the flowchart by passing through it with a simple test data.

## Flowchart Examples

### Example1

Flowchart to find largest among two numbers A and B



In the above example flowchart

1. The execution of steps starts at start box,
2. The next step is Read A & B which is used to read/input the values of A & B.
3. The next step is decision box. The result of the decision box is always Yes or No. If the condition

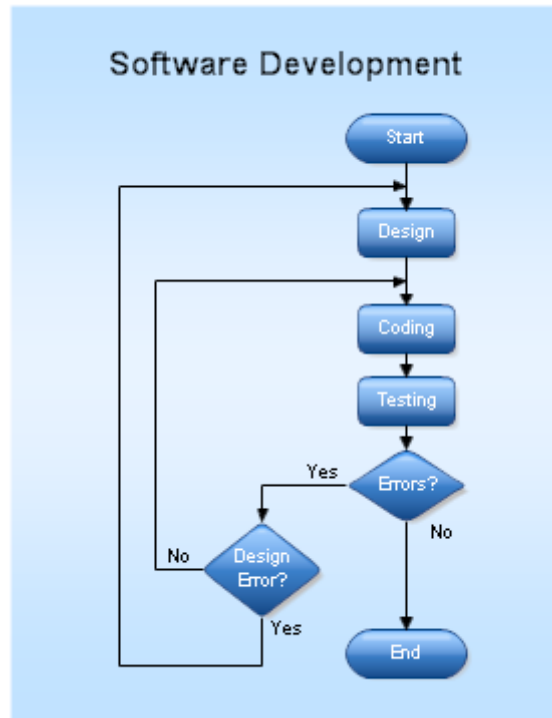
## Lesson1: Introduction to Programming

$A > B$  i.e value of A is greater than value of B then follow Yes arrow otherwise follow No arrow.

4. The next step is print A or print B. if  $A > B$  is yes then it will print A otherwise if  $A > B$  is no then follow no arrow and print B.

### Example 2

Flowchart Illustrating Software Development Life Cycle



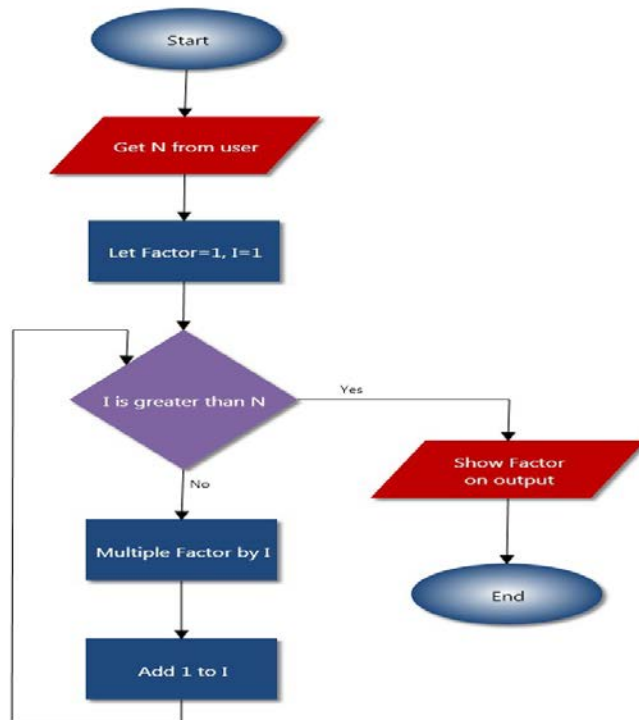
### Example 3

Flowchart to calculate Factorial of a Number:

Factorial of a number say 5 is calculated as

$$5 * 4 * 3 * 2 * 1 = 120.$$





## INTRODUCING DECISION TABLES

When an algorithm involves a large number of conditions, **decision tables** are a more compact and readable format for presenting the algorithm. **Decision tables** are used to lay out in tabular form all possible situations which a decision may encounter and to specify which action to take in each of these situations.

### Structure

Decision tables are typically divided into four quadrants. The upper left quadrant is the conditions quadrant. In this part of decision tables all conditions are listed. The lower left quadrant contains the actions. All conditions and actions are listed as text in these two quadrants. All possible states of conditions, in other words the indicators for conditions, are represented in the upper right quadrant of a decision table. The indicators of the actions are shown in the lower right quadrant of a decision table.

<b>IF...</b> Conditions	CONDITIONS	CONDITION INDICATORS
	ACTIONS	ACTION INDICATORS
Text		Rules/Indicators

## Lesson1: Introduction to Programming

In a nominal limited entry decision table a decision indicator is just a Y(Yes) or N(No) to indicate that the given condition must be fulfilled or not. An indicator of an action is an "X" for those actions to execute and "-" for those actions to drop.

In the right part of a decision table, each column is named by a rule number or rule identifier.

### Example 1

Table 1-2 presents a decision table for calculating a discount. This table generates a discount percentage depending on the quantity of product purchased. The table can be read as four quadrants.

The first quadrant (top left) specifies the conditions ("Quantity < 10," etc.). The second quadrant (top right) specifies the rules. The rules are the possible combinations of the outcome of each condition. The third quadrant (bottom left) specifies the action ("Discount," in this case), and the last quadrant (bottom right) specifies the action items corresponding to each rule.

Table 1.2

Quantity < 10	Y	N	N	N
Quantity < 50	Y	Y	N	N
Quantity < 100	Y	Y	Y	N
Discount	5%	10%	15%	20%

To find out which action item to apply, you must evaluate each condition to find the matching rule and then choose the action specified in the column with the matching rule. For example, if the value of "Quantity" in the test data is 75, then the first rule evaluates to "No," the second rule evaluates to "No," and the third rule evaluates to "Yes." Therefore, you will pick the action item from the column (N, N, and Y), which sets the discount at 15%.

### Example 2

The number and size of meeting rooms needed in an office building depends on the activities of the organization to be accommodated. The critical factors include the frequency formal meetings take place,

the number of persons involved in meetings and the frequency of central meetings involving all employees of the organization. The decision table in such a case may look as follows.

Required number and size of meeting rooms									
C <sub>1</sub>	Formal meetings take place	No	Yes						
C <sub>2</sub>	Number of persons involved in a formal meeting	-	< 5				[5,10)		
C <sub>3</sub>	Frequency (/week) formal meetings	-	< 2	[2,5)	>= 5		< 2	[2,5)	...
C <sub>4</sub>	Frequency (/week) formal central meetings	-	-	-	< 2	>= 2	-	-	...
A <sub>1</sub>	Required number of meeting rooms	0	0	0	1	2	3	0	...
A <sub>1</sub>	Required size of meeting rooms (m <sup>2</sup> )	0	0	0	15-30	15-30	15-30	0	...
		R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>	R <sub>5</sub>	R <sub>6</sub>	R <sub>7</sub>	...

### Introduction C#

C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft within its .NET initiative led by Anders Hejlsberg.

C# is a popular high-level programming language that allows you to write computer programs in a human-readable format. C# is a part of the .NET Framework and benefits from the runtime support and class libraries provided by the .Framework.

Each programming language provides its own set of vocabulary and grammar (also known as syntax). The .NET Framework provides a runtime execution environment for the C# program. The Framework also contains class libraries that provide a lot of reusable core functionality that you can use directly in your C# program.

#### Take Note

*The .NET Framework provides three major components:  
a runtime execution environment,  
a set of class libraries that provide a great deal of reusable functionality,  
and language compilers for C#, Visual Basic, and Managed C  
The .NET Framework supports multiple programming languages and also has support for adding additional languages to the system. Although the syntax and vocabulary of each language may differ, each can still use the base class libraries provided by the Framework.*

You will be using an integrated development environment (IDE) to develop your code. You can use either Visual Studio or the free Visual Studio Express edition to write your code. Either of these tools provides you with a highly productive environment for developing and testing your programs.

### ILLUSTRATION1: WRITE A C# PROGRAM

GET READY. To write a C# program, perform these steps:

1. Start Visual Studio. Select File >New Project. Select the Visual C# Console Application templates.
2. Type IntroducingCS in the Name box. Make sure that the Create directory for solution checkbox is checked, and enter the name Lesson01 in the Solution name box. Click OK to create the project.
3. When the project is created, you'll note that Visual Studio has already created a file named Program.cs and written a template for you.
4. Modify the template to resemble the following code:

```
using System;
namespace Lesson01
{
    class Program
    {
        static void Main(string[] args)
        {
```

## Lesson1: Introduction to Programming

```
        Console.WriteLine("Hello,World");  
    }  
}  
}
```

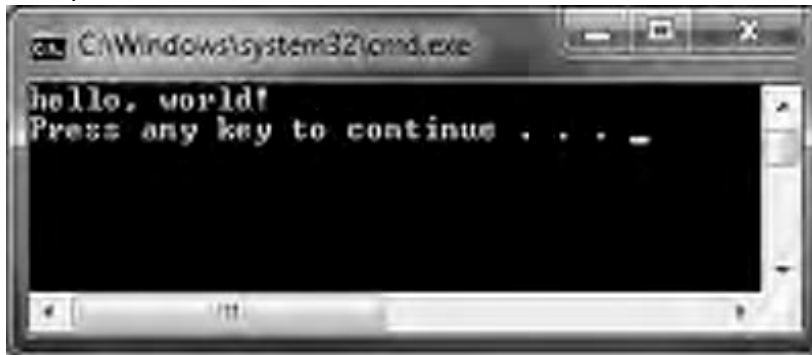
### Take Note

*C# is a Case – sensitive programming language. As a result, typing “Class” instead of “class” (for example) will result in a syntax error.*

5. Select Debug > Start Without Debugging, or press Ctrl F5. You can also execute the program by opening a command Window (cmd.exe) and then navigating to the project’s output folder, which by default is the bin\debug subfolder under the project’s location. Start the program by typing the name of the program in the command window and pressing Enter
6. You will see the output of the program in a command Window, as shown in Figure 1.1
7. Press a key to close the command Window.

Figure 1.1

Program output in a command window



*PAUSE. Leave the project open to use in the next exercise.*

The program you just created is trivial in what it does, but it is nonetheless useful for understanding program structure, build, and execution. Let’s first talk about the build and execution part. Here is what happens when you select the Debug > Start Without Debugging option in step 5 above:

1. Visual Studio invokes the C# compiler to translate the C# code into a lower-level language, Common Intermediate Language (CIL) code. This low-level code is stored in an executable file named (Lesson01.exe). The name of the output file can be changed by modifying a project’s properties.
2. Next, Visual Studio takes the project output and requests that the operating system execute it. This is when you see the command window displaying the output.

## Lesson1: Introduction to Programming

3. When the program finishes, Visual Studio displays the following message: “Press any key to continue . . .”. Note that this message is only generated when you run the program using the Start Without Debugging option.

When you select the Debug > Start Without Debugging menu option, Visual Studio automatically displays the prompt “Press any key to continue . . .”. The command window then stays open for you to review the output. If, however, you select the Debug > Start Debugging option, the command window closes as soon as program execution completes. It is important to know that the Start Debugging option provides debugging capabilities such as the ability to pause a running program at a given point and review the value of various variables in memory.

### Take Note

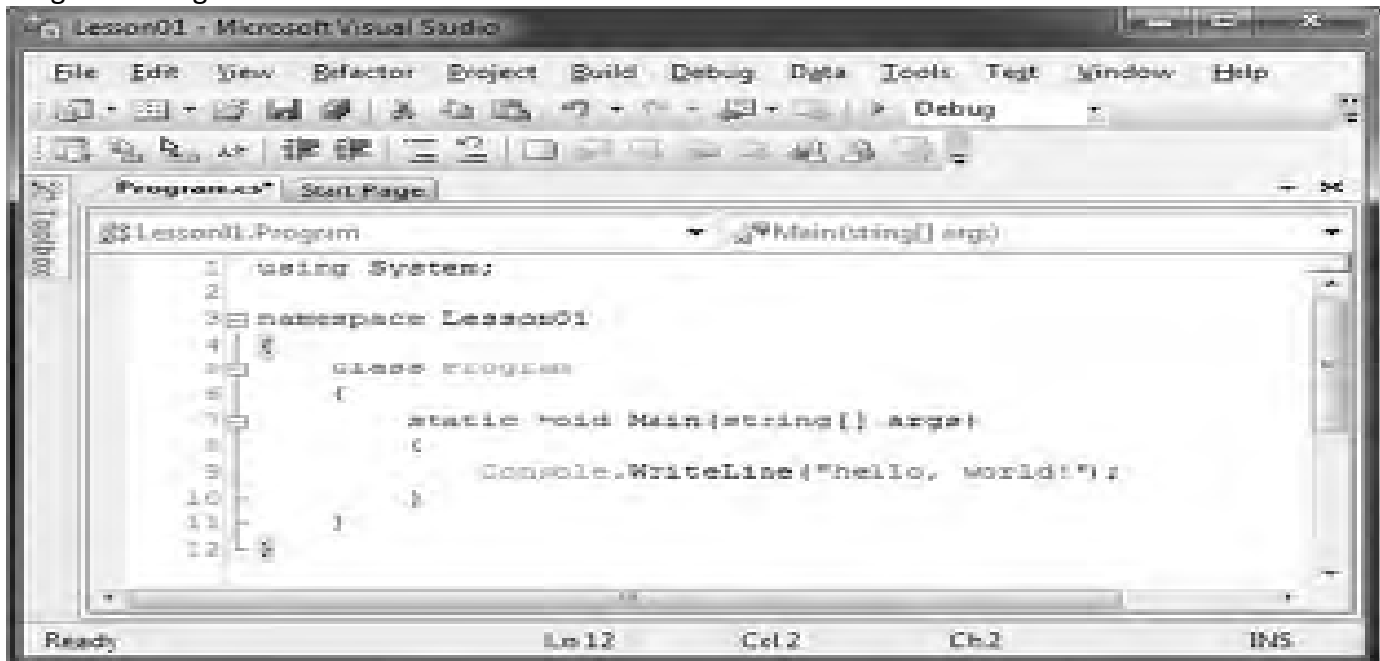
*Before Common Intermediate Language (CIL) code can be executed, it must first be translated for the architecture of the machine on which it will run. The .NET Framework’s runtime execution system takes care of this translation behind the scenes using a process called just-in-time compilation*

## UNDERSTANDING THE STRUCTURE OF A C# PROGRAM

Figure 1.2 depicts the program you created in the previous exercise with line numbers. Throughout this section, these numbers will be used to refer to different structures in the program.

Figure1.2

Program Listing line Numbers



### Take Note

*To enable the display of line numbers in Visual Studio, select the Tools > Options menu. Next, expand the Text Editor node and select C#. Finally, in the Display section, check the Line Numbers option*

## Lesson1: Introduction to Programming

A C# program is made of one or more classes. A class is a set of data and methods. For example, the code in Figure 1.2 defines a single class named Program on lines 5 through 11.

A class is defined by using the keyword `class` followed by the class name. The contents of a class are defined between an opening brace (`{`) and a closing brace (`}`).

Line 3 of the code in Figure 1.2 defines a namespace, `Lesson01`. Namespaces are used to organize classes and uniquely identify them. The namespace and the class names are combined together to create a fully qualified class name. For example, the fully qualified class name for the class `Program` is `Lesson01.Program`. C# requires that the fully qualified name of a class be unique. As a result, you can't have another class by the name `Program` in the namespace `Lesson01`, but you can have a class by the name `Program` in another namespace, say, `Lesson02`. Here, the class `Program` defined in the namespace `Lesson02` is uniquely identified by its fully qualified class name, `Lesson02.Program`.

The .NET Framework provides a large number of useful classes organized into many namespaces. The `System` namespace contains some of the most commonly used base classes. One such class in the `System` namespace is `Console`. The `Console` class provides functionality for console application input and output. The line 9 of the code in Figure 1.2 refers to the `Console` class and calls its `WriteLine` method. To access the `WriteLine` method in an unambiguous way, you must write it like this:

```
System.Console.WriteLine("hello, world!");
```

Because class names frequently appear in the code, writing the fully qualified class name every time will be tedious and make the program verbose. You can solve this problem by using the C# `using` directive (see the code in line 1 in Figure 1.2). The `using` directive allows you to use the classes in a namespace without having to fully qualify the class name.

The `Program` class defines a single method by the name `Main` (see lines 7 to 10 of the code listing in Figure 1-3). `Main` is a special method in that it also serves as an entry point to the program. When the runtime executes a program, it always starts at the `Main` method.

A program can have many classes and each class can have many methods, but it should have only one `Main` method. A method can in turn call other methods. In line 9, the `Main` method is calling the `WriteLine` method of the `System.Console` class to display a string of characters on the command window and that's how the message is displayed.

### TakeNote

*The `Main` method must be declared as static. A static method is callable on a class even when no instance of the class has been created. You will learn more about this in the next lesson*

## UNDERSTANDING VARIABLES

Variables provide temporary storage during the execution of a program.

## Lesson1: Introduction to Programming

The variables in C# are placeholders used to store values. A variable has a name and a data type. A variable's data type determines what values it can contain and what kind of operations may be performed on it. For example, the following declaration creates a variable named number of the data type int and assigns a value of 10 to the variable:

```
int number = 10;
```

When a variable is declared, a location big enough to hold the value for its data type is created in the computer memory. For example, on a 32-bit machine, a variable of data type int will need two bytes of memory. The value of a variable can be modified by another assignment, such as:

```
number = 20
```

The above code changes the contents of the memory location identified by the name number.

### TakeNote

*A variable name must begin with a letter or an underscore and can contain only letters, numbers, or underscores. A variable name must not exceed 255 characters. A variable must also be unique within the scope in which it is defined*

## UNDERSTANDING CONSTANTS

Constants are data fields or local variables whose value cannot be modified.

Constants are declared by using the const keyword. For example, a constant can be declared as follows:

```
const int i = 10;
```

This declares a constant i of data type int and stores a value of 10. Once declared, the value of the constant cannot be changed.

## UNDERSTANDING DATA TYPES

Data types specify the type of data that you work with in a program. The data type defines the size of memory needed to store the data and the kinds of operations that can be performed on the data. C# is a strongly-typed language. Every variable and constant has a type, as does every expression that evaluates to a value.

C# provides several built-in data types that you can use in your programs. You can also define new types by defining a data structure, such as a class or a struct. This lesson focuses on some of the most commonly used built-in data types.

The information stored in a type can include the following:

1. The storage space that a variable of the type requires.
2. The maximum and minimum values that it can represent.
3. The members (methods, fields, events, and so on) that it contains.
4. The base type it inherits from.
5. The location where the memory for variables will be allocated at run time.

## 6. The kinds of operations that are permitted.

Table 1.3 lists several commonly used built-in data types available in C#. The sizes listed in the table refer to a computer running a 32-bits operating system such as Windows 7, 32-bit.

**Table 1.3**

Short Name	.NET Class	Type	Width(bits)	Range (bits)
byte	<a href="#">Byte</a>	Unsigned integer	8	0 to 255
sbyte	<a href="#">SByte</a>	Signed integer	8	-128 to 127
int	<a href="#">Int32</a>	Signed integer	32	-2,147,483,648 to 2,147,483,647
uint	<a href="#">UInt32</a>	Unsigned integer	32	0 to 4294967295
short	<a href="#">Int16</a>	Signed integer	16	-32,768 to 32,767
ushort	<a href="#">UInt16</a>	Unsigned integer	16	0 to 65535
long	<a href="#">Int64</a>	Signed integer	64	-9223372036854775808 to 9223372036854775807
ulong	<a href="#">UInt64</a>	Unsigned integer	64	0 to 18446744073709551615
float	<a href="#">Single</a>	Single-precision floating point type	32	-3.402823e38 to 3.402823e38
double	<a href="#">Double</a>	Double-precision floating point type	64	-1.79769313486232e308 to 1.79769313486232e308
char	<a href="#">Char</a>	A single Unicode character	16	Unicode symbols used in text
bool	<a href="#">Boolean</a>	Logical Boolean type	8	True or false
string	<a href="#">String</a>	A sequence of characters		
decimal	<a href="#">Decimal</a>	Precise fractional or integral type that can represent decimal numbers with 29 significant digits	128	$\pm 1.0 \times 10e-28$ to $\pm 7.9 \times 10e28$



### TakeNote

*Size is given in Bytes and 8 bits = 1 byte*

*The unsigned versions of short, int, and long are ushort, uint, and ulong, respectively. The unsigned types have the same size as their signed versions but store much larger ranges of only positive values.*

All the data types listed in Table 1.3 are value types except for string, which is a reference type. The variables that are based directly on the value types contain the value. In the case of the reference type, the variable holds the address of the memory location where the actual data is stored. You will learn more about the differences between value types and reference types in Lesson 2.

## UNDERSTANDING ARRAYS

An array is a collection of items in which each item can be accessed by using a unique index.

An array in C# is commonly used to represent a collection of items of similar type. A sample array declaration is shown in the following code:

```
int[] numbers = { 1, 2, 3, 4, 5 };
```

This declaration creates an array identified by the name numbers. This array is capable of storing a collection of five integers. This declaration also initializes each of the array items respectively by the numbers 1 through 5.

Any array item can be directly accessed by using an index. In the .NET Framework, array indexes are zero-based. This means that to access the first element of an array, you use the index 0; to access the second element, you use the index 2, and so on.

To access an individual array element, you use the name of the array followed by the index enclosed in square brackets. For example, numbers[0] will return the value 1 from the above-declared array, and numbers[4] will return the value 5. It is illegal to access an array outside its defined boundaries. For example, you'll get an error if you try to access the array element numbers[5].

## UNDERSTANDING OPERATORS

Operators are symbols that specify which operation to perform on the operands before returning a result. It tells the compiler to perform specific mathematical or logical manipulations. C# is rich in built-in operators and provides the following type of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Primary Operators

## Lesson1: Introduction to Programming

Depending on how many operands are involved, there are three kinds of operators:

1. **Unary operators:** The unary operators work with only one operand. Can be used as prefix operator or postfix operator  
Examples include ++x(prefix), x++ (postfix) where x is of integer data type or isEven, where isEven is of Boolean data type.
2. **Binary operators:** The binary operators take two operands. Examples include x + y or x > y.
3. **Ternary operators:** Ternary operators take three operands. There is just one ternary operator, ? : in C#.

### Arithmetic Operators

Following table shows all the arithmetic operators supported by C#. Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator increases integer value by one	A++ will give 11
--	Decrement operator decreases integer value by one	A-- will give 9

### Relational Operators

Following table shows all the relational operators supported by C#. Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.

## Lesson1: Introduction to Programming

<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

### Logical Operators

Following table shows all the logical operators supported by C#. Assume variable A holds Boolean value true and variable B holds Boolean value false, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A    B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

### Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. The truth tables for &, |, and ^ are as follows:

p	q	p & q	p   q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

### Assignment Operators

There are following assignment operators supported by C#:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A +

## Lesson1: Introduction to Programming

		B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
=	bitwise inclusive OR and assignment operator	C  = 2 is same as C = C   2

### Primary Operators

There are few other important operators including sizeof, typeof and ? : supported by C#.

Operator	Description	Example
sizeof()	Returns the size of a data type.	sizeof(int), will return 4.
typeof()	Returns the type of a class.	typeof(StreamReader);
&	Returns the address of an variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
? :	Conditional Expression	If Condition is true ? Then value X :

## Lesson1: Introduction to Programming

		Otherwise value Y
is	Determines whether an object is of a certain type.	If( Ford is Car) // checks if Ford is an object of the Car class.
as	Cast without raising an exception if the cast fails.	Object obj = new StringReader("Hello"); StringReader r = obj as StringReader;

Often, expressions involve more than one operator. In this case, the compiler needs to determine which operator takes precedence over the other(s). The following table lists the C# operators grouped in order of precedence. Operators within each group have equal precedence.

Operator category	Operators
Primary	<a href="#">x.y</a> <a href="#">f(x)</a> <a href="#">a[x]</a> <a href="#">x++</a> <a href="#">x--</a> <a href="#">new</a> <a href="#">typeof</a> <a href="#">checked</a> <a href="#">unchecked</a> <a href="#">default(T)</a> <a href="#">delegate</a> <a href="#">sizeof</a> <a href="#">-&gt;</a> <a href="#">_</a>
Unary	<a href="#">+x</a> <a href="#">-x</a> <a href="#">!x</a> <a href="#">~x</a> <a href="#">++x</a> <a href="#">--x</a> <a href="#">(T)x</a> <a href="#">await</a> <a href="#">&amp;x</a> <a href="#">*x</a>
Multiplicative	<a href="#">x * y</a> <a href="#">x / y</a> <a href="#">x % y</a>
Additive	<a href="#">x + y</a> <a href="#">x - y</a>
Shift	<a href="#">x &lt;&lt; y</a>

## Lesson1: Introduction to Programming

	<a href="#"><u>x &gt;&gt; y</u></a>
Relational and type testing	<a href="#"><u>x &lt; y</u></a> <a href="#"><u>x &gt; y</u></a> <a href="#"><u>x &lt;= y</u></a> <a href="#"><u>x &gt;= y</u></a> <a href="#"><u>is</u></a> <a href="#"><u>as</u></a>
Equality	<a href="#"><u>x == y</u></a> <a href="#"><u>x != y</u></a>
Logical AND	<a href="#"><u>x &amp; y</u></a>
Logical XOR	<a href="#"><u>x ^ y</u></a>
Logical OR	<a href="#"><u>x   y</u></a>
Conditional AND	<a href="#"><u>x &amp;&amp; y</u></a>
Conditional OR	<a href="#"><u>x    y</u></a>
Null-coalescing	<a href="#"><u>x ?? y</u></a>
Conditional	<a href="#"><u>?:</u></a>
Assignment and lambda expression	<a href="#"><u>x = y</u></a> <a href="#"><u>x += y</u></a> <a href="#"><u>x -= y</u></a> <a href="#"><u>x *= y</u></a> <a href="#"><u>x /= y</u></a> <a href="#"><u>x %= y</u></a> <a href="#"><u>x &amp;= y</u></a> <a href="#"><u>x  = y</u></a> <a href="#"><u>x ^= y</u></a> <a href="#"><u>x &lt;&lt;= y</u></a> <a href="#"><u>x &gt;&gt;= y</u></a>

### Unary operator Revisited

The unary increment operator (++) adds 1 to the value of an identifier. Similarly, the decrement (--) operator subtracts 1 from the value of an identifier. The unary increment and decrement can be used either as prefixes or suffixes. For example:

```
int x = 10;  
x++; //value of x is now 11  
++x; //value of x is now 12
```

## Lesson1: Introduction to Programming

However, the way the unary increment and decrement operators work when used as part of an assignment can affect the results. In particular, when the unary increment and decrement operators are used as prefixes, the current value of the identifier is returned prior to the increment or decrement. On the other hand, when used as a suffix, the value of the identifier is returned after the increment or decrement is complete. To understand what this means, consider the following code sample:

```
int y = x++; // the value of y is 12
int z = ++x; // the value of z is 14
```

Here, in the first statement, the value of x is returned prior to the increment. As a result ,after the statement is executed, the value of y is 12 and the value of x is 13. In contrast, in the second statement, the value of x is incremented prior to returning its value for assignment. As a result, after the statement is executed, the value of both x and z is 14.

### UNDERSTANDING METHODS

Methods are code blocks containing a series of statements. It is a group of statements that together perform a task. Methods can receive input via arguments and can return a value to the caller.

Every C# program has at least one class with a method named Main.To use a method, you need to:

- Define the method
- Call the method

#### Defining Methods in C#

When you define a method, you basically declare the elements of its structure. The syntax for defining a method in C# is as follows:

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
    Method Body
}
```

Following are the various elements of a method:

**Access Specifier:** This determines the visibility of a variable or a method from another class. . You'll learn more about these modifiers in Lesson 2.

**Return type:** A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is void.

**Method name:** Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.

## Lesson1: Introduction to Programming

**Parameter list:** Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.

**Method body:** This contains the set of instructions needed to complete the required activity.

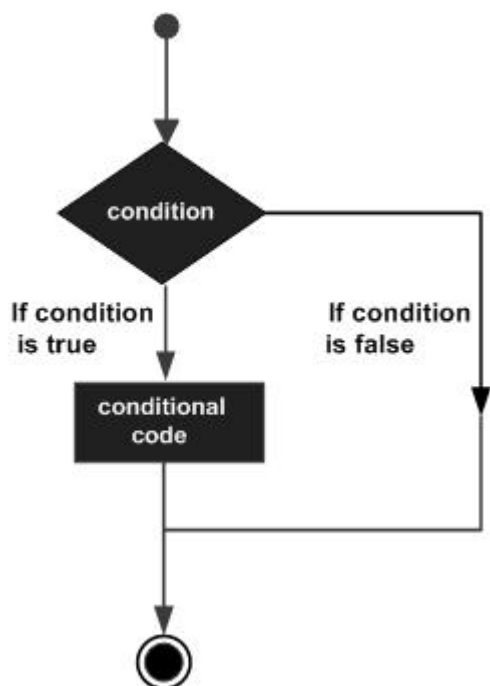
In the previous code listing, you learned about the Main method. Methods are where the action is in a program. More precisely, a method is a set of statements that are executed when the method is called.

The Main method doesn't return a value back to the calling code. This is indicated by using the void keyword. If a method were to return a value, the appropriate data type for the return value would be used instead of void.

### OBJECTIVE2: Decision Structures

Decision structures introduce decision-making ability into a program. They enable you to branch to different sections of the code depending on the truth value of a Boolean expression. Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general Flowchart typical decision making structure found in most of the programming



The decision-making control structures in C# are the if, if-else, and switch statements.



## Lesson1: Introduction to Programming

C# provides following types of decision making statements.

Statement	Description
<u>if statement</u>	An <b>if statement</b> consists of a boolean expression followed by one or more statements.
<u>if...else statement</u>	An <b>if statement</b> can be followed by an optional <b>else statement</b> , which executes when the boolean expression is false.
<u>switch statement</u>	A <b>switch</b> statement allows a variable to be tested for equality against a list of values.

The following sections discuss each of these statements in more detail.

### The If Statement

The **if statement** will execute a given sequence of statements only if the corresponding Boolean expression evaluates to true

The syntax of an if statement in C# is

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
```

Sometimes in your programs, you will want a sequence of statements to be executed only if a certain condition is true. In C#, you can do this by using the if statement. Take the following steps to create a program that uses an if statement.

### ILLUSTRATION2: Use of If statement

GET READY. To use the if statement, perform the following tasks:

1. Add a new Console Application project (named if\_Statement) to the Lesson01 solution.
2. Add the following code to the Main method of the Program.cs class:

```
int number1 = 10;
int number2 = 20;
if (number2 > number1)
{
    Console.WriteLine("number2 is greater than number1");
}
```

3. Select Debug > Start Without Debugging, or press Ctrl+F5.

## Lesson1: Introduction to Programming

4. You will see the output of the program in a command window.
5. Press a key to close the command window.

**PAUSE. Leave the project open to use in the next exercise.**

In C# code, the parentheses surrounding the condition are required. However, the braces are optional if there is only one statement in the code block. So, the above if statement is equivalent to the following:

```
if (number2 > number1)
    Console.WriteLine("number2 is greater than number1");
```

In contrast, look at this example:

```
if (number2 > number1)
    Console.WriteLine("number2 is greater than number1"); Console.WriteLine(number2);
```

Here, only the first `Console.WriteLine` statement is part of the if statement. The second `Console.WriteLine` statement is always executed regardless of the value of the Boolean expression.

For clarity, it is always a good idea to enclose the statement that needs to be conditionally executed in braces.

If statements can also be nested within other if statements, as in the following example:

```
int number1 = 10;
if (number1 > 5)
{
    Console.WriteLine("number1 is greater than 5");
    if (number1 < 20)
    {
        Console.WriteLine("number1 is less than 20");
    }
}
```

Because both the conditions evaluate to true, this code would generate the following output:

number1 is greater than 5 number1 is less than 20

But what would happen if the value of `number1` was 25 instead of 10 prior to the execution of the outer if statement? In this case, the first Boolean expression will evaluate to true, but the second Boolean expression will evaluate to false and the following output will be generated:

number1 is greater than 5

## The if-else Statement

## Lesson1: Introduction to Programming

The if-else statement allows your program to perform one action if the Boolean expression evaluates to true and a different action if the Boolean expression evaluates to false.

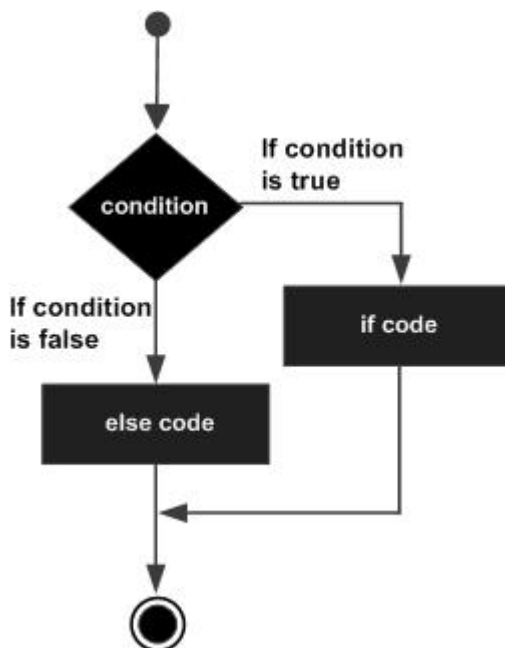
An if statement can be followed by an optional else statement, which executes when the boolean expression is false.

Syntax:

The syntax of an if...else statement in C# is:

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
else
{
    /* statement(s) will execute if the boolean expression is false */
}
```

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed



Take the following steps to create an example program that uses the if-else statement.

### ILLUSTRATION 3: USE THE IF-ELSE STATEMENT

GET READY. To use the if-else statement, do the following:

1. Add a new Console Application project (named ifelse\_Statement) to the Lesson01 solution.
2. Add the following code to the Main method of the Program.cs class:  
TestIfElse(10); // call to method named TestIfElse
3. Next, add the following method to the Program.cs class:

```
public static void TestIfElse(int n)
{
    If(n<10)
    {
        Console.WriteLine("n is less than 10");
    }
    else if (n < 20)
    {
        Console.WriteLine("n is less than 20");
    }
    else if (n < 30)
    {
        Console.WriteLine("n is less than 30");
    }
    else
    {
        Console.WriteLine("n is equal or greater than 30");
    }
}
```

4. Select Debug > Start Without Debugging, or press Ctrl+F5.
5. You will see the output of the program in a command window.
6. Press a key to close the command window.
7. Modify the Main method's code to call the TestIfElse method with different values.(like TestIfElse(20)); Notice how a different branch of the if-else statement is executed as a result of your changes.

**PAUSE. Leave the project open to use in the next exercise.**

Here, the code in the TestIfElse method combines several if-else statements to test for multiple conditions. For example, if the value of n is 25, then the first two conditions (n < 10 and n < 20) will evaluate to false, but the third condition (n < 30) will evaluate to true. As a result, the method will print the following output:

n is less than 30

### The Switch Statement

The switch statement allows multi-way branching. In many cases, using a switch statement can simplify a complex combination of if-else statements.

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

Syntax: The syntax for a switch statement in C# is as follows:

```
switch(expression)
{
    case constant-expression :
        statement(s);
        break; /* optional */
    case constant-expression :
        statement(s);
        break; /* optional */

    /* you can have any number of case statements */
    default : /* Optional */
        statement(s);
}
```

The following rules apply to a switch statement:

1. The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
2. You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
3. The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
4. When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
5. When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
6. Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
7. A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

*TakeNote*

*The expression following the case statement must be a constant expression and must be of the matching data type*

to the switch expression.

#### Illustration4: USE THE SWITCH STATEMENT

GET READY. To use the switch statement, do the following:

1. Add a new Console Application project named switch\_Statement to the Lesson01 solution.
2. Add the following code to the Main method of the Program.cs class:  
TestSwitch(10, 20, '+');
3. Add the following method to the Program.cs class:

```
public static void TestSwitch(int op1, int op2, char opr)
{
    int result;
    switch (opr)
    {
        case '+':
            result = op1 + op2;
            break;
        case '-':
            result = op1 - op2;
            break;
        case '*':
            result = op1 * op2;
            break;
        case '/':
            result = op1 / op2;
            break;
        default:
            Console.WriteLine("Unknown Operator");
            return;
    }
    Console.WriteLine("Result: {0}", result);
    return;
}
```

4. Select Debug > Start Without Debugging, or press Ctrl+F5.
5. You will see the output of the program in a command window.
6. Press a key to close the command window.
7. Modify the Main method's code to call the TestSwitch method with different values. Notice how a different branch of the switch statement is executed as a result of your changes.
8. TakeNote

The Console.Write and the Console.WriteLine methods can use format strings such as "Results:

*{0}" to format the out put. Here, the string {0} stands for the first argument provided after the format string. In the TestSwitch method, the format string "{0}" is replaced by the value of the following argument, result.*

**PAUSE. Leave the project open to use in the next exercise**

Here, the TestSwitch method accepts two operands (op1 and op2) and an operator (opr) and evaluates the resulting expression. The value of the switch expression is compared to the case statements in the switch block. If there is a match, the statements following the matching case are executed. If none of the case statements match, then control is transferred to the optional default branch.

Note that there is a break statement after each case. The break statement terminates the switch statement and transfers control to the next statement outside the switch block. Using a break ensures that only one branch is executed and helps avoid programming mistakes. In fact, if you specify code after the case statement, you must include break (or another control- transfer statement, such as return) to make sure that control does not fall through from one case label to another

However, if no code is specified after the case statement, it is okay for control to fall through to the subsequent case statement. The following code demonstrates how this might be useful:

```
public static void TestSwitchFallThrough()
{
    DateTime dt = DateTime.Today;
    switch (dt.DayOfWeek)
    {
        case DayOfWeek.Monday:
        case DayOfWeek.Tuesday:
        case DayOfWeek.Wednesday:
        case DayOfWeek.Thursday:
        case DayOfWeek.Friday:

            Console.WriteLine("Today is a weekday")
            break;
        default:
            Console.WriteLine("Today is a weekend day");
            break;
    }
}
```

Here, if the value of expression `dt.DayOfWeek` is `DayOfWeek.Monday`, then the first case is matched, but because no code (or a control-transfer statement) is specified, the execution will fall through the next statement, resulting in display of the message “Today is a weekday ” on the command window.

### TakeNote

You can decide whether to use if-else statements or a switch statement based on the nature of the comparison and readability of the code. For example, the code of the `TestIfElse` method makes decisions based on conditions that are more suited for use with if-else statements. In the `TestSwitch` method, the decisions are based on constant values, so the code is much more readable when written as a switch statement.

## OBJECTIVE3: Understanding Repetition Structures

There may be a situation, when you need to execute a block of code several number of times. A loop statement allows us to execute a statement or group of statements multiple times.

C# has four different control structures that allow programs to perform repetitive tasks: the while loop, the do-while loop, the for loop, and the foreach loop.

These repetition control statements can be used to execute the statements in the loop body a number of times, depending on the loop termination criterion.

C# provides following types of loop to handle looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body

### Loop Control Statements:

A loop can also be terminated by using one of several control transfer statements that transfer control outside the loop. These statements are `break`, `goto`, `return`, or `throw`. Finally, the `continue` statement can be used to pass control to next iteration of the loop without exiting the loop.



## Lesson1: Introduction to Programming

C# provides the following control statements.

Control Statement	Description
break statement	Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

The Infinite Loop:

A loop becomes infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

### While Loop

The while loop repeatedly executes a block of statements until a specified Boolean expression evaluates to false. Or while loop statement in C# repeatedly executes a target statement as long as a given condition is true.

Syntax: The syntax of a while loop in C# is

```
while(condition)
{
    statement(s);
}
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

*TakeNote*

With the while loop, the Boolean test must be placed inside parentheses. If more than one statement needs to be executed as part of the while loop, they must be placed together inside curly braces

### Illustration5: USE THE WHILE LOOP

GET READY. To use the while loop, perform the following tasks:

1. Add a new Console Application project named while\_Statement to the Lesson01 solution.
2. Add the following code to the Main method of the Program.cs class:

WhileTest();

3. Add the following method to the Program.cs class:

```
private static void WhileTest()
{
    int i = 1;
    while(i<=5)
    Console.WriteLine("The value of i = {0}", i);
    i++;
}
```

4. Select Debug > Start Without Debugging, or press Ctrl+F5.
5. You will see the output of the program in a command window.
6. Press a key to close the command window.

**PAUSE. Leave the project open to use in the next exercise.**

In this exercise, the variable *i* is assigned the value 1. Next, the condition in the while loop is evaluated. Because the condition is true ( $1 \leq 5$ ), the code within the while statement block is executed. The value of *i* is written in the command window, and then the value of *i* is increased by 1 so that it becomes 2. Control then passes back to the while statement, and the condition is evaluated again. Because the condition is still true ( $2 \leq 5$ ), the statement block is executed yet again. The loop continues until the value of *i* becomes 6 and the condition in the while loop becomes false ( $6 \leq 5$ ). The above method, when executed, generates the following output:

```
The value of i = 1
The value of i = 2
The value of i = 3
The value of i = 4
The value of i = 5
```

The statement in the loop, which increments the value of *i*, plays a critical role. If you miss this statement, the termination condition will never be achieved, and as a result, you will have a never-ending loop.

In most cases, to have a well-designed while loop, you must have three parts:

1. **Initializer:** The initializer sets the loop counter to the correct starting value. In the above example, the variable *i* is set to 1 before the loop begins.

## Lesson1: Introduction to Programming

2. Loop test: The loop test specifies the termination condition for the loop. In the above example, the expression (i <= 5) is the condition expression.
3. Termination expression: The termination expression changes the value of the loop counter in such a way that the termination condition is achieved. In the above example, the expression i++ is the termination expression.

### TakeNote

*To avoid an infinite loop, you must make sure that your while loop is designed in such a way that it leads to termination.*

## Understanding the Do-While Loop

The do-while loop repeatedly executes a block of statements until a specified Boolean expression evaluates to false. The do-while loop tests the condition at the bottom of the loop.

The do-while loop is similar to the while loop but, unlike the while loop, the body of the do-while loop must be executed at least once.

Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop checks its condition at the bottom of the loop.

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax: The syntax of a do...while loop in C# is

```
do
{
    statement(s);

}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested. If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

### Illustration6: USE THE DO WHILE LOOP

## Lesson1: Introduction to Programming

GET READY. To use the do-while loop, do the following tasks:

1. Add a new Console Application project named `dowhile_Statement` to the Lesson01 solution.
2. Add the following code to the Main method of the Program.cs class:  
`DoWhileTest();`
3. Add the following method to the Program.cs class:

```
private static void DoWhileTest()
{
    int i = 1;
    do
    {
        Console.WriteLine("The value of i = {0}", i); i++;
    }
    while (i <= 5);
}
```

4. Select Debug > Start Without Debugging, or press Ctrl+F5.
5. You'll see the output of the program in a command window.
6. Press a key to close the command window.

**PAUSE.** Leave the project open to use in the next exercise.

In this exercise, after the variable `i` is assigned the value 1, the control directly enters the loop. At this point, the code within the do-while statement block is executed. Specifically, the value of `i` is written in the command window and the value of `i` is increased by 1 so that it becomes 2. Next, the condition for the do-while loop is evaluated. Because the condition is still true ( $2 \leq 5$ ), control passes back to the do-while statement, and the statement block is executed again. The loop continues until the value of `i` becomes 6 and the condition of the do-while loop becomes false ( $6 \leq 5$ ). The above method, when executed, generates the same output as the `WhileTest` method.

The choice between a while loop and a do-while loop depends on whether you want the loop to execute at least once. If you want the loop to execute zero or more times, choose the while loop. In contrast, if you want the loop to execute one or more times, choose the do-while loop.

### Understanding the For Loop

The for loop combines the three elements of iteration—the initialization expression, the termination condition expression, and the counting expression—into a more readable code.

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

## Lesson1: Introduction to Programming

Syntax: The syntax of a for loop in C# is

```
for ( init; condition; increment )  
{  
    statement(s);  
}
```

Here is the flow of control in a for loop:

1. The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
3. After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

### TakeNote

*With the for loop, the three control expressions must be placed inside parentheses. If more than one statement needs to be executed as part of the loop, these statements must be placed together inside curly braces.*

## Illustration7: USE THE FOR LOOP

GET READY. To use the for loop, perform the following tasks:

1. Add a new Console Application project named for\_Statement to the Lesson01 solution.
2. Add the following code to the Main method of the Program.cs class:  
ForTest();
3. Add the following method to the Program.cs class:

```
private static void ForTest()  
{  
    for(int i = 1; i<= 5; i++)  
    {  
        Console.WriteLine("The value of i = {0}", i);  
    }  
}
```

- 
4. Select Debug > Start Without Debugging, or press Ctrl+F5.
  5. You will see the output of the program in a command window.
  6. Press a key to close the command window.

### **PAUSE. Leave the project open to use in the next exercise**

The ForTest method, when executed, generates the same output as the WhileTest method. Here, the variable *i* is created within the scope of the for loop and its value is assigned to 1. The loop continues as long as the value of *i* is less than or equal to 5. After the loop body, the count-expr is evaluated and the control goes back to the cond-expr.

All the control expressions of a for loop are optional. For example, you can omit all the expressions to create an infinite loop like this:

```
for (; ;)  
{  
    //do nothing  
}
```

### Understanding the Foreach Loop

The foreach loop can be thought of as enhanced version of the for loop for iterating through collections such as arrays and lists. The general form of the foreach statement is as follows:

foreach (ElementType element in collection) statement

The control expressions for the foreach loop must be placed inside parentheses. If more than one statement needs to be executed as part of the loop, these statements must be placed together inside curly braces.

Take the following steps to create a program that shows how the foreach loop provides a simple way to iterate through a collection.

### **Illustration8: USE THE FOREACH LOOP**

GET READY. To use the foreach loop, do the following:

1. Add a new Console Application project named foreach\_Statement to the Lesson01 solution.
2. Add the following code to the Main method of the Program.cs class:  
    ForEachTest();
3. Add the following method to the Program.cs class:

## Lesson1: Introduction to Programming

```
private static void ForEachTest()
{
    int[] numbers = { 1, 2, 3, 4, 5 };
    foreach (int i in numbers)
    {
        Console.WriteLine("The value of i = {0}", i);
    }
}
```

4. Select Debug > Start Without Debugging, or press Ctrl+F5.
5. You will see the output of the program in a command window.
6. Press a key to close the command window.

**PAUSE. Leave the project open to use in the next exercise.**

In this exercise, the loop sequentially iterates through every element of the collection, numbers it, and displays it in the command window. This method generates the same output as the ForTest method.

### Understanding Recursion

Recursion is a programming technique that causes a method to call itself in order to compute a result. Recursion and iteration are related. You can write a method that generates the same results with either recursion or iteration. Usually, the nature of the problem itself will help you choose between an iterative or a recursive solution. For example, a recursive solution is more elegant when you can define the solution of a problem in terms of a smaller version of the same problem.

To better understand this concept, take the example of the factorial operation from mathematics. The general recursive definition for n factorial (written n!) is as follows:

$$n! = 1 \text{ if } n = 0, (n - 1)! \times n \text{ if } n > 0.$$

According to this definition, if the number is 0, the factorial is one. If the number is larger than zero, the factorial is the number multiplied by the factorial of the next smaller number.

For example, you can break down 3! like this:

$$\begin{aligned} 3! &= 3 * 2! \rightarrow 3 * 2 * 1! \rightarrow 3 * 2 * 1 * 0! \rightarrow \\ &3 * 2 * 1 * 1 \rightarrow 6. \end{aligned}$$

Take the following steps to create a program that presents a recursive solution to a factorial problem.

### Illustration9: USE THE RECURSIVE METHOD

## Lesson1: Introduction to Programming

GET READY. To use the recursive method, perform the following actions:

1. Add a new Console Application project named RecursiveFactorial to the Lesson01 solution.
2. Add the following code to the Main method of the Program.cs class:  
`Console.WriteLine("5! = {0}", Factorial(5)); // call to method and it returns value so displayed it.`
3. Add the following method to the Program.cs class:

```
public static int Factorial(int n)
{
    If(n==0)
    {
        return 1; // base condition
    }
    else
    {
        Return n * Factorial(n-1) // recursive
    }
}
```

4. Select Debug > Start Without Debugging, or press Ctrl+F5.
5. You will see the output of the program in a command window.
6. Press a key to close the command window. Modify the Main method to pass a different value to the Factorial method

**PAUSE. Leave the project open to use in the next exercise.**

As seen in the above exercise, a recursive solution has two distinct parts:

1. Base case: This is the part that specifies the terminating condition and doesn't call the method again. The base case in the Factorial method is  $n == 0$ . If you don't have a base case in your recursive algorithm, you create an infinite recursion. An infinite recursion will cause your computer to run out of memory and throw a `System. StackOverflowException` exception.
2. Recursive case: This is the part that moves the algorithm toward the base case. The recursive case in the Factorial method is the else part, where you call the method again but with a smaller value progressing toward the base case.

### Objective 4: Understanding Exception handling.

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. Exceptions provide a way to transfer control from one part of a program to another.



## Lesson1: Introduction to Programming

The .NET Framework supports standard exception handling to raise and handle runtime errors. In this section, you'll learn how to use the try, catch, and finally keywords to handle exceptions.

An exception is an error condition that occurs during the execution of a C# program. When this happens, the runtime creates an object to represent the error and "throws" it. Unless you "catch" the exception by writing proper exception-handling code, program execution will terminate.

For example, if you attempt to divide an integer by zero, a `DivideByZeroException` exception will be thrown. In the .NET Framework, an exception is represented by using an object of the `System.Exception` class or one of its derived classes. There are predefined exception classes that represent many commonly occurring error situations, such as the `DivideByZeroException` mentioned earlier. If you are designing an application that needs to throw any application-specific exceptions, you should create a custom exception class that derives from the `System.Exception` class.

### Handling Exceptions

To handle exceptions, place the code that throws the exceptions inside a try block and place the code that handles the exceptions inside a catch block.

The following exercise shows how to use a try-catch block to handle an exception. The exercise uses the `File.OpenText` method to open a disk file. This statement will execute just fine in the normal case, but if the file (or permission to read the file) is missing, then an exception will be thrown.

### Illustration 10: HANDLE EXCEPTIONS

GET READY. To handle exceptions, perform the following steps:

1. Add a new Console Application project named `HandlingExceptions` to the Lesson01 solution.
2. Add the following code to the Main method of the `Program.cs` class:  
`ExceptionTest();`
3. Add the following method to the `Program.cs` class:

```
private static void ExceptionTest()
{
    StreamReader sr = null;
    try
    {
        sr = File.OpenText(@"c:\data.txt");
        Console.WriteLine(sr.ReadToEnd());
    }
    catch (FileNotFoundException fnfe)
    {
    }
```

## Lesson1: Introduction to Programming

```
        Console.WriteLine(fnfe.Message);  
    }  
    catch(Exception ex)  
    {  
        Console.WriteLine(ex.Message);  
    }  
}
```

4. Create a text file ("data.txt") using Notepad or Visual Studio on the C: drive. It is acceptable to create the file at a different location, but if you do so, remember to modify the file location in the program. Enter some text in the file.
5. Select Debug > Start Without Debugging, or press Ctrl+F5.
6. You will see the contents of the text file displayed in a command window.
7. Press a key to close the command window.
8. Delete the data.txt file and run the program again. This time, you'll get a FileNotFoundException exception, and an appropriate message will be displayed in the output window.

### TakeNote

*The StreamReader class is part of the System.IO namespace. When running this code, you'll need to add a using directive for the System. IO namespace*

*In the ExceptionTest method, it is incorrect to change the order of the two catch blocks. The more specific exceptions need to be listed before the generic exceptions, or else you'll get compilation errors.*

To handle an exception, you enclose the statements that could cause the exception in a try block, then you add catch blocks to handle one or more exceptions. In this example, in addition to handling the more specific FileNotFoundException exception, we are also using a catch block with more generic exceptions to catch all other exceptions. The exception name for a catch block must be enclosed within parentheses. The statements that are executed when an exception is caught must be enclosed within curly braces.

Code execution stops when an exception occurs. The runtime searches for a catch statement that matches the type of exception. If the first catch block doesn't catch the raised exception, control moves to the next catch block, and so on. If the exception is not handled in the method, the runtime checks for the catch statement in the calling code and continue for the rest of the call stack.

### TakeNote

*A try block must have at least a catch block or a finally block associated with it.*

### Using Try-Catch-Finally

The finally block is used in association with the try block. The finally block is always executed regardless of whether an exception is thrown. The finally block is often used to write clean-up code.

When an exception occurs, it often means that some lines of code after the exception were not executed. This may leave your program in a dirty or unstable state. To prevent such situations, you can use the finally statement to guarantee that certain cleanup code is always executed. This may involve closing connections, releasing resources, or setting variables to their expected values. Let's look at a finally block in the following exercise.

#### Illustration 11: USE TRY-CATCH-FINALLY

GET READY. To use the try-catch-finally statement, perform the following steps:

1. Add a new Console Application project named trycatchfinally to the Lesson01 solution.
2. Add the following code to the Main method of the Program.cs class:  
TryCatchFinallyTest(); // call to methodAdd the following method to the Program.cs class:

```
private static void TryCatchFinallyTest()
{
    StreamReader sr = null;
    try
    {
        sr = File.OpenText("data.txt");
        Console.WriteLine(sr.ReadToEnd());
    }
    catch (FileNotFoundException fnfe)
    {
        Console.WriteLine(fnfe.Message);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        if (sr != null)
        {
            sr.Close();
        }
    }
}
```

3. Create a text file ("data.txt") using Notepad or Visual Studio on the C: drive. It is acceptable to create the file at a different location, but if you do so, remember to modify the file location in the program. Enter some text in the file.
4. Select Debug > Start Without Debugging, or press Ctrl+ F5.
5. You will see the contents of the text file displayed in a command window.
6. Press a key to close the command window.
7. Delete the data.txt file and run the program again. This time, you'll get a FileNotFoundException exception, and an appropriate message will be displayed in the output window.

In this exercise, the program makes sure that the StreamReader object is closed and any resources are released when the operation completes. The code in the finally block is executed regardless of whether an exception is thrown.













