LESSON 1

Introduction to Programming

LESSON SKILL MATRIX

SKILLS/CONCEPTS	MTA EXAM OBJECTIVE	MTA EXAM OBJECTIVE NUMBER
Understanding Computer Programming	Understand computer storage and data types.	1.1
Understanding Decision Structures	Understand computer decision structures.	1.2
Understanding Repetition Structures	Identify the appropriate method for handling repetition.	1.3
Understanding Exception Handling	Understand error handling.	1.4

KEY TERMS

algorithm
array
binary code
binary number system
case
class
computer programs (programs)
constant
data types
decision structures
decision table
default statement
do-while loop
exception
finally block
flowchart
for loop
foreach loop
high-level language
if statement
if-else statement
methods
operator
recursion
switch block
switch statement
try-catch-finally block
variable
while loop

Imagine that you are a software developer for the Northwind Corporation. As part of your job, you develop computer programs to solve business problems. Examples of the work you do include analyzing customer orders to determine applicable discounts, updating stock information for thousands of items in a company's inventory, and writing interactive reports that allow users to sort and filter data.

It is important for you to make sure your programs are designed exactly according to specifications. You also need to ensure that all computations are accurate and complete. The programs that you write need to be robust, and they should be able to display error messages but continue processing.

The programming language that you use provides you with various tools and techniques to get your tasks done. Based on the task at hand, you select the data types and the control structures that are best suited for solving the problem.

Understanding Computer Programming

THE BOTTOM LINE

A computer program is a set of precise instructions to complete a task. In this section, you'll learn how to write algorithms and computer programs to solve a given problem. In addition to writing your first computer program using the C# programming language, you'll also learn about the basic structure of computer programs and how to compile, execute, provide input to, and generate output from a program.

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.

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#. You'll learn about that in the next section.

In the meantime, 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.

INTRODUCING FLOWCHARTS

A ***flowchart*** is a graphical representation of an algorithm. A flowchart is usually drawn using standard symbols. Some common flowchart symbols are shown in Table 1-1.

Table 1-1
Common flowchart symbols

Flowchart Symbol	Description
	Start or end of an algorithm
	A process or computational operation
	Input or output operation
	Decision-making operation
	Specifies the flow of control

For example, Figure 1-1 shows a flowchart that inputs two numbers, compares them, and then outputs the larger number.

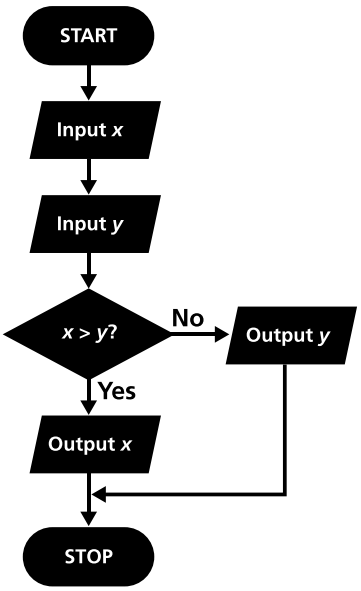


Figure 1-1

A simple flowchart that compares two numbers and outputs the larger of the two

As you can see, this flowchart lists in the correct order all the necessary steps to perform the operation. The flow of control starts with the start symbol and ends at the stop symbol. The process and input/output operation symbols always have a single entry and a single exit. In contrast, the decision symbol has a single entry but multiple exits. You can test a flowchart by performing a "dry run." In a dry run, you manually trace through the steps of the flowchart with test data to check whether the correct paths are being followed.

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. 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 bold lines in the decision table divide the table in 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
A decision table for calculating discounts

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%.

Introducing C#

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.

As discussed in the previous section, computers need precise and complete instructions to accomplish a task. These sets of instructions are called ***computer programs***, or just ***programs*** for short.

At the most basic level, computers use the ***binary number system*** to represent information and code. In this system, each value is represented using only two symbols, 0 and 1. A computer program written using the binary number system is called ***binary 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 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.

Most modern programs are written in a ***high-level language*** such as C#, Visual Basic, or Java. These languages allow you to write precise instructions in a human-readable form. A language compiler then translates the high-level language into a lower-level language that can be understood by the runtime execution system.

Each programming language provides its own set of vocabulary and grammar (also known as syntax). In this course, you'll learn how to program by using the C# programming language on the .NET Framework. 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.

MORE INFORMATION

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.

In this course, 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.

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)
        {
            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**.
6. You will see the output of the program in a command Window, as shown in Figure 1-2.

Figure 1-2
Program output in a command window

ANOTHER WAY

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.

7. Press a key to close the 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.
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.

TAKE NOTE*

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.

If you don't use an Integrated Development Environment (IDE) like Visual Studio, you can still compile your program manually using the command-line tools. Visual Studio, of course, makes it easier and quicker to test your programs.

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

In this section of the lesson, you'll learn about the structural elements of the simple C# program you created in the previous section.

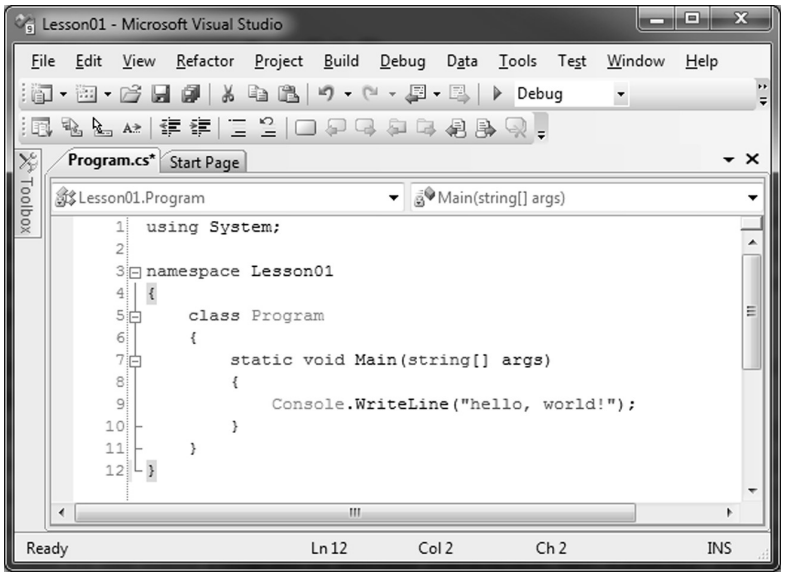
Figure 1-3 depicts the program you created in the previous exercise with line numbers. Throughout this section, these numbers will be used to refer to different structures in the program.

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.

Figure 1-3

Program listing with line numbers



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 1-3 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 (}).

XREF

You can find more information on classes in Lesson 2.

Line 3 of the code in Figure 1-3 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.

XREF

You can find more information on methods in Lesson 2.

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 1-3 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!");
```

TAKE NOTE*

Every C# statement must end with a semicolon (;).

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 1-3). 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.

TAKE NOTE*

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 following lesson.

UNDERSTANDING VARIABLES

Variables provide temporary storage during the execution of a program.

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.

TAKE NOTE*

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.

XREF

You can find more information about how to create your own data types in Lesson 2.

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 chapter focuses on some of the most commonly used built-in data types.

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. For a 64-bits operating system, such as Windows 7 64-bit, these sizes will be different.

Table 1-3

Commonly used built-in data types in C#

Data type	Size	Range of values
byte	1 byte	0 to 255
char	2 bytes	U+0000 to U+ffff (Unicode characters)
short	2 bytes	−32,768 to 32,767
int	4 bytes	−2,147,483,648 to 2,147,483,647
long	8 bytes	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	±1.5 × 10-45 to ±3.4 × 1038
double	8 bytes	±5.0e−324 to ±1.7e308
bool	2 bytes	True or false
string	-	Zero or more Unicode characters

TAKE NOTE*

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 1; 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].

XREF
The topic of arrays is covered in more detail in Lesson 3, Understanding General Software Development.

UNDERSTANDING OPERATORS

Operators are symbols that specify which operation to perform on the operands before returning a result.

Examples of operators include +, -, *, /, and so on, and operands can be variables, constants, literals, etc. Depending on how many operands are involved, there are three kinds of operators:

- **Unary operators:** The unary operators work with only one operand. Examples include ++x, x++, or isEven, where x is of integer data type and isEven is of Boolean data type.
- **Binary operators:** The binary operators take two operands. Examples include x + y or x > y.
- **Ternary operators:** Ternary operators take three operands. There is just one ternary operator, ?:, in C#.

Often, expressions involve more than one operator. In this case, the compiler needs to determine which operator takes precedence over the other(s). Table 1-4 lists the C# operators in order of precedence. The higher an operator is located in the table, the higher its precedence. Operators with higher precedence are evaluated before operators with lower precedence. Operators that appear in the same row have equal precedence.

Table 1-4
Operator precedence in C#

CATEGORY	OPERATORS
Primary	x.y f(x) a[x] x++ x -- new typeof checked unchecked
Unary	+ - ! ~ ++x -- x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational and type testing	< > <= >= is as
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional ternary	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

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
```

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. Methods can receive input via arguments and can return a value to the caller.

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.

CERTIFICATION READY
Do you understand the core elements of programming, such as variables, data types, operators, and methods?
1.1

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.

Class members can have modifiers such as static, public, and private. These modifiers specify how and where class members can be accessed. You'll learn more about these modifiers in Lesson 2.

■ Understanding Decision Structures

THE BOTTOM LINE

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.

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

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.

USE THE 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**.
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.

This code is functionally equivalent to the flowchart shown in Figure 1-4.

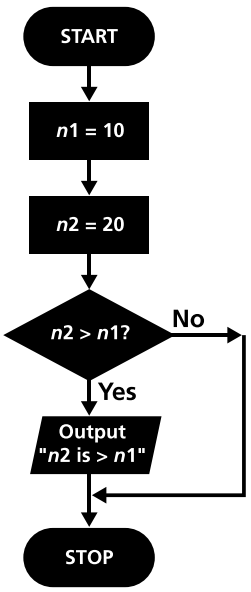


Figure 1-4
The flowchart equivalent of the example if statement

Here, the output statement will be executed only if the Boolean expression in the parentheses is true. If the expression is false, control passes to the next statement following the if statement.

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

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.

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

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);
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 greater than or equal to 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. 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.

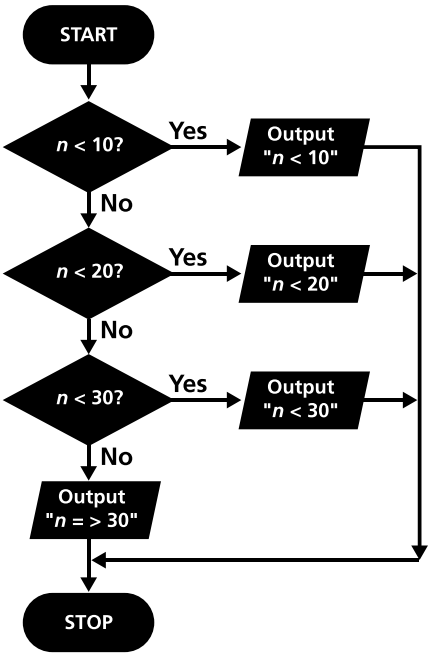
Here, the code in the TestIfElse method combines several if-else statements to test for multiple conditions. For example, if the value of n is 25, then the first two conditions (n < 10 and n < 20) will evaluate to false, but the third condition (n < 30) will evaluate to true. As a result, the method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

```
n is less than 30
```

This C# program is equivalent to the flowchart shown in Figure 1-5.

Figure 1-5

The flowchart equivalent of the example if-else statement



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.

TAKE NOTE*

The expression following the case statement must be a constant expression and must be of the matching data type to the switch expression.

The switch statement consists of the keyword switch, followed by an expression in parentheses, followed by a switch block. The ***switch block*** can include one or more ***case*** statements or a ***default*** statement. When the switch statement executes, depending on the value of the switch expression, control is transferred to a matching case statement. If the expression does not match any of the case statements, then control is transferred to the default statement. The switch expression must be surrounded by parentheses.

Take the following steps to create a program that uses the switch statement to evaluate simple expressions.

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.

TAKE NOTE*

The Console.Write and the Console.WriteLine methods can use format strings such as "Results: {0}" to format the output. 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;
    }
}
```

CERTIFICATION READY
Do you understand computer decision structures, such as branching and repetition?
1.2

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.

TAKE NOTE*

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.

■ Understanding Repetition Structures

THE BOTTOM LINE

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.

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.

Understanding the While Loop

The ***while loop*** repeatedly executes a block of statements until a specified Boolean expression evaluates to false.

The general form of the while loop is as follows:

```
while (boolean test)
    statement
```

TAKE NOTE*

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.

Here, a Boolean test is performed at the beginning of the loop. If the test evaluates to true, the loop body is executed and the test is performed again. If the test evaluates to false, the loop terminates and control is transferred to the next statement following the loop.

Because the Boolean test is performed before the execution of the loop, it is possible that the body of a while loop is never executed. This happens if the test evaluates to false the first time.

Take the following steps to create a program that uses the while loop.

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.

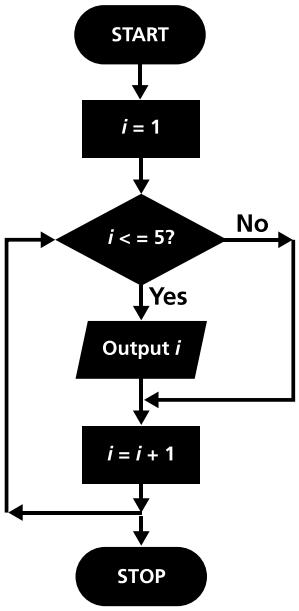
In this exercise, the variable i is assigned the value 1. Next, the condition in the while loop is evaluated. Because the condition is true (1 <= 5), the code within the while statement block is executed. The value of i is written in the command window, and then the value of i is increased by 1 so that it becomes 2. Control then passes back to the while statement, and the condition is evaluated again. Because the condition is still true (2 <= 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 <= 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 flowchart equivalent of this while loop is shown in Figure 1-6.

Figure 1-6

The flowchart equivalent of the example while loop



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.
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 in achieved. In the above example, the expression i++ is the termination expression.

TAKE NOTE*

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.

TAKE NOTE*

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

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.

The general form of the do-while loop is as follows:

```
do
      statement
while (boolean test);
```

Take the following steps to create a program that uses the do-while loop.

USE THE DO-WHILE LOOP

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 <= 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 <= 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.

TAKE NOTE*

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.

The for loop is similar to the while loop; it allows a statement or a statement block to be executed repeatedly until an expression evaluates to false. The general form of the for loop is as follows:

```
for (init-expr; cond-expr; count-expr)
    statement
```

As you can see, the for loop combines the three essential control expressions of an iteration. This results in more readable code. The for loop is especially useful for creating iterations that must execute a specified number of times.

Take the following steps to create a program that uses the for loop.

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*** is useful for iterating through the elements of a collection.

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.

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:

```
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:

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

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: 3! = 3 * 2! → 3 * 2 * 1! → 3 * 2 * 1 * 0! → 3 * 2 * 1 * 1 → 6.

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

USE THE RECURSIVE METHOD

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:
   ```
   Factorial(5);
   ```
3. Add the following method to the Program.cs class:
   ```
   public static int Factorial(int n)
   {
       if (n == 0)
       {
           return 1; //base case
       }
       else
       {
           return n * Factorial(n - 1); //recursive case
       }
   }
   ```
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 to pass a different value to the Factorial method, and note the results.

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

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

- **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.
- **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.

CERTIFICATION READY
Can you identify the appropriate methods for handling repetition?
1.3

Understanding Exception Handling

THE BOTTOM LINE

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.

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)
       {
           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.

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

TAKE NOTE*

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.

TAKE NOTE*

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.

TAKE NOTE*

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

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 continues for the rest of the call stack.

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.

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();
```

3. Add 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();
            }
        }
}
```

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.

CERTIFICATION READY
Do you understand how to handle errors in your programs?
1.4

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.

SKILL SUMMARY

IN THIS LESSON, YOU LEARNED THE FOLLOWING:

- An algorithm is a set of ordered and finite steps to solve a given problem. You may find it useful to express an algorithm as a flowchart or a decision table before you develop a formal computer program.
- The C# programming language is a part of the .NET Framework and benefits from the runtime support and class libraries provided by the .NET Framework.
- Main is a special method because it also serves as an entry point to a program. When the runtime executes a program, it always starts at the Main method.
- 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 value it can contain and what kind of operations may be performed on it.

- Operators are symbols, such as +, -, *, and /, that specify which operation to perform on one or more operands before returning a result.
- If-else statements allow a program to perform one action if the Boolean expression evaluates to true and a different action if the Boolean expression evaluates to false.
- The switch statement allows multi-way branching. In many cases, using a switch statement can simplify a complex combination of if-else statements.
- 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.
- The while and do-while loops repeatedly execute 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 for loop combines the three elements of iteration—the initialization statement, the termination condition, and the increment/decrement statement—into more readable code.
- The foreach loop is useful for iterating through the elements of a collection.
- Recursion is a programming technique that causes a method to call itself in order to compute a result.
- The .NET Framework supports standard exception handling to raise and handle runtime errors. To handle exceptions, place the code that throws exceptions inside a try block, and place the code that handles the exceptions inside a catch block.
- 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.

■ Knowledge Assessment

Fill in the Blank

Complete the following sentences by writing the correct word or words in the blanks provided.

1. The _____ statement selects for execution a statement list having an associated label that corresponds to the value of an expression.
2. The _____ loop tests the condition at the bottom of the loop instead of at the top.
3. The only operator that takes three arguments is the _____ operator.
4. The _____ loop is the most compact way to iterate through the items in a collection.
5. On a 32-bit computer, a variable of int data type takes _____ bytes of memory.
6. To access the first element of an array, you use an index of _____.
7. _____ is a programming technique that causes a method to call itself in order to compute a result.
8. _____ are data fields or local variables whose value cannot be modified.
9. When an algorithm involves a large number of conditions, a(n) _____ is a compact and readable format for presenting the algorithm.
10. A(n) _____ is a graphical representation of an algorithm.

Multiple Choice

Circle the letter that corresponds to the best answer.

1. Write the following code snippet:

```
int n = 20;
int d = n++ + 5;
```

What will be the value of d after this code snippet is executed?
a. 25
b. 26
c. 27
d. 28

2. Write the following code snippet:

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

How many times will the while loop be executed in this code snippet?
a. 0
b. 1
c. 4
d. 5

3. Write the following code snippet:

```
int number1 = 10;
int number2 = 20;
if (number2 > number1)
Console.WriteLine("number1");
Console.WriteLine("number2");
```

What output will be displayed after this code snippet is executed?
a. number1
b. number2
c. number1
number2
d. number2
number1

4. In a switch statement, if none of the case statements match the switch expression, then control is transferred to which statement?
a. break
b. continue
c. default
d. return

5. You need to write code that closes a connection to a database, and you need to make sure this code is always executed regardless of whether an exception is thrown. Where should you write this code?
a. Within a try block
b. Within a catch block

c. Within a finally block
d. Within the Main method

6. You need to store values ranging from 0 to 255. You also need to make sure that your program minimizes memory use. Which data type should you use to store these values?
   a. byte
   b. char
   c. short
   d. int

7. If you don't have a base case in your recursive algorithm, you create an infinite recursion. An infinite recursion will cause your program to throw an exception. Which exception will your program throw in such a case?
   a. OutOfMemoryException
   b. StackOverflowException
   c. DivideByZeroException
   d. InvalidOperationException

8. You are learning how to develop repetitive algorithms in C#. You write the following method:

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

   How many repetitions will the for loop in this code perform?
   a. 0
   b. 4
   c. 5
   d. Infinite repetitions

9. Which of the following C# features should you use to organize code and create globally unique types?
   a. Assembly
   b. Namespace
   c. Class
   d. Data type

10. You write the following code snippet:

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

    You also create a variable of the RectangleHandler type like this:

```
RectangleHandler handler;
```

    What is the value of the variable val after this code snippet is executed?
    a. 1
    b. 2
    c. 3
    d. 4

■ Competency Assessment

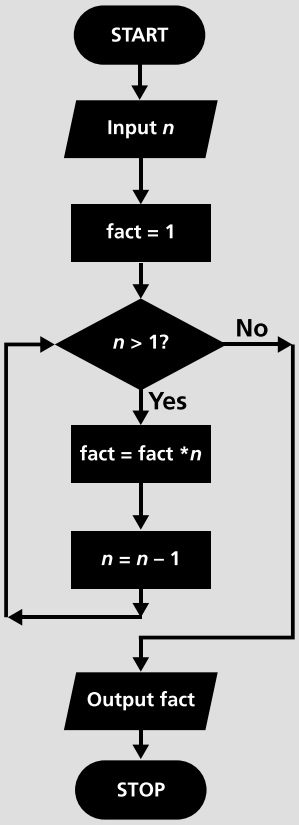
Scenario 1-1: Converting a Decision Table into a C# Program

You are developing an invoicing application that calculates discount percentages based on the quantity of a product purchased. The logic for calculating discounts is listed in the following decision table. If you need to write a C# method that uses the same logic to calculate the discount, how would you write such a program?

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

Scenario 1-2: Converting a Flowchart into a C# Program

You are developing a library of mathematical functions. You first develop the following flowchart describing the algorithm for calculating the factorial of a number. You need to write an equivalent C# program for this flowchart. How would you write such a program?



■ Proficiency Assessment

Scenario 1-3: Handling Exceptions

You are writing code for a simple arithmetic library. You decide to create a method named Divide that takes two arguments, x and y, and returns the value of x/y. You need to catch any arithmetic exceptions that might be thrown for errors in arithmetic, casting, or data type conversions. You also need to catch any other exceptions that may be thrown from the code. To address this requirement, you need to create properly structured exception-handling code. How would you write such a program?

Scenario 1-4: Creating a Recursive Algorithm

You are developing a library of utility functions for your application. You need to write a method that takes an integer and counts the number of significant digits in it. You need to create a recursive program to solve this problem. How would you write such a program?