C# BASIC TUTORIAL (1)

GVC Ths. Nguyễn Minh Đạo

BM CNPM - Khoa CNTT - ĐHSPKT TP.HCM

OBJECTIVES

- C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft within its .NET initiative led by Anders Hejlsberg.
- This tutorial will teach you basic C# programming and will also take you through various advanced concepts related to C# programming language.

C# - OVERVIEW

C# was developed by Anders Hejlsberg and his team during the development of .NET Framework.

C# is a modern, general-purpose, object-oriented programming language developed by Microsoft and approved by Ecma and ISO.

OVERVIEW

- C# is a modern, general-purpose, object-oriented programming language developed by Microsoft and approved by Ecma and ISO.
- C# was developed by Anders Hejlsberg and his team during the development of .NET Framework.
- C# is designed for Common Language Infrastructure (CLI), which consists of the executable code and runtime environment that allows use of various high-level languages to be used on different computer platforms and architectures.

THE CHARACTERISTICS OF C#

- The following reasons make C# a widely used professional language:
  - Modern, general-purpose programming language
  - Object oriented.
  - Component oriented.
  - Easy to learn.
  - Structured language.
  - It produces efficient programs.
  - It can be compiled on a variety of computer platforms.
  - Part of .NET Framework.

STRONG PROGRAMMING FEATURES OF C#

- Although C# constructs closely follow traditional high-level languages C and C++ and being an object-oriented programming language, it has strong resemblance with Java, it has numerous strong programming features that make it endearing to multitude of programmers worldwide.

STRONG PROGRAMMING FEATURES OF C#

- Following is the list of few important features:
  - Boolean Conditions
  - Automatic Garbage Collection
  - Standard Library
  - Assembly Versioning
  - Properties and Events
  - Delegates and Events Management

STRONG PROGRAMMING FEATURES OF C#

- Following is the list of few important features:
  - Easy-to-use Generics
  - Indexers
  - Conditional Compilation
  - Simple Multithreading
  - LINQ and Lambda Expressions
  - Integration with Windows

STRONG PROGRAMMING FEATURES OF C#

- Following is the list of few important features:
  - Easy-to-use Generics
  - Indexers
  - Conditional Compilation
  - Simple Multithreading
  - LINQ and Lambda Expressions
  - Integration with Windows

C# - ENVIRONMENT

In this chapter, we will discuss the tools required for creating C# programming. We have already mentioned that C# is part of .NET framework and is used for writing .NET applications. Therefore, before discussing the available tools for running a C# program, let us understand how C# relates to the .NET framework.

THE .NET FRAMEWORK

- The .NET framework is a revolutionary platform that helps you to write the following types of applications:
  - Windows applications
  - Web applications
  - Web services
- The .NET framework applications are multi-platform applications. The framework has been designed in such a way that it can be used from any of the following languages: C#, C++, Visual Basic, Jscript, COBOL, etc. All these languages can access the framework as well as communicate with each other.

THE .NET FRAMEWORK

- The .NET framework consists of an enormous library of codes used by the client languages like C#. Following are some of the components of the .NET framework:
  - Common Language Runtime (CLR)
  - The .NET Framework Class Library
  - Common Language Specification
  - Common Type System
  - Metadata and Assemblies
  - Windows Forms

THE .NET FRAMEWORK

- The .NET framework consists of an enormous library of codes used by the client languages like C#. Following are some of the components of the .NET framework:
  - ASP.NET and ASP.NET AJAX
  - ADO.NET
  - Windows Workflow Foundation (WF)
  - Windows Presentation Foundation
  - Windows Communication Foundation (WCF)
  - LINQ

INTEGRATED DEVELOPMENT ENVIRONMENT (IDE) FOR C#

- Microsoft provides the following development tools for C# programming:
  - Visual Studio 2010 (VS)
  - Visual C# 2010 Express (VCE)
  - Visual Web Developer
- The last two are freely available from Microsoft official website. Using these tools, you can write all kinds of C# programs from simple command-line applications to more complex applications.

INTEGRATED DEVELOPMENT ENVIRONMENT (IDE) FOR C#

- You can also write C# source code files using a basic text editor, like Notepad, and compile the code into assemblies using the command-line compiler, which is again a part of the .NET Framework.
- Visual C# Express and Visual Web Developer Express edition are trimmed down versions of Visual Studio and has the same look and feel. They retain most features of Visual Studio. In this tutorial, we have used Visual C# 2010 Express.
- You can download it from Microsoft Visual Studio. It gets automatically installed in your machine. Please note that you need an active internet connection for installing the express edition.

C# - PROGRAM STRUCTURE

In this chapter, we will discuss the structure of C# program.

PARTS OF C# PROGRAM

- A C# program basically consists of the following parts:
  - Namespace declaration
  - A class
  - Class methods
  - Class attributes
  - A Main method
  - Statements & Expressions
  - Comments

C# HELLO WORLD EXAMPLE

- Let us look at a simple code that would print the words "Hello World":

```
using System;
namespace HelloWorldApplication
{
   class HelloWorld
   {
      static void Main(string[] args)
      {
         /* my first program in C# */
         Console.WriteLine("Hello World");
         Console.ReadKey();
      }
   }
}
```

- When the above code is compiled and executed, it produces the following result:

```
Hello World
```

C# HELLO WORLD PROGRAM STRUCTURE

- Let us look at various parts of the above program:
  - The first line of the program **using System;** - the **using** keyword is used to include the **System** namespace in the program. A program generally has multiple **using** statements.
  - The next line has the **namespace** declaration. A **namespace** is a collection of classes. **The*HelloWorldApplication*** namespace contains the class ***HelloWorld***.
  - The next line has a **class** declaration, the class ***HelloWorld*** contains the data and method definitions that your program uses. Classes generally would contain more than one method. Methods define the behavior of the class. However, the ***HelloWorld*** class has only one method **Main**.

C# HELLO WORLD PROGRAM STRUCTURE

- Let us look at various parts of the above program:
  - The next line /*...*/ will be ignored by the compiler and it has been put to add additional **comments** in the program.
  - The Main method specifies its behavior with the statement **Console.WriteLine("Hello World");**

  WriteLine is a method of the *Console* class defined in the *System* namespace. This statement causes the message "Hello, World!" to be displayed on the screen.
  - The last line **Console.ReadKey();** is for the VS.NET Users. This makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

C# PROGRAM STRUCTURE

- It's worth to note the following points:
  - C# is case sensitive.
  - All statements and expression must end with a semicolon (;).
  - The program execution starts at the **Main** method.
  - Unlike Java, file name could be different from the class name.

C# - BASIC SYNTAX

C# is an object-oriented programming language. In Object-Oriented Programming methodology, a program consists of various objects that interact with each other by means of actions. The actions that an object may take are called methods. Objects of the same kind are said to have the same type or, more often, are said to be in the same class.

C# - BASIC SYNTAX

- For example, let us consider a Rectangle object. It has attributes like length and width. Depending upon the design, it may need ways for accepting the values of these attributes, calculating area and display details.
- Let us look at an implementation of a Rectangle class and discuss C# basic syntax, on the basis of our observations in it.

C# RECTANGLE EXAMPLE

```
using System;
namespace RectangleApplication
{
    class Rectangle
    {
        // member variables
        double length;
        double width;
        public void Acceptdetails()
        {
            length = 4.5;
            width = 3.5;
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }
```

```
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Length: 4.5
Width: 3.5
Area: 15.75
```

THE *USING* KEYWORD

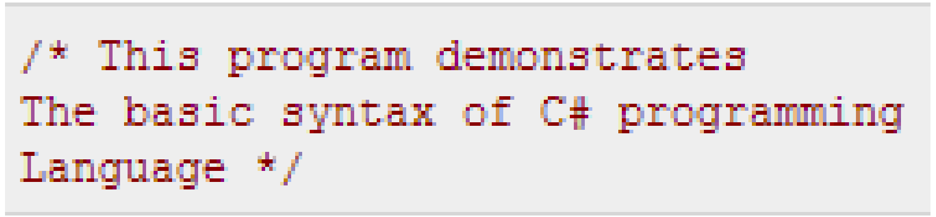
- The first statement in any C# program is
  - **using System;**
- The using keyword is used for including the namespaces in the program. A program can include multiple using statements.

THE *CLASS* KEYWORD

- The **class** keyword is used for declaring a class.
- A class is used in object-oriented programming to describe one or more objects. It serves as a template for creating, or instantiating, specific objects within a program. While each object is created from a single class, one class can be used to instantiate multiple objects.

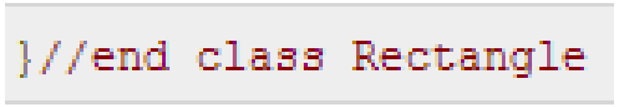
COMMENTS IN C#

- **Comments** are used for explaining code. Compilers ignore the comment entries.
- The multiline comments in C# programs start with /* and terminates with the characters */ as shown below:



```
/* This program demonstrates
The basic syntax of C# programming
Language */
```

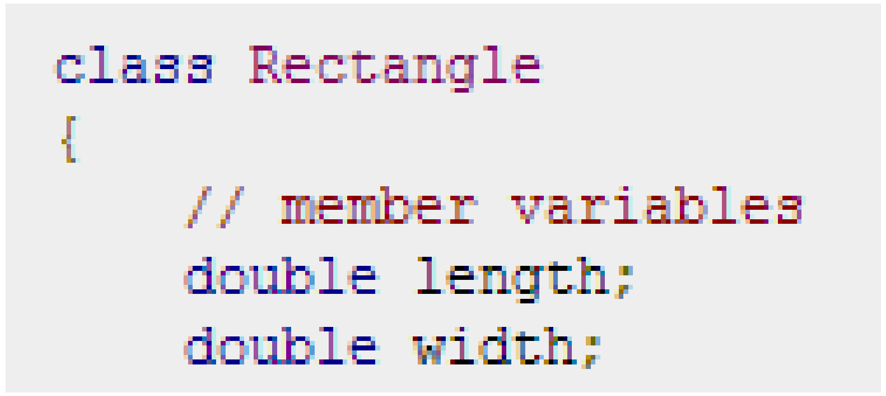
- Single-line comments are indicated by the '//' symbol. For example,



```
}//end class Rectangle
```

MEMBER VARIABLES

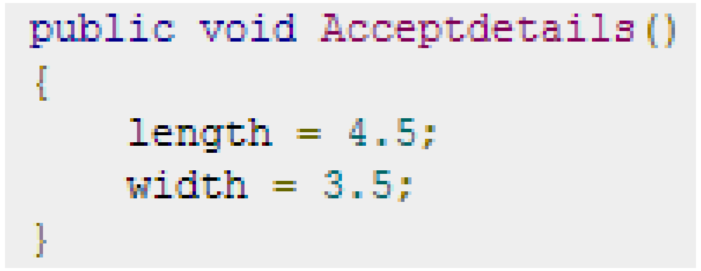
- **Variables** are attributes or data members of a class, used for storing data.
- In the preceding program, the ***Rectangle*** class has two member variables named ***length*** and ***width***.



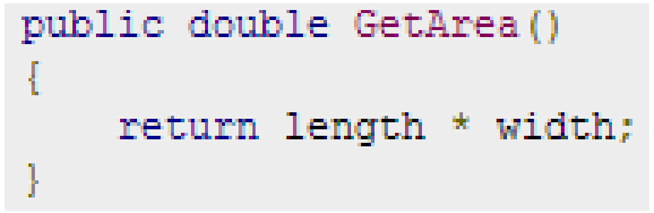
```
class Rectangle
{
    // member variables
    double length;
    double width;
```

MEMBER FUNCTIONS

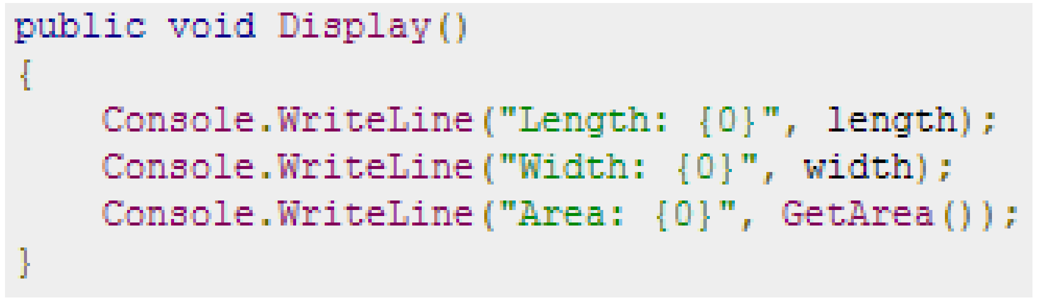
- **Functions** are set of statements that perform a specific task. The member functions of a class are declared within the class.
- Our sample class **Rectangle** contains three member functions: ***AcceptDetails***, ***GetArea*** and ***Display***.



```
public void Acceptdetails()
{
    length = 4.5;
    width = 3.5;
}
```



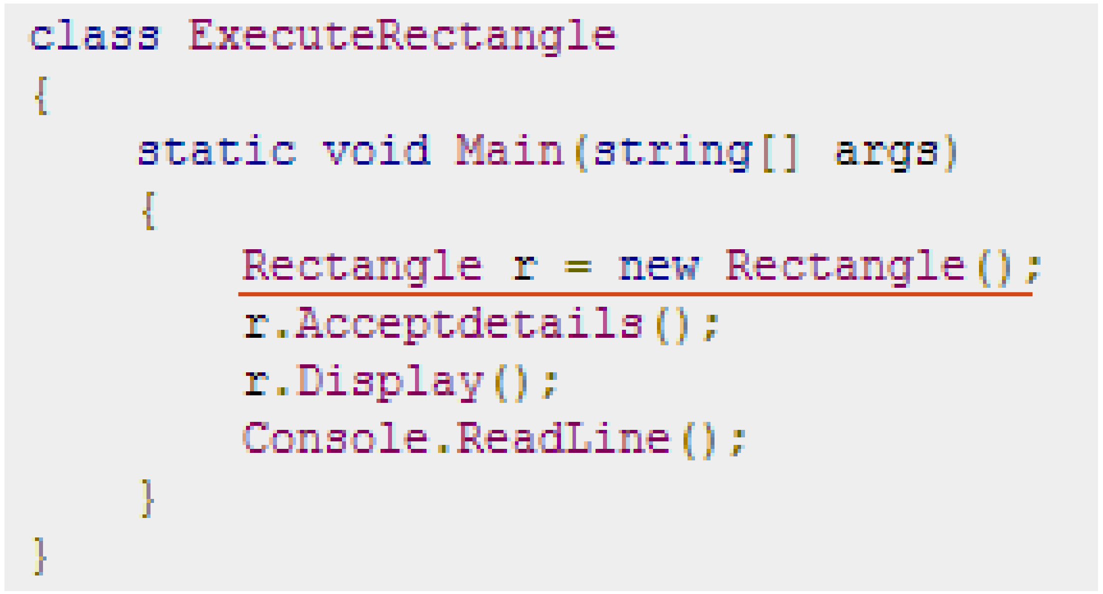
```
public double GetArea()
{
    return length * width;
}
```



```
public void Display()
{
    Console.WriteLine("Length: {0}", length);
    Console.WriteLine("Width: {0}", width);
    Console.WriteLine("Area: {0}", GetArea());
}
```

INSTANTIATING A CLASS

- In the preceding program, the class ***ExecuteRectangle*** is used as a class, which contains the ***Main*()** method and instantiates the ***Rectangle*** class.



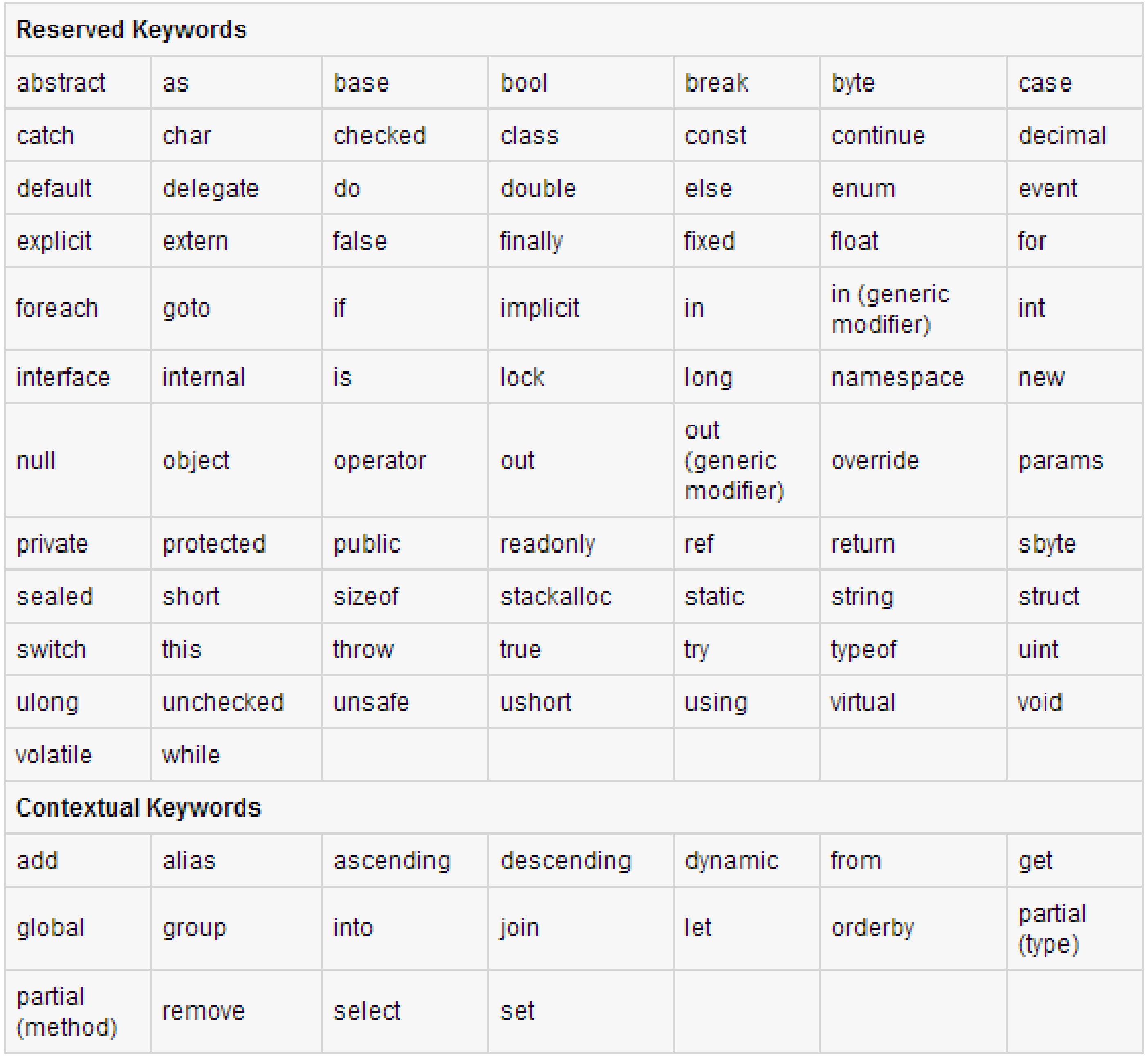
```
class ExecuteRectangle
{
    static void Main(string[] args)
    {
        Rectangle r = new Rectangle();
        r.Acceptdetails();
        r.Display();
        Console.ReadLine();
    }
}
```

IDENTIFIERS

- An **identifier** is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows:
  - A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore. The first character in an identifier cannot be a digit.
  - It must not contain any embedded space or symbol like ? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / and \. However, an underscore (_) can be used.
  - It should not be a C# keyword.

C# KEYWORDS

- **Keywords** are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers; however, if you want to use these keywords as identifiers, you may prefix the keyword with the @ character.
- In C#, some identifiers have special meaning in context of code, such as get and set, these are called contextual keywords.



Reserved Keywords						
abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	in (generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out (generic modifier)	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct
switch	this	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	while					
Contextual Keywords						
add	alias	ascending	descending	dynamic	from	get
global	group	into	join	let	orderby	partial (type)
partial (method)	remove	select	set			

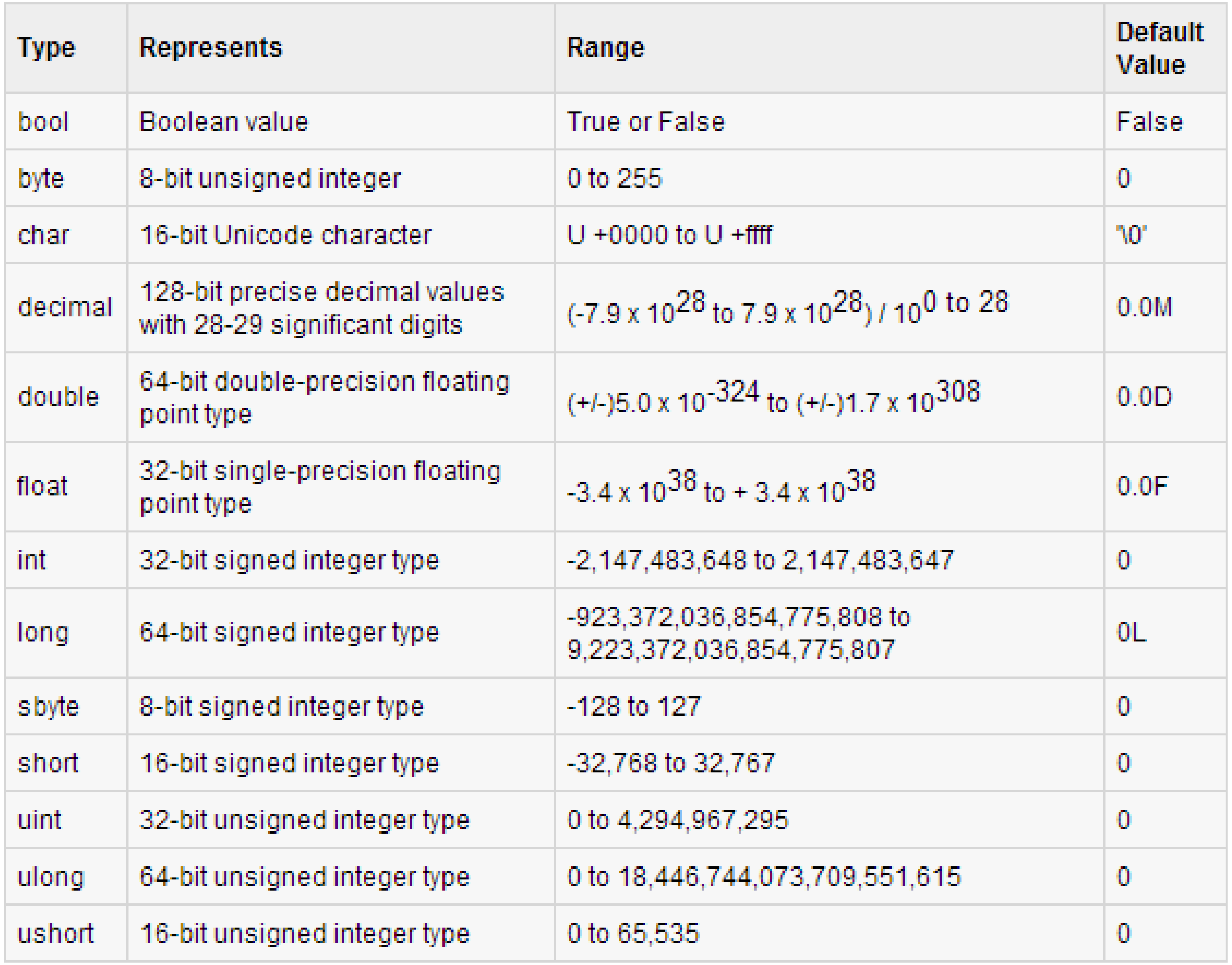
C# - DATA TYPES

In C#, variables are categorized into the following types: Value types, Reference types, Pointer types.

VALUE TYPES

- **Value type variables can be assigned a value directly.**
- They are derived from the class **System.ValueType**.
- The value types directly contain data. Some examples are **int, char, float**, which stores numbers, alphabets, floating point numbers, respectively. When you declare an **int** type, the system allocates memory to store the value.

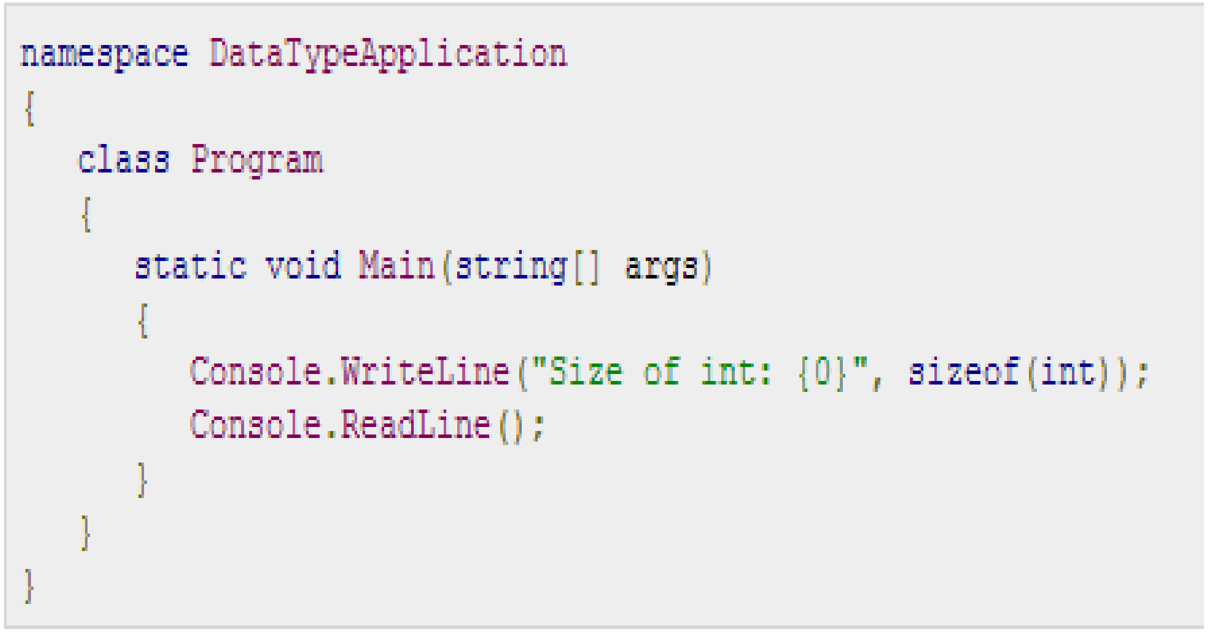
The following table lists the available value types in C# 2010



Type	Represents	Range	Default Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	(-7.9 x 10^{28} to 7.9 x 10^{28}) / 10^{0} to 28	0.0M
double	64-bit double-precision floating point type	(+/-)5.0 x 10^{-324} to (+/-)1.7 x 10^{308}	0.0D
float	32-bit single-precision floating point type	-3.4 x 10^{38} to + 3.4 x 10^{38}	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-923,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0
short	16-bit signed integer type	-32,768 to 32,767	0
uint	32-bit unsigned integer type	0 to 4,294,967,295	0
ulong	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit unsigned integer type	0 to 65,535	0

VALUE TYPES EXAMPLE

- To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** method. The expression ***sizeof(type)*** yields the storage size of the object or type in bytes.
- Following is an example to get the size of ***int*** type on any machine:



```
namespace DataTypeApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Size of int: {0}", sizeof(int));
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

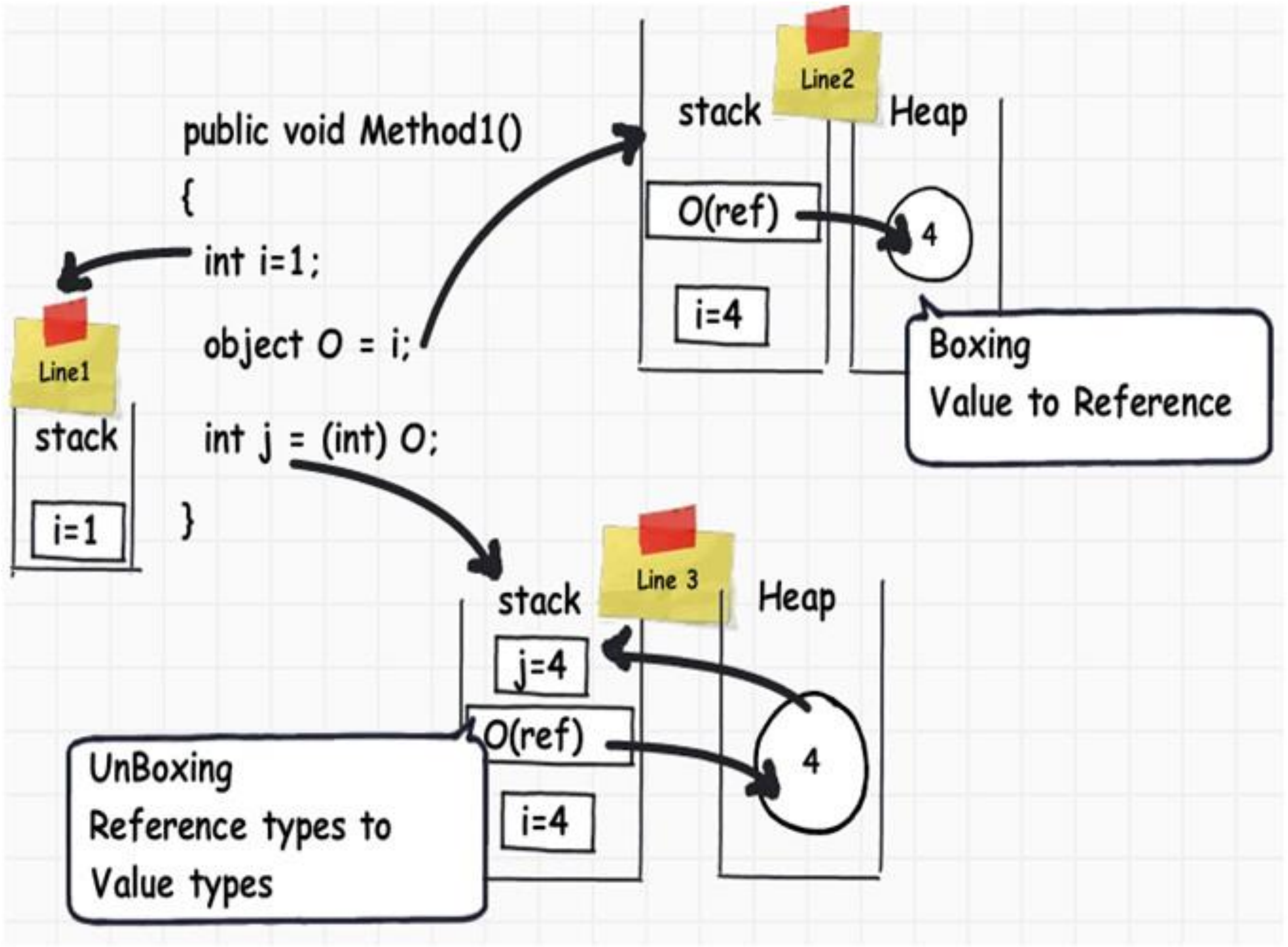
```
Size of int: 4
```

REFERENCE TYPES

- **The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.**
- In other words, they refer to a memory location. Using more than one variable, the reference types can refer to a memory location.
- If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of **built-in** reference types are: **object**, **dynamic** and **string**.

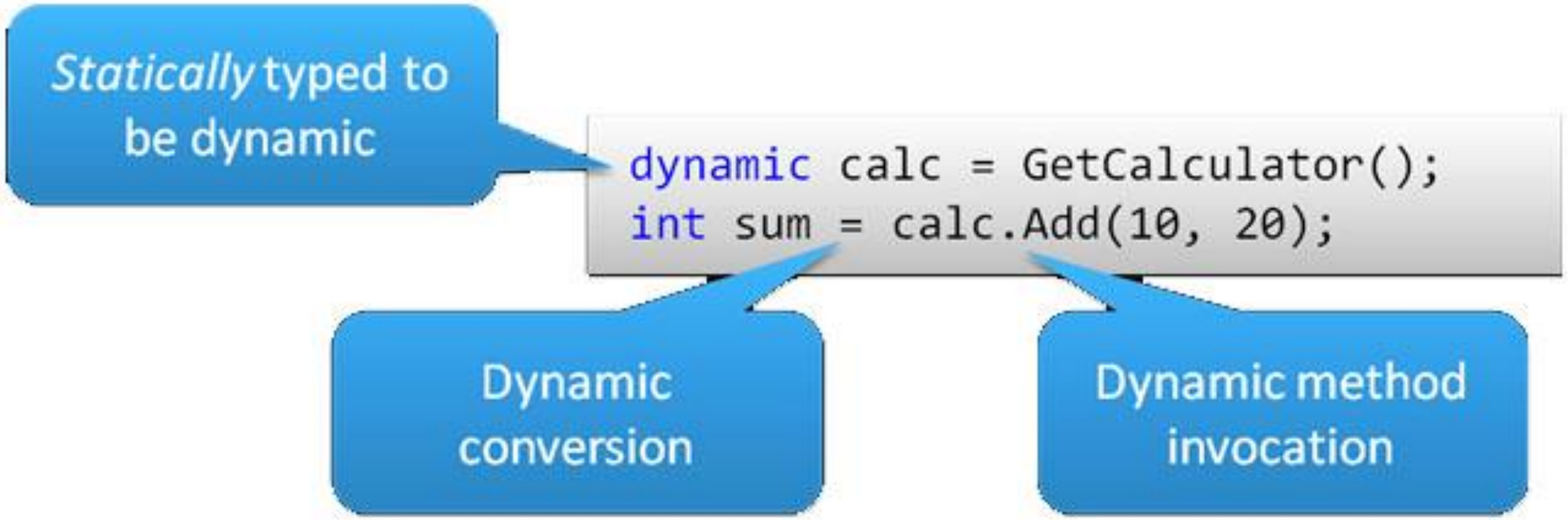
REFERENCE TYPES: OBJECT TYPE

- The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for **System.Object** class.
- So object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.
- When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.


public void Method1()
{
int i=1;
object O = i;
int j = (int) O;
}
Line1
stack
i=1
Line2
stack
Heap
O(ref)
4
i=4
Boxing
Value to Reference
Line 3
stack
Heap
j=4
O(ref)
4
i=4
UnBoxing
Reference types to
Value types

REFERENCE TYPES: DYNAMIC TYPE

- You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.
- Syntax for declaring a dynamic type is:
  - **dynamic <variable_name> = value;**
- For example,
  - dynamic d = 20;
- **Dynamic types** are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables take place at run time.



```
dynamic calc = GetCalculator();
int sum = calc.Add(10, 20);
```

REFERENCE TYPES: STRING TYPE

- The String Type allows you to assign any string values to a variable. The string type is an alias for the **System.String** class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.
- For example,
  - String str = "Tutorials Point";
- A @quoted string literal looks like:
  - @"Tutorials Point";
- The **user-defined reference types** are: **class**, **interface**, or **delegate**. We will discuss these types in later chapter.

REFERENCE TYPES: POINTER TYPES

- Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as in C or C++.
- Syntax for declaring a pointer type is:
  - **type* identifier;**
- For example,
  - char* cptr;
  - int* iptr;
- We will discuss pointer types in the chapter 'Unsafe Codes'.

C# - TYPE CONVERSION

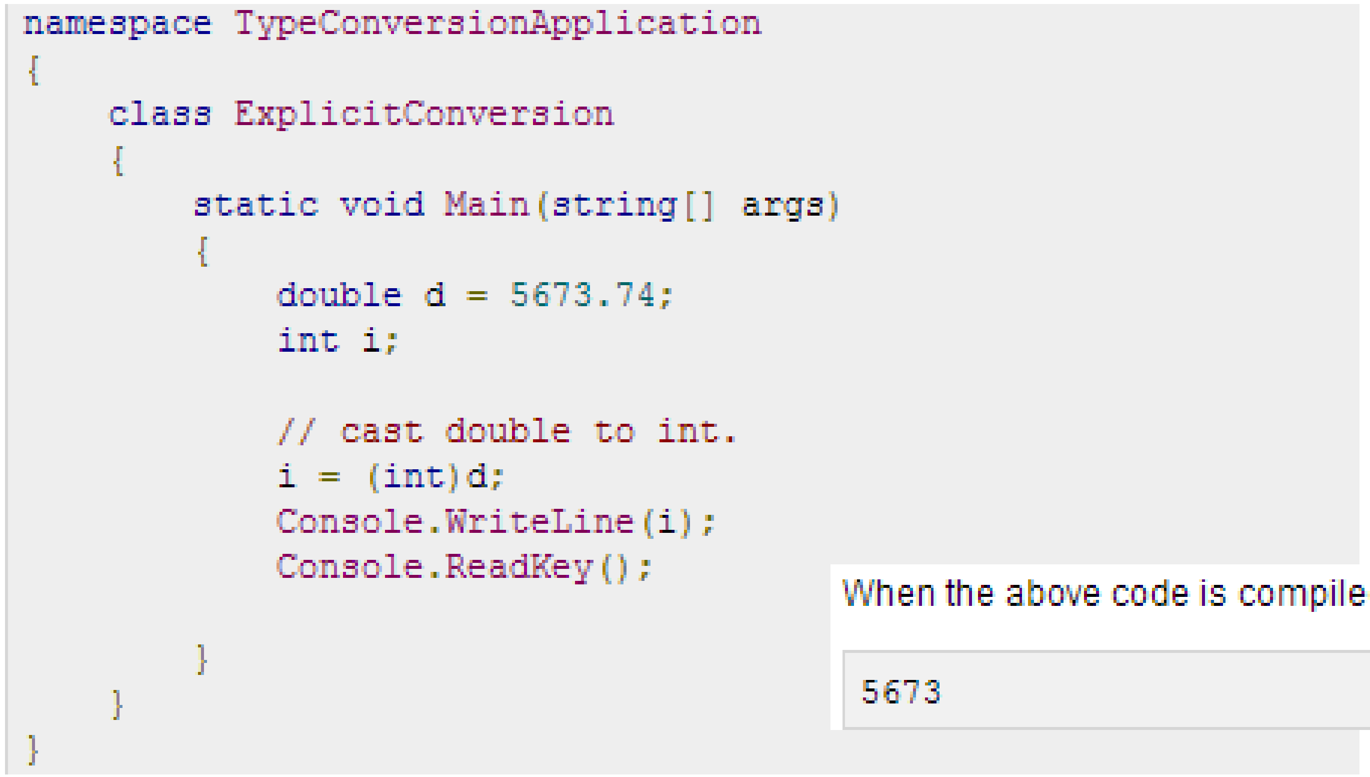
Type conversion is basically type casting or converting one type of data to another type

C# - TYPE CONVERSION

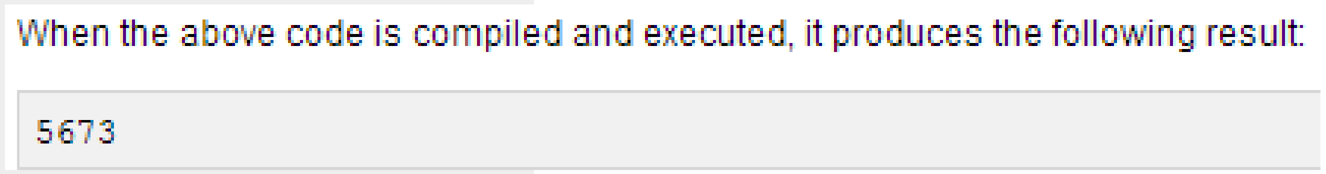
- Type conversion is basically type casting or converting one type of data to another type. In C#, type casting has two forms:
  - **Implicit type conversion** - these conversions are performed by C# in a type-safe manner. Examples are conversions from smaller to larger integral types and conversions from derived classes to base classes.
  - **Explicit type conversion** - these conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

C# - TYPE CONVERSION

- The following example shows an **explicit type** conversion:



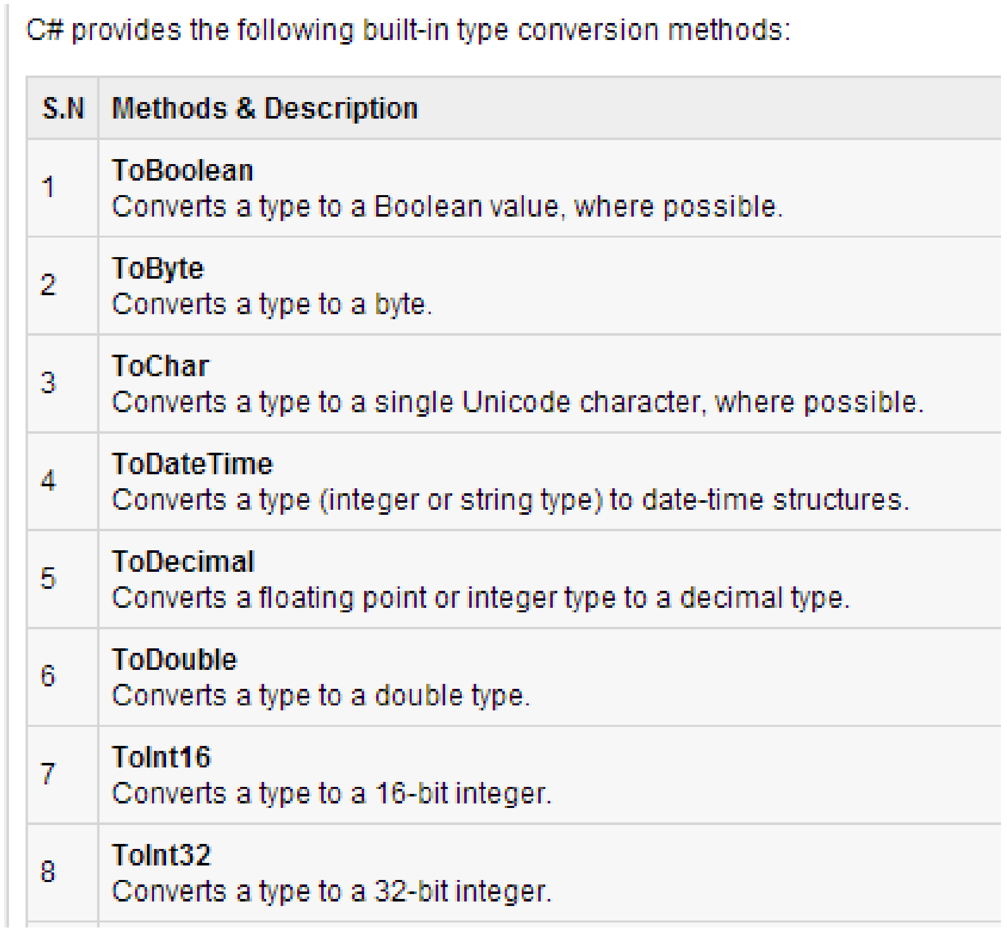
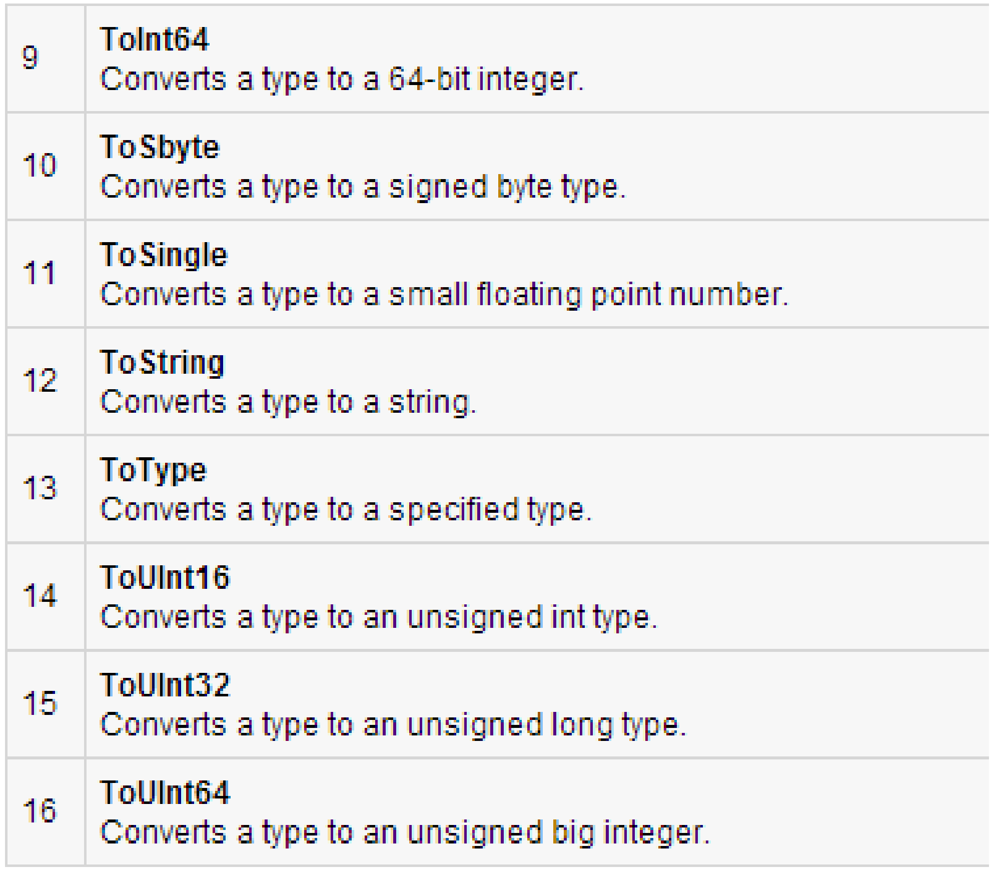
```
namespace TypeConversionApplication
{
    class ExplicitConversion
    {
        static void Main(string[] args)
        {
            double d = 5673.74;
            int i;

            // cast double to int.
            i = (int)d;
            Console.WriteLine(i);
            Console.ReadKey();

        }
    }
}
```



When the above code is compiled and executed, it produces the following result:

```
5673
```

C# TYPE CONVERSION METHODS

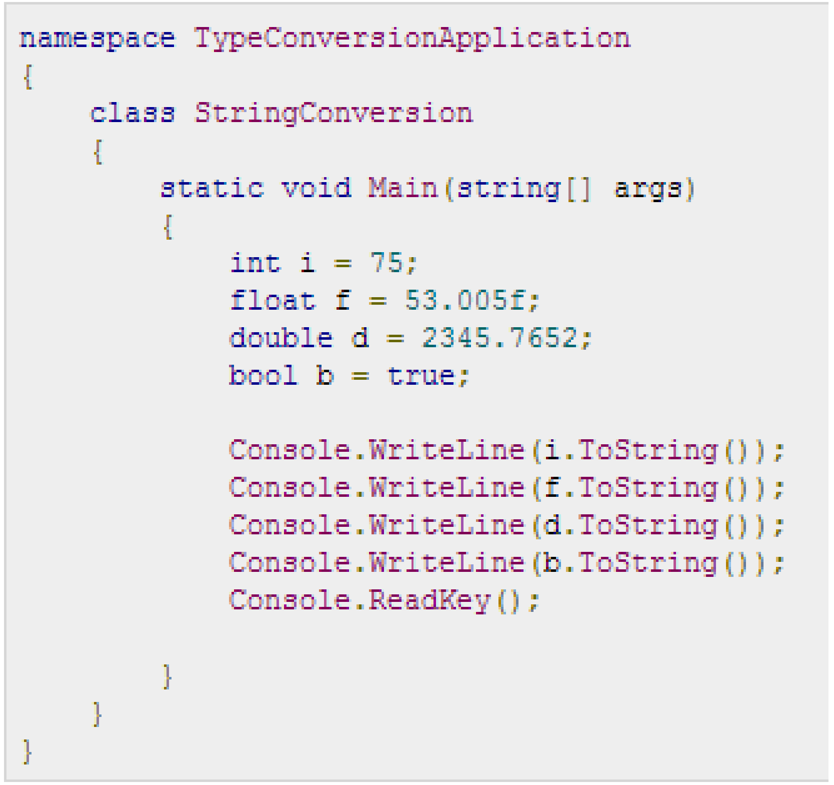



C# provides the following built-in type conversion methods:

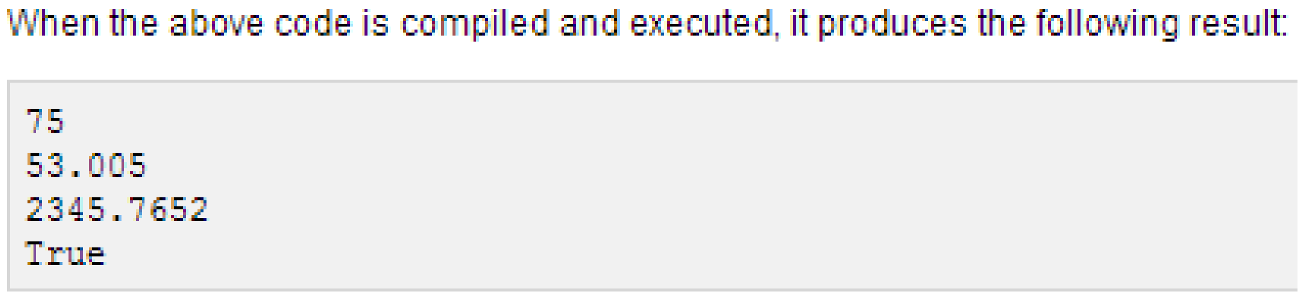
S.N	Methods & Description
1	**ToBoolean** Converts a type to a Boolean value, where possible.
2	**ToByte** Converts a type to a byte.
3	**ToChar** Converts a type to a single Unicode character, where possible.
4	**ToDateTime** Converts a type (integer or string type) to date-time structures.
5	**ToDecimal** Converts a floating point or integer type to a decimal type.
6	**ToDouble** Converts a type to a double type.
7	**ToInt16** Converts a type to a 16-bit integer.
8	**ToInt32** Converts a type to a 32-bit integer.
9	**ToInt64** Converts a type to a 64-bit integer.
10	**ToSbyte** Converts a type to a signed byte type.
11	**ToSingle** Converts a type to a small floating point number.
12	**ToString** Converts a type to a string.
13	**ToType** Converts a type to a specified type.
14	**ToUInt16** Converts a type to an unsigned int type.
15	**ToUInt32** Converts a type to an unsigned long type.
16	**ToUInt64** Converts a type to an unsigned big integer.

C# - TYPE CONVERSION

- The following example converts various value types to string type:



```
namespace TypeConversionApplication
{
    class StringConversion
    {
        static void Main(string[] args)
        {
            int i = 75;
            float f = 53.005f;
            double d = 2345.7652;
            bool b = true;

            Console.WriteLine(i.ToString());
            Console.WriteLine(f.ToString());
            Console.WriteLine(d.ToString());
            Console.WriteLine(b.ToString());
            Console.ReadKey();

        }
    }
}
```



When the above code is compiled and executed, it produces the following result:

```
75
53.005
2345.7652
True
```

C# - VARIABLES

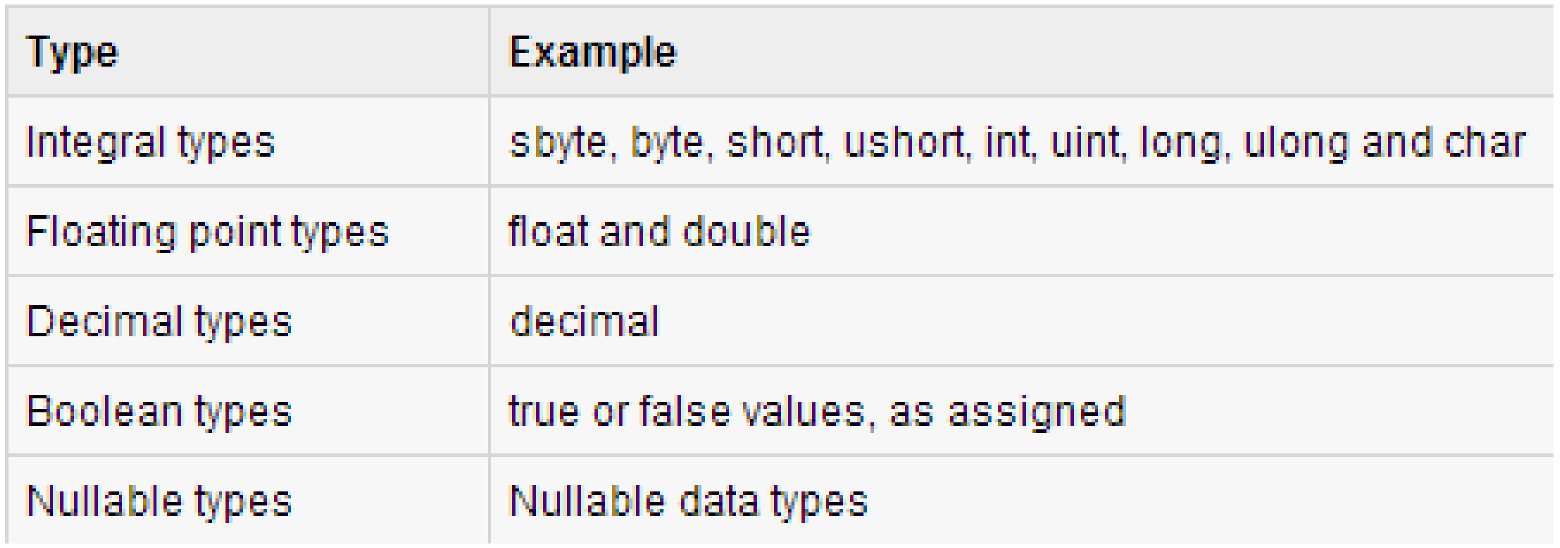
A variable is nothing but a name given to a storage area that our programs can manipulate.

C# - VARIABLES

- A variable is nothing but a name given to a storage area that our programs can manipulate.
- Each variable in C# has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.
- C# also allows defining other value types of variable like **enum** and reference types of variables like **class**, which we will cover in subsequent chapters. For this chapter, let us study only basic variable types.

C# - VARIABLES

- The basic value types provided in C# can be categorized as:



Type	Example
Integral types	sbyte, byte, short, ushort, int, uint, long, ulong and char
Floating point types	float and double
Decimal types	decimal
Boolean types	true or false values, as assigned
Nullable types	Nullable data types

VARIABLE DEFINITION IN C#

- Syntax for variable definition in C# is:
  - **<data_type> <variable_list>;**
- Here, **data_type** must be a valid C# data type including **char**, **int**, **float**, **double**, or any **user-defined data type**, etc., and **variable_list** may consist of one or more identifier names separated by commas.
- Some valid variable definitions are shown here:
  - **int** i, j, k;
  - **char** c, ch;
  - **float** f, salary;
  - **double** d;
- You can initialize a variable at the time of definition as:
  - **int** i = 100;

VARIABLE INITIALIZATION IN C#

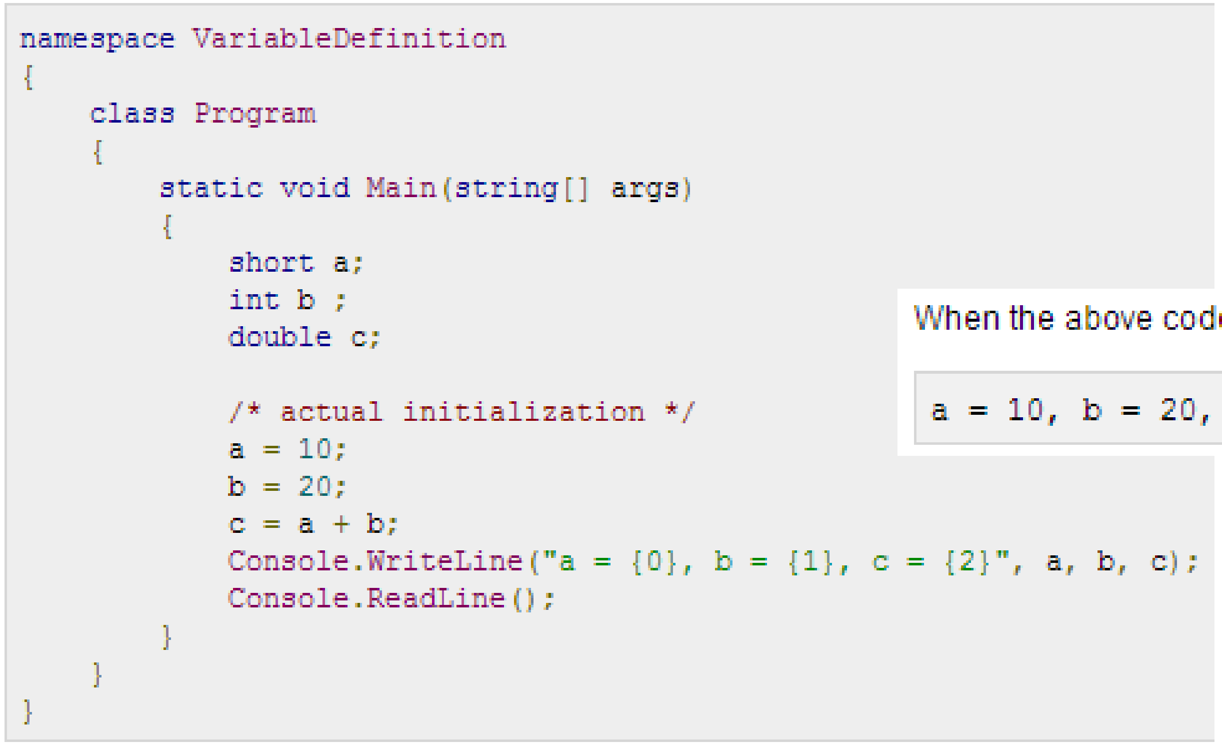
- Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is:
  - **variable_name = value;**
- Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as:
  - **<data_type> <variable_name> = value;**

VARIABLE INITIALIZATION IN C#

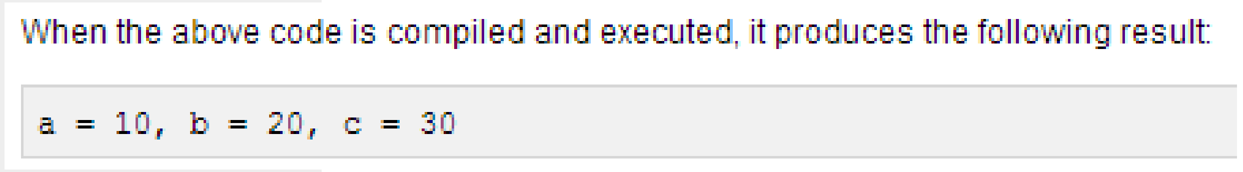
- Some examples are:
  - **int** d = 3, f = 5;   /* initializing d and f. */
  - **byte** z = 22;   /* initializes z. */
  - **double** pi = 3.14159; /* declares an approximation of pi. */
  - **char** x = 'x';   /* the variable x has the value 'x'. */
- It is a good programming practice to initialize variables properly, otherwise sometimes program would produce unexpected result.

VARIABLE INITIALIZATION IN C#

- Try the following example which makes use of various types of variables:



```
namespace VariableDefinition
{
    class Program
    {
        static void Main(string[] args)
        {
            short a;
            int b ;
            double c;

            /* actual initialization */
            a = 10;
            b = 20;
            c = a + b;
            Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);
            Console.ReadLine();
        }
    }
}
```



When the above code is compiled and executed, it produces the following result:

```
a = 10, b = 20, c = 30
```

ACCEPTING VALUES FROM USER

- The **Console** class in the **System** namespace provides a function **ReadLine**() for accepting input from the user and store it into a variable.
- For example,
  - **int** num;
  - **num** = **Convert.ToInt32(Console.ReadLine**());
- The function **Convert.ToInt32()** converts the data entered by the user to **int** data type, because **Console.ReadLine**() accepts the data in string format.

LVALUES AND RVALUES IN C#

- There are two kinds of expressions in C#:
  - **lvalue**: An expression that is an **lvalue** may appear as either the left-hand or right-hand side of an assignment.
  - **rvalue**: An expression that is an **rvalue** may appear on the right- but not left-hand side of an assignment.
- Variables are **lvalues** and so may appear on the left-hand side of an assignment. Numeric literals are **rvalues** and so may not be assigned and can not appear on the left-hand side. Following is a valid statement:
  - **int** g = 20;
- But following is not a valid statement and would generate compile-time error:
  - 10 = 20;

C# - CONSTANTS AND LITERALS

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.

INTEGER LITERALS

- An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: **0x** or **0X** for **hexadecimal**, **0** for **octal**, and no prefix id required for decimal.
- An integer literal can also have a suffix that is a combination of **U** and **L**, for **unsigned** and **long**, respectively. The suffix can be uppercase or lowercase and can be in any order.

INTEGER LITERALS

- Here are some examples of integer literals:
  - 212 /* Legal */
  - 215**u** /* Legal */
  - **0x**Fee**L** /* Legal */
  - 07**8** /* Illegal: 8 is not an octal digit */
  - 032**UU** /* Illegal: cannot repeat a suffix */

INTEGER LITERALS

- Following are other examples of various types of Integer literals:
  - 85   /* decimal */
  - 0213   /* octal */
  - **0x**4b   /* hexadecimal */
  - 30   /* int */
  - 30**u**   /* unsigned int */
  - 30**l**   /* long */
  - 30**ul**   /* unsigned long */

FLOATING-POINT LITERALS

- A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.
- Here are some examples of floating-point literals:
  - 3.14159   /* Legal */
  - 314159**E**-5**L**   /* Legal */
  - 510**E**   /* Illegal: incomplete exponent */
  - 210**f**   /* Illegal: no decimal or exponent */
  - .e55   /* Illegal: missing integer or fraction */

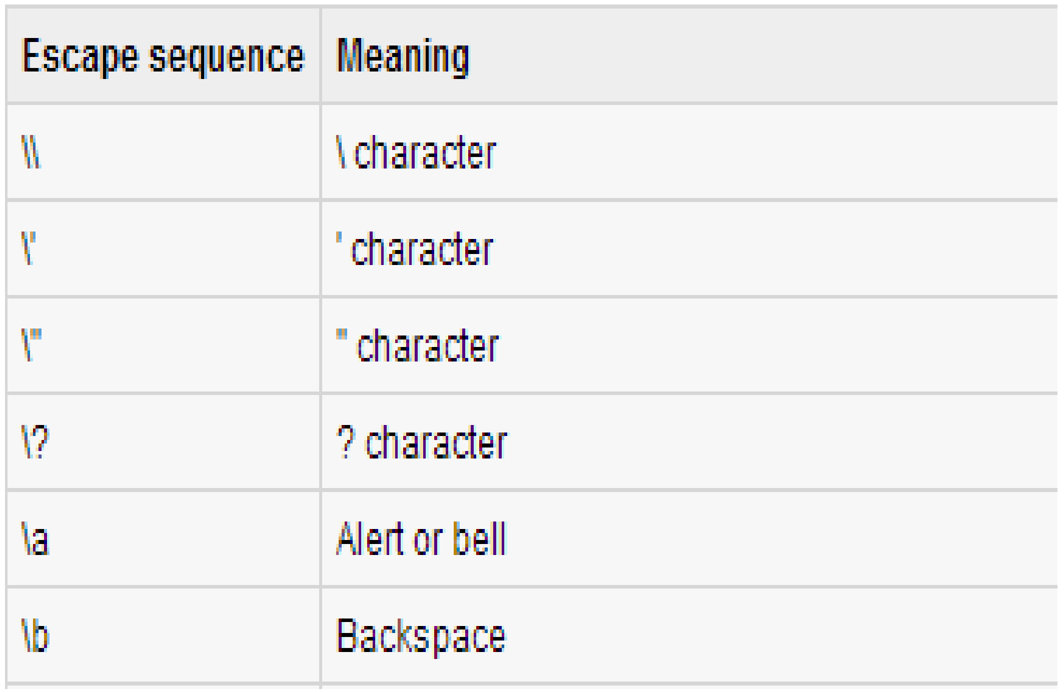
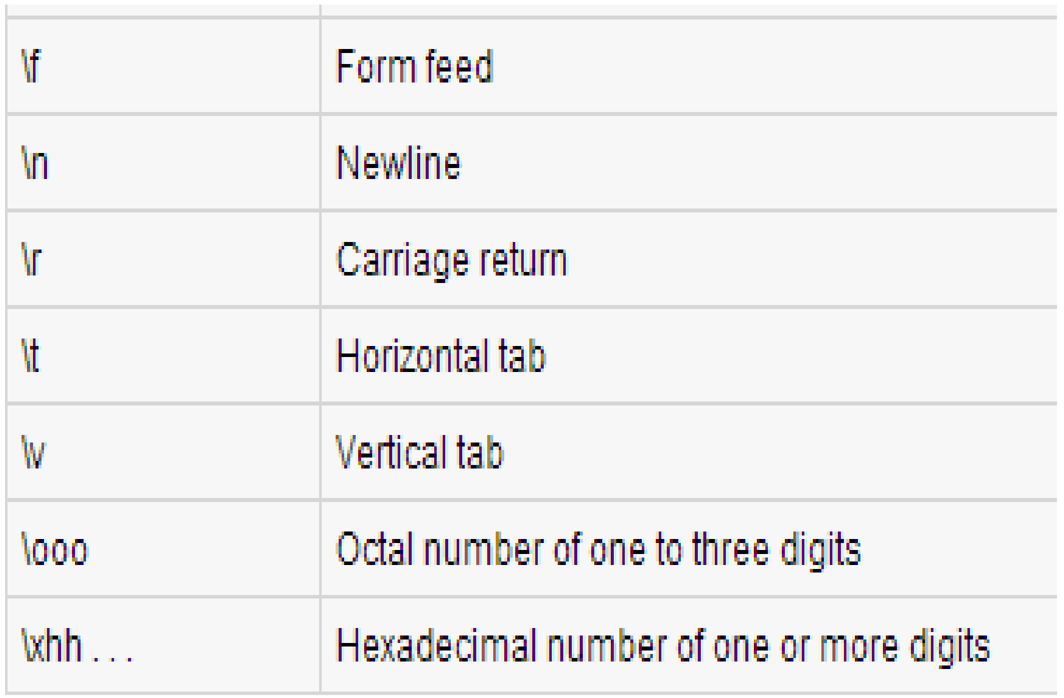
FLOATING-POINT LITERALS

- While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form you must include the integer part, the fractional part, or both.
- The signed exponent is introduced by e or E.

CHARACTER CONSTANTS

- Character literals are enclosed in single quotes, e.g., 'x' and can be stored in a simple variable of char type. A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').
- There are certain characters in C# when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t).

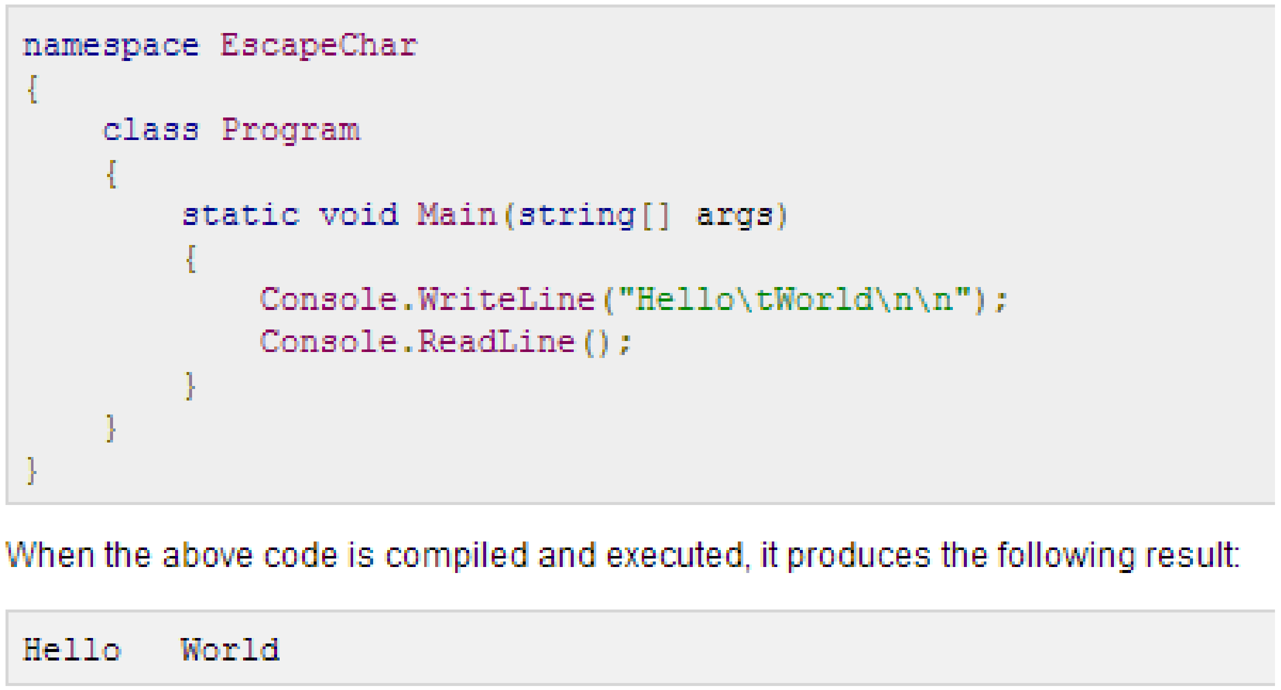
CHARACTER CONSTANTS

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

CHARACTER CONSTANTS

- Following is the example to show few escape sequence characters:



```
namespace EscapeChar
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello\tWorld\n\n");
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Hello   World
```

STRING LITERALS

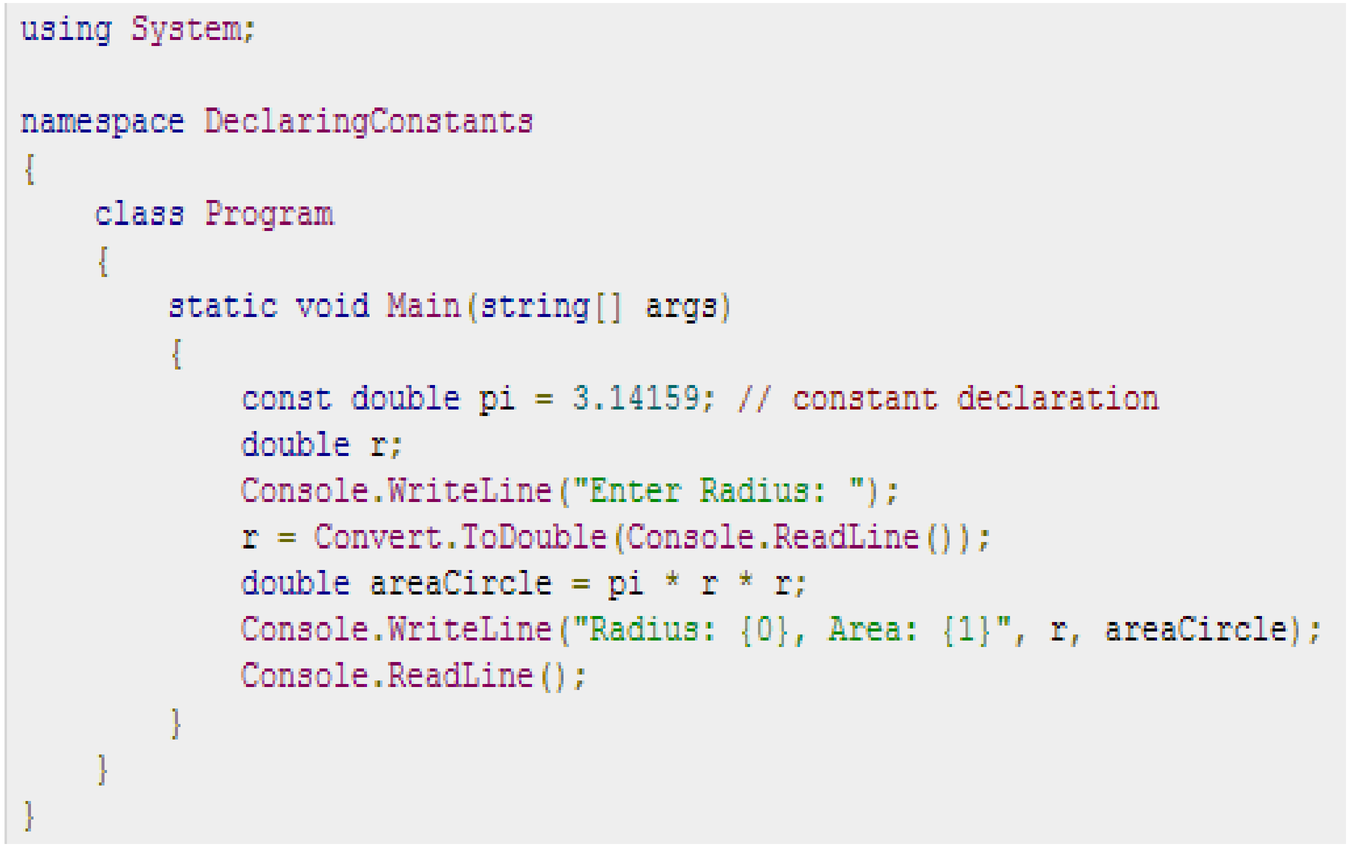
- String literals or constants are enclosed in double quotes "" or with @"".
- A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.
- You can break a long line into multiple lines using string literals and separating the parts using whitespaces.

STRING LITERALS

- Here are some examples of string literals. All the three forms are identical strings.
  - "hello, dear"
  - "hello, \
  - dear"
  - "hello, " "d" "ear"
  - @"hello dear"

DEFINING CONSTANTS

- Constants are defined using the **const** keyword. Syntax for defining a constant is:
  - **const <data_type> <constant_name> = value;**
- The following program demonstrates defining and using a constant in your program:



```
using System;

namespace DeclaringConstants
{
    class Program
    {
        static void Main(string[] args)
        {
            const double pi = 3.14159; // constant declaration
            double r;
            Console.WriteLine("Enter Radius: ");
            r = Convert.ToDouble(Console.ReadLine());
            double areaCircle = pi * r * r;
            Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);
            Console.ReadLine();
        }
    }
}
```

DEFINING CONSTANTS

- When the above code is compiled and executed, it produces the following result:
  - Enter Radius:
  - 3
  - Radius: 3, Area: 28.27431

C# - OPERATORS

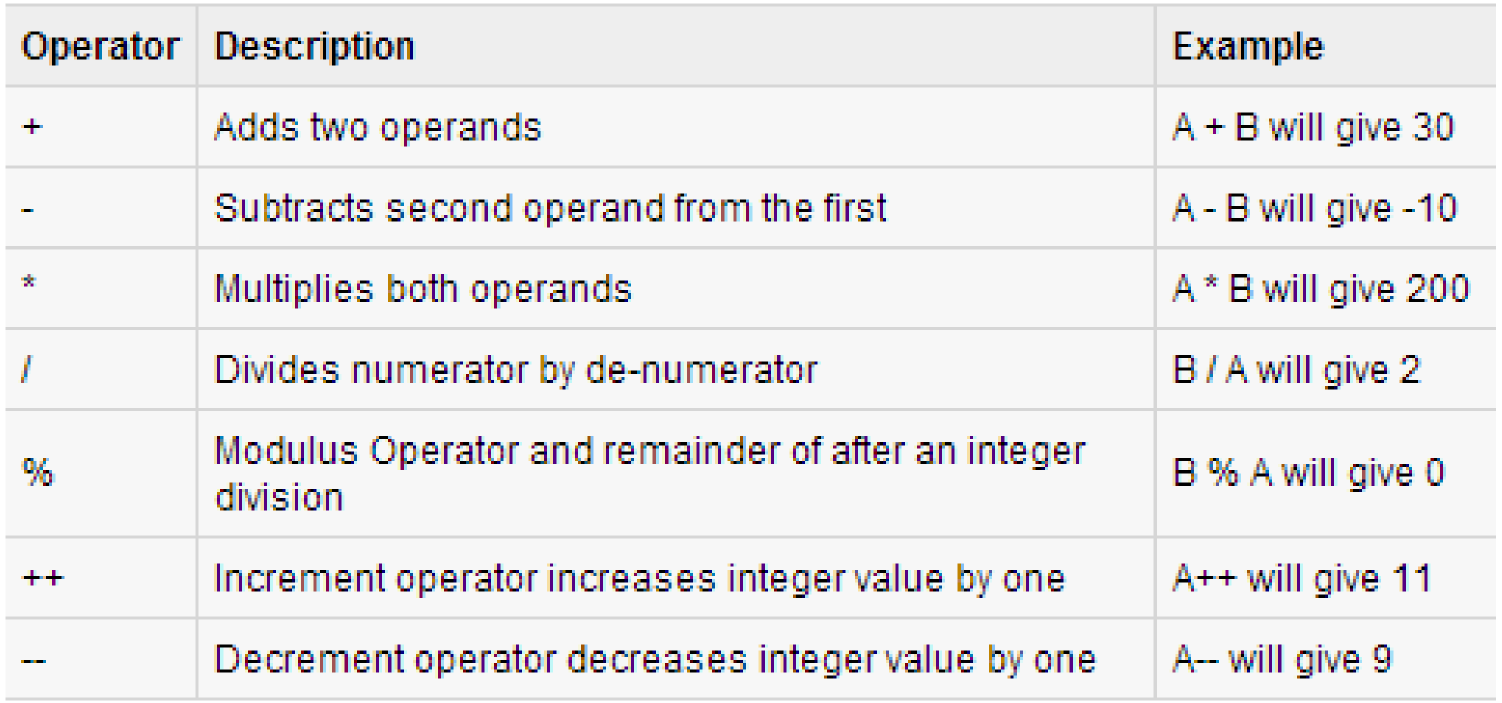
An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

C# - OPERATORS

- An **operator** is a symbol that 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
  - **Misc** Operators
- This tutorial will explain the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

ARITHMETIC OPERATORS

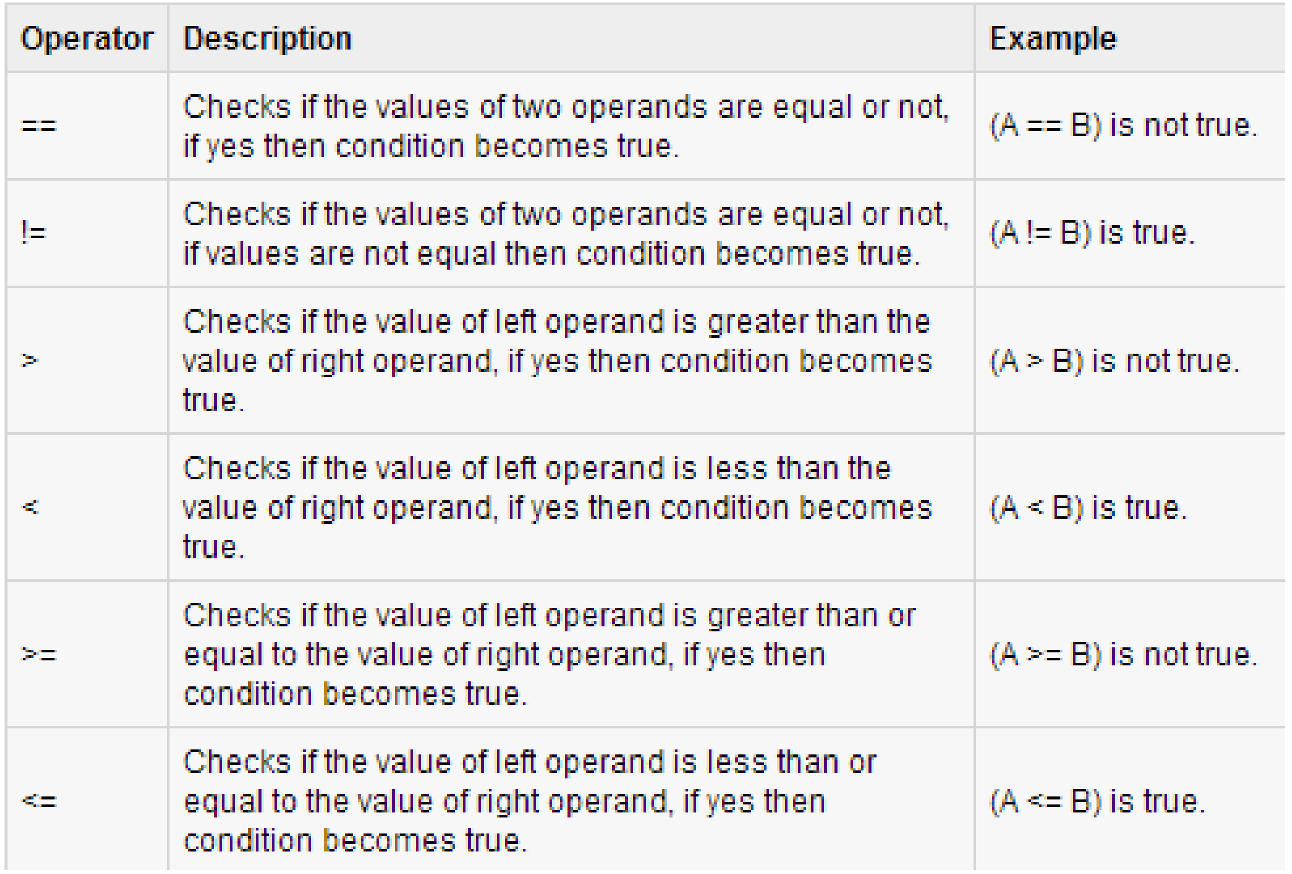
- Following table shows all the arithmetic operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20 then:



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

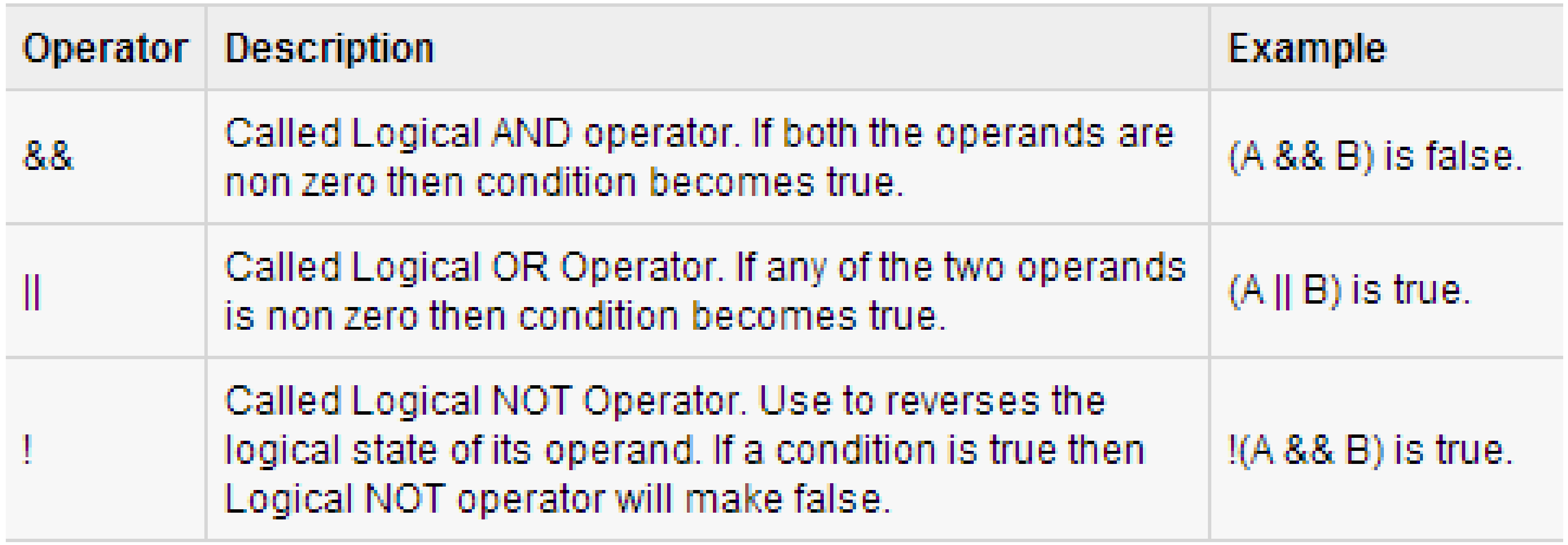
- Following table shows all the relational operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20, then:



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.
<	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

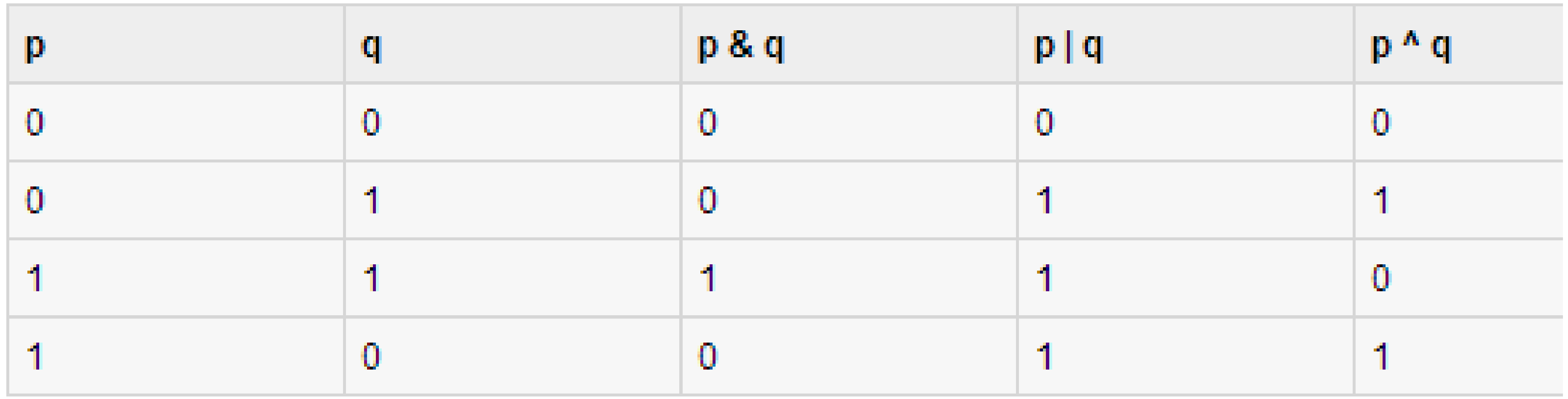
- Following table shows all the logical operators supported by C#. Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then:



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

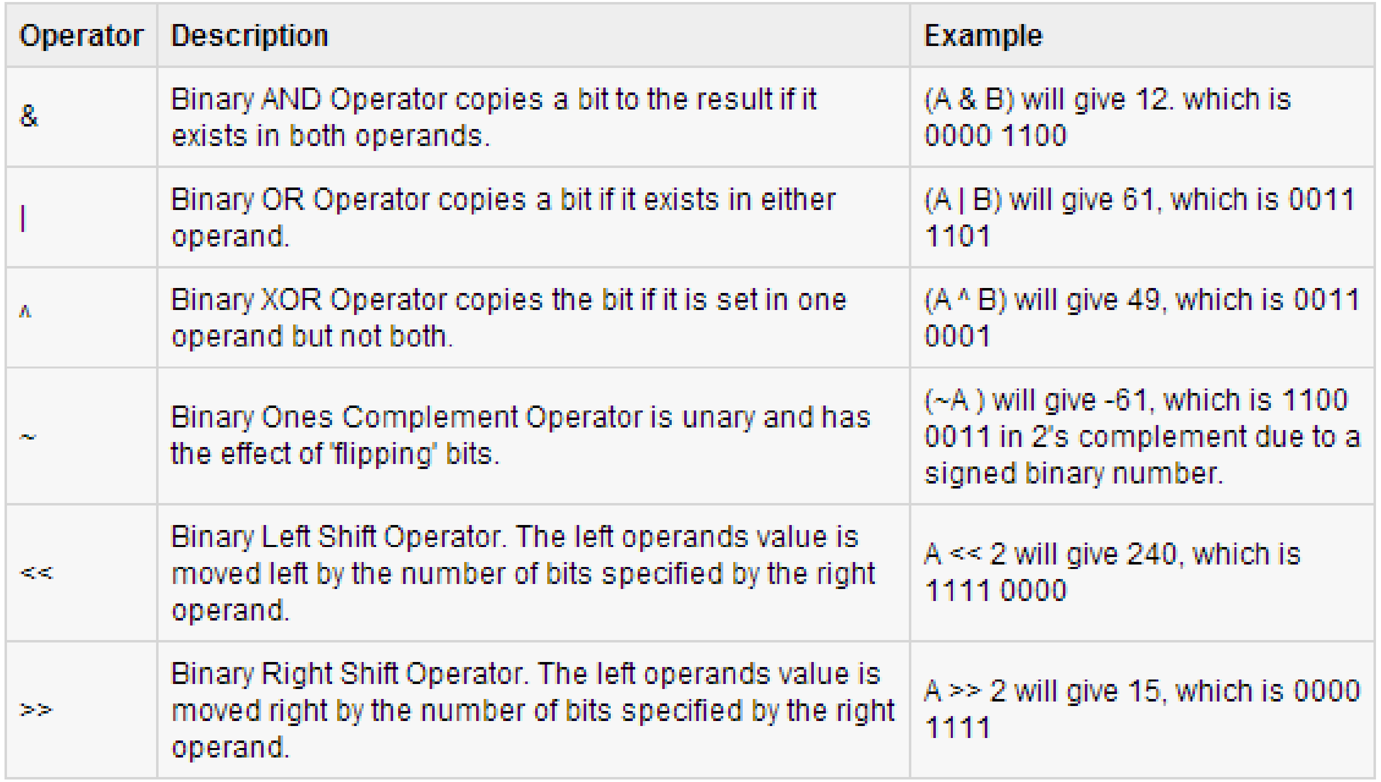
- Bitwise operator works on bits and perform bit by bit operation. The truth tables for &, | , and ^ are as follows:



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

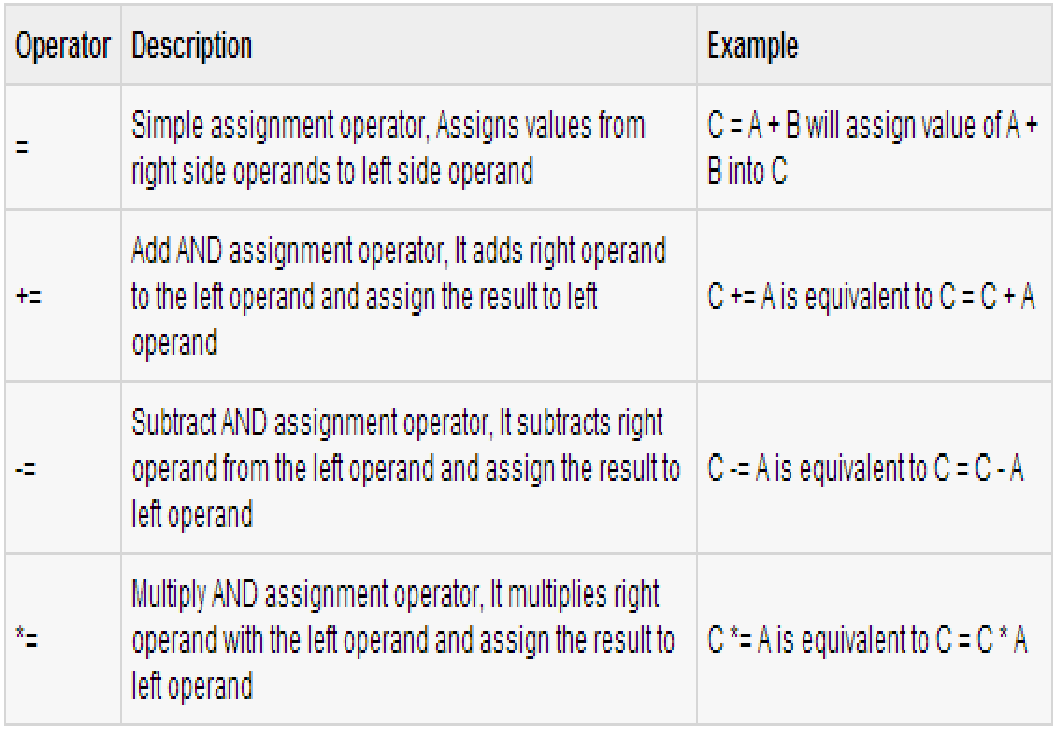
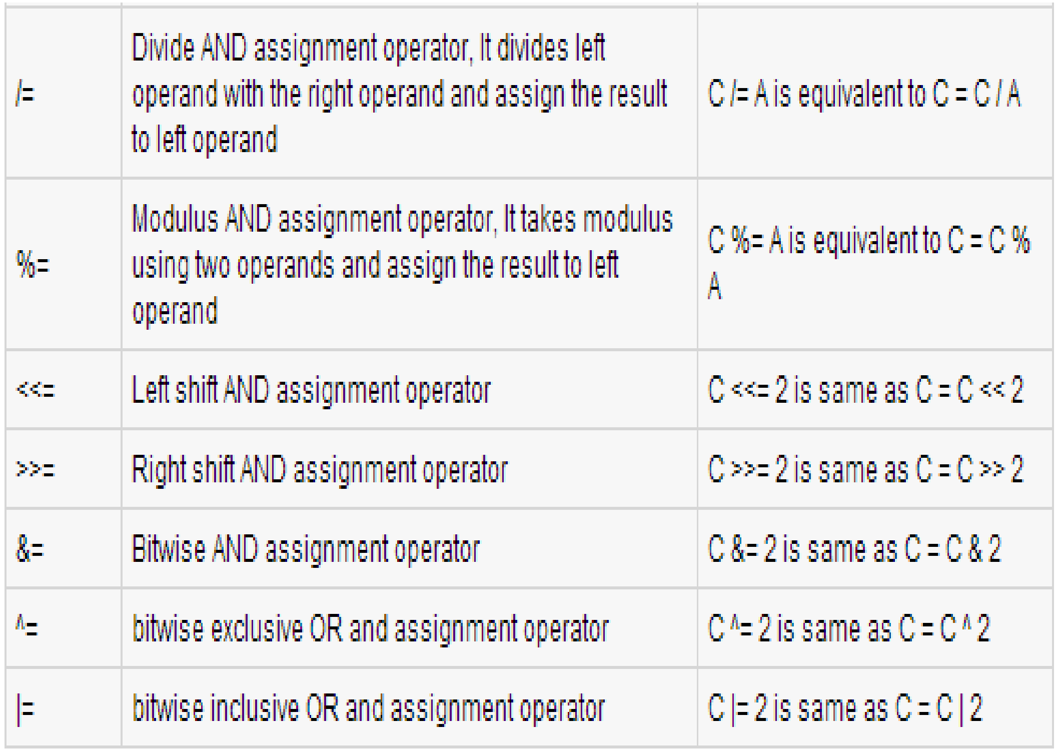
BITWISE OPERATORS

- The Bitwise operators supported by C# are listed in the following table. Assume variable A holds 60 and variable B holds 13 then:



Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12. which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61, which is 1100 0011 in 2's complement due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111

ASSIGNMENT OPERATORS

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

MISC OPERATORS

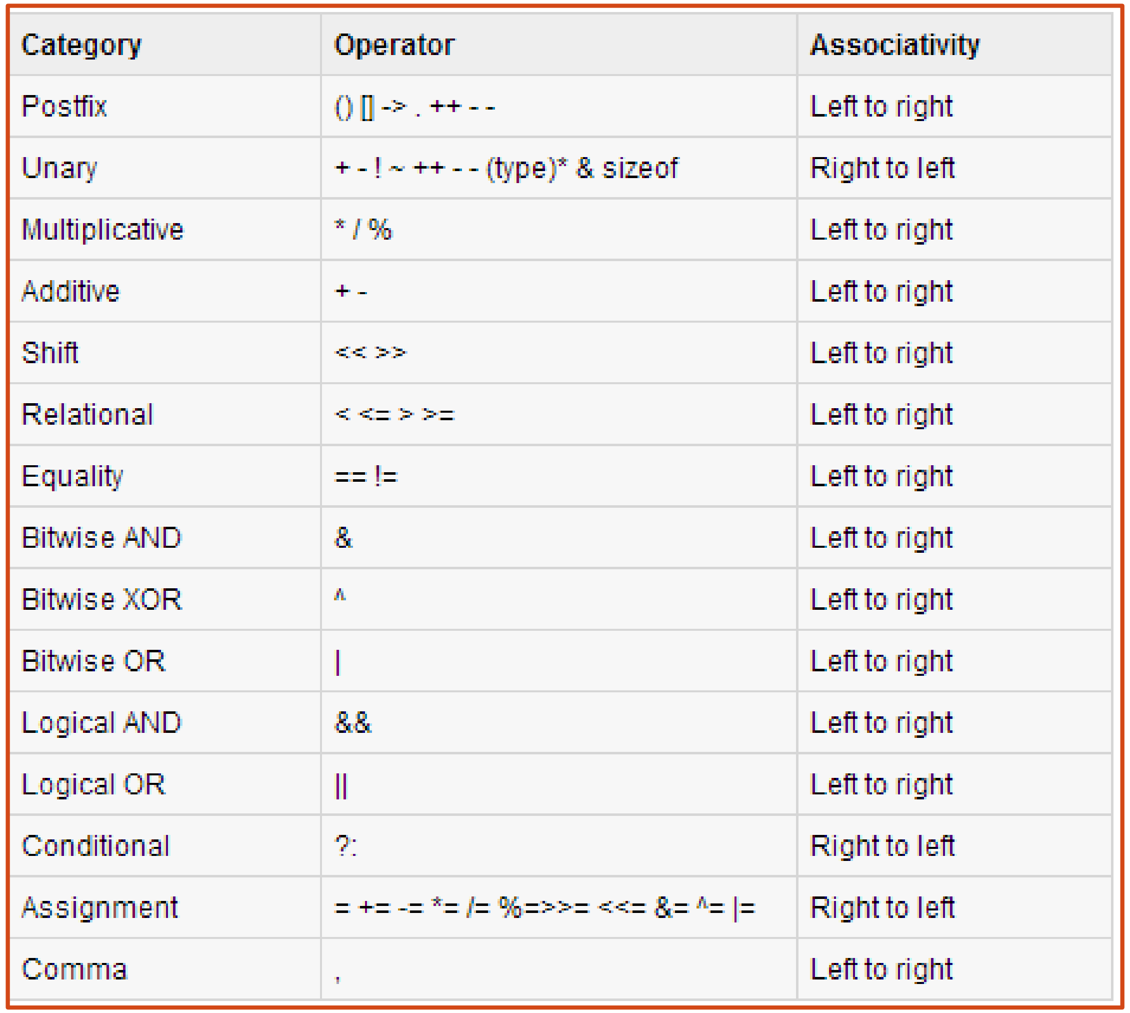
Operator	Description	Example
sizeof()	Returns the size of a data type.	sizeof(int), will return 4.
typeof()	Returns the type of a class.	typeof(StreamReader);
&	Returns the address of an variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
?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

OPERATORS PRECEDENCE IN C#

- Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:
- For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

OPERATORS PRECEDENCE IN C#

- Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.
- Within an expression, higher precedence operators will be evaluated first.



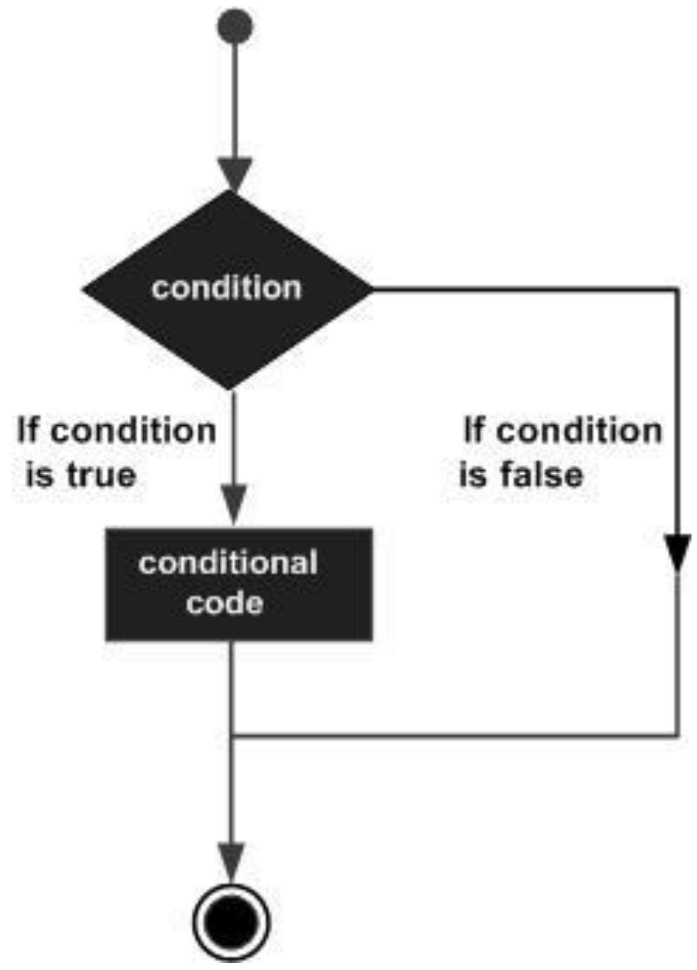
Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

C# - DECISION MAKING

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.

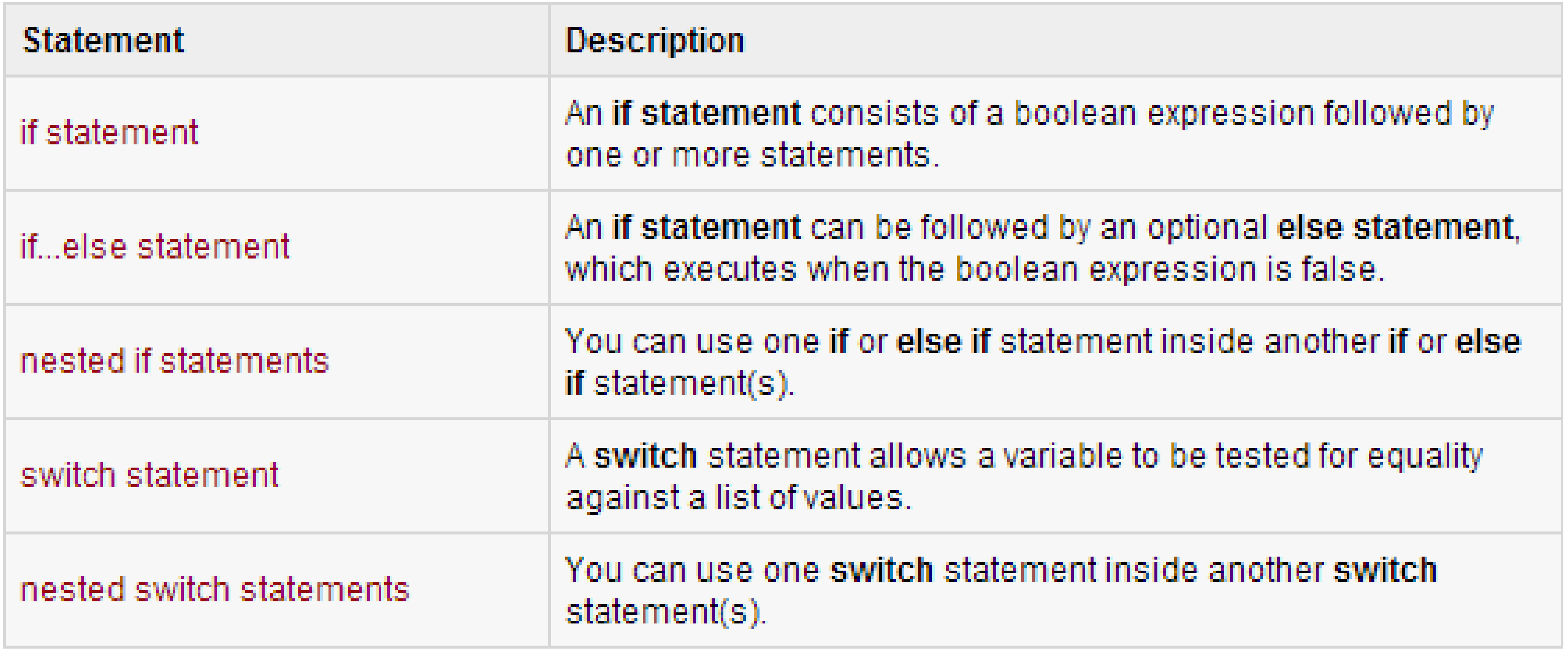
C# - DECISION MAKING

- Following is the general from of a typical decision making structure found in most of the programming languages:



C# - DECISION MAKING

- C# provides following types of decision making statements.

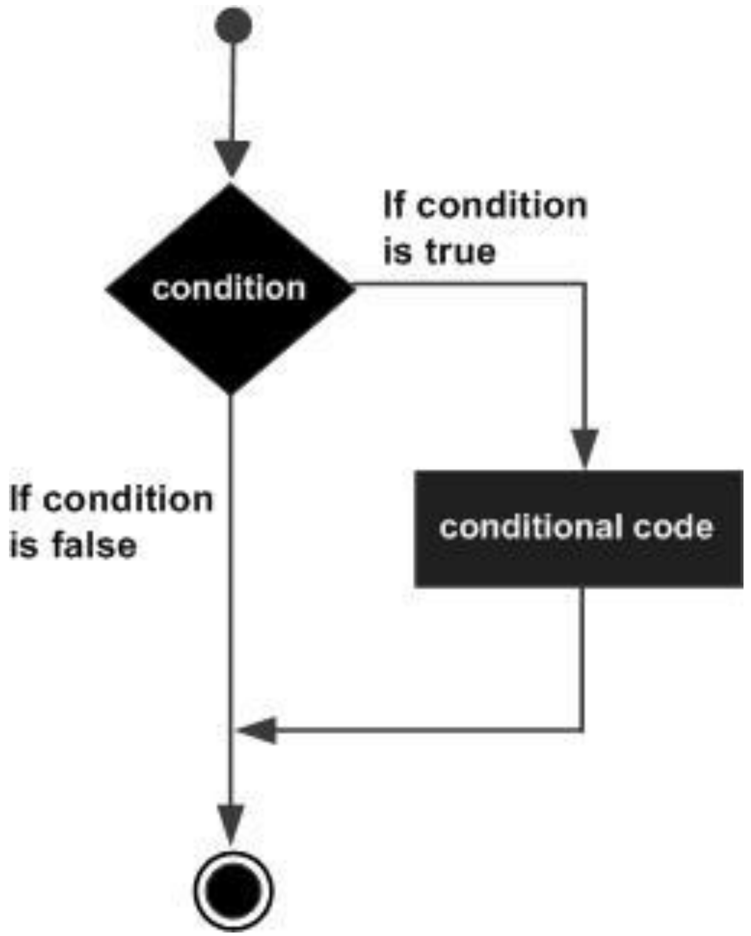


Statement	Description
if statement	An **if statement** consists of a boolean expression followed by one or more statements.
if...else statement	An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
switch statement	A **switch** statement allows a variable to be tested for equality against a list of values.
nested switch statements	You can use one **switch** statement inside another **switch** statement(s).

C# - IF STATEMENT

- The syntax of an if statement in C# is:

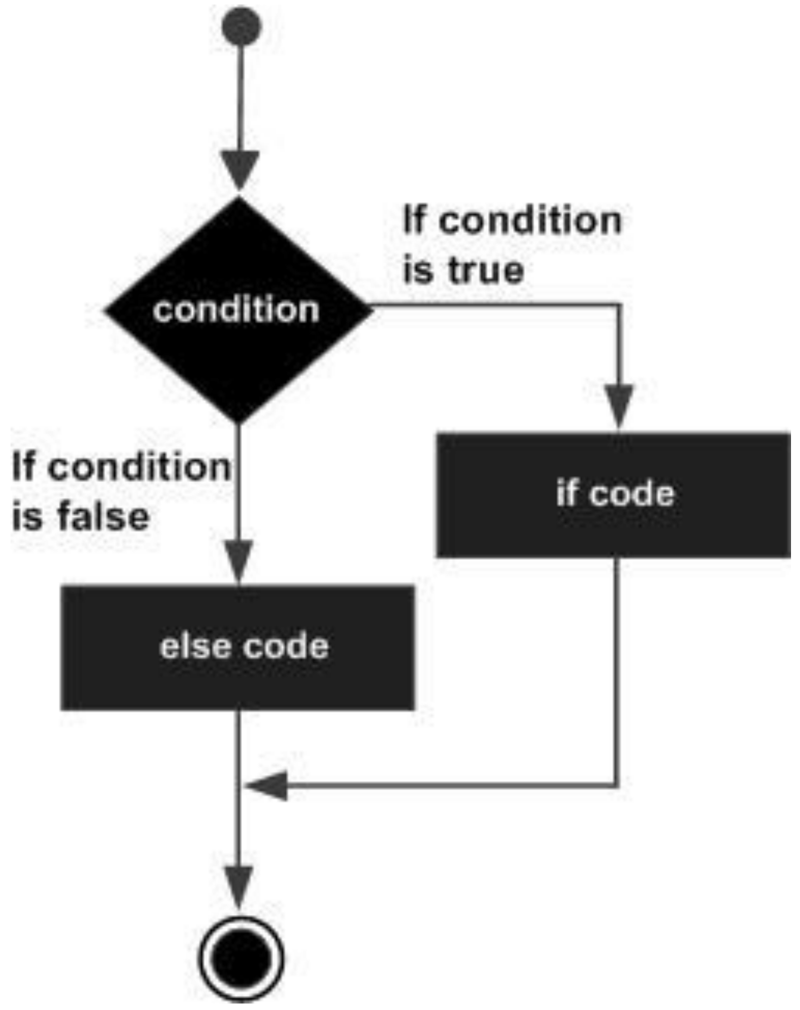
```
if(boolean_expression)
{
   /* statement(s) will execute if the
boolean expression is true */
}
```



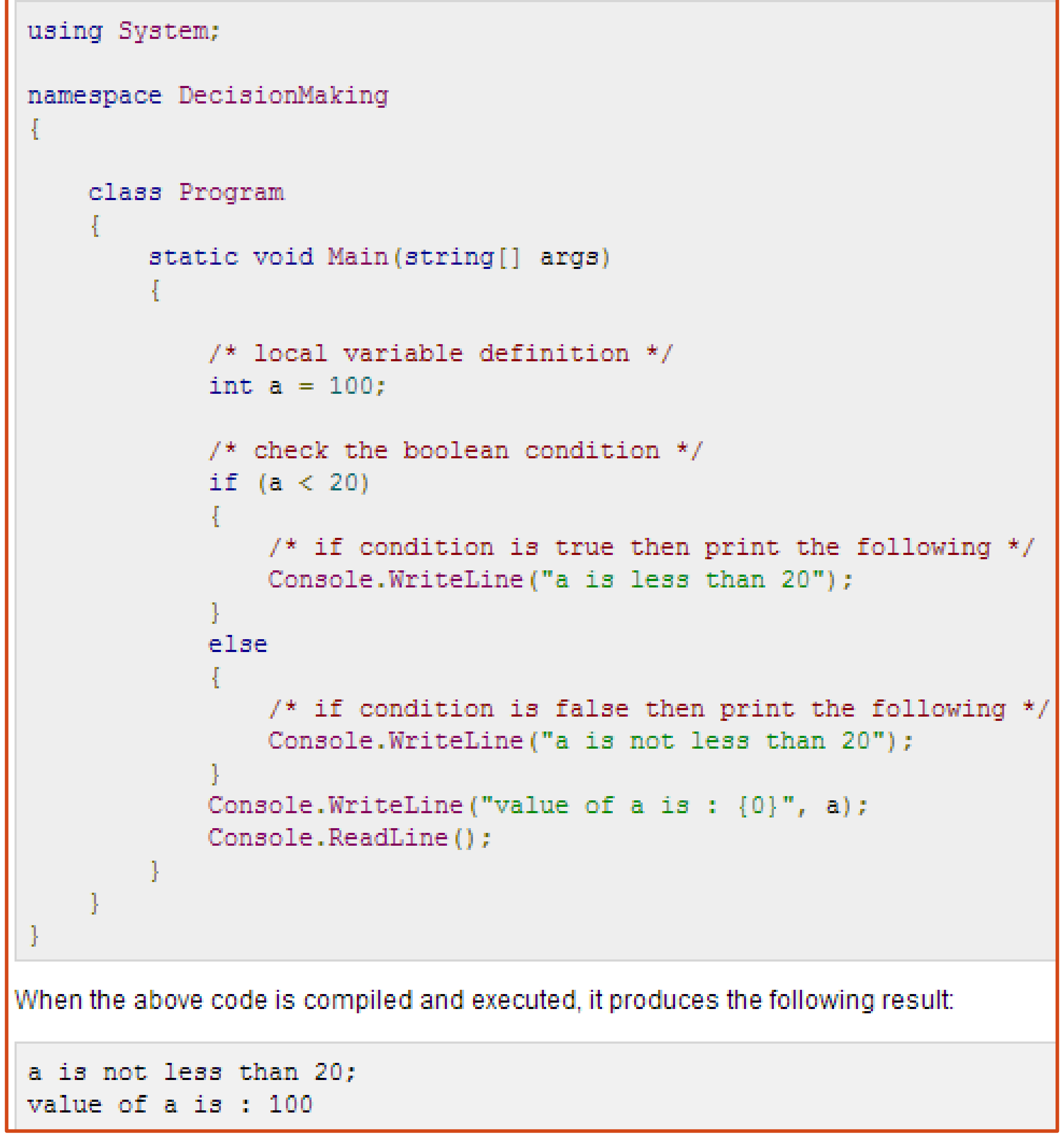
C# - IF...ELSE STATEMENT

- The syntax of an if 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 */
}
```



C# - IF ...ELSE STATEMENT



```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {

            /* local variable definition */
            int a = 100;

            /* check the boolean condition */
            if (a < 20)
            {
                /* if condition is true then print the following */
                Console.WriteLine("a is less than 20");
            }
            else
            {
                /* if condition is false then print the following */
                Console.WriteLine("a is not less than 20");
            }
            Console.WriteLine("value of a is : {0}", a);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

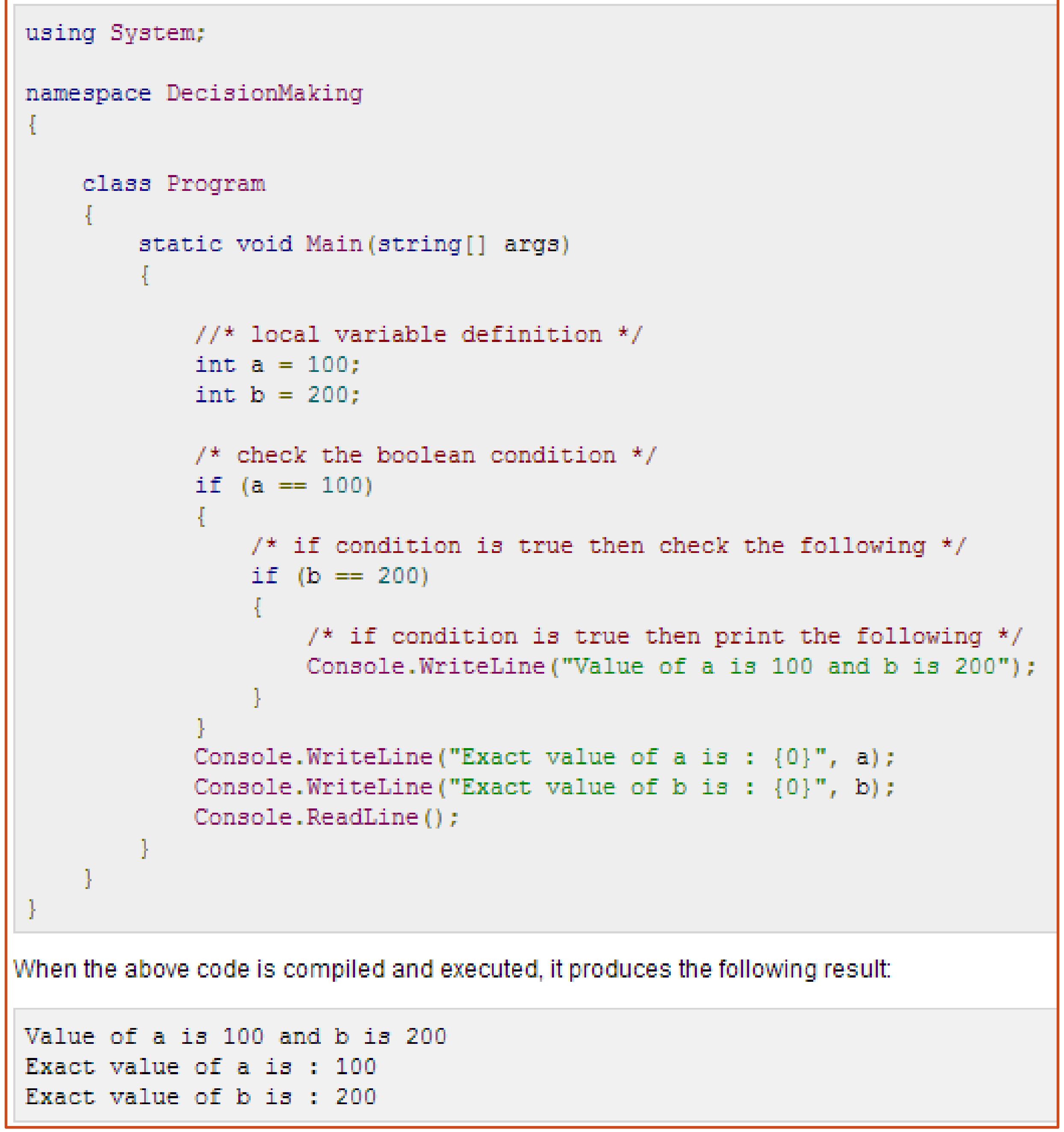
```
a is not less than 20;
value of a is : 100
```

C# - NESTED IF STATEMENTS

- The syntax for a nested if statement is as follows:

```
if( boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
   if(boolean_expression 2)
   {
     /* Executes when the boolean expression 2 is true */
   }
}
```

C# - NESTED IF STATEMENTS



```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {

            //* local variable definition */
            int a = 100;
            int b = 200;

            /* check the boolean condition */
            if (a == 100)
            {
                /* if condition is true then check the following */
                if (b == 200)
                {
                    /* if condition is true then print the following */
                    Console.WriteLine("Value of a is 100 and b is 200");
                }
            }
            Console.WriteLine("Exact value of a is : {0}", a);
            Console.WriteLine("Exact value of b is : {0}", b);
            Console.ReadLine();
        }
    }
}
```

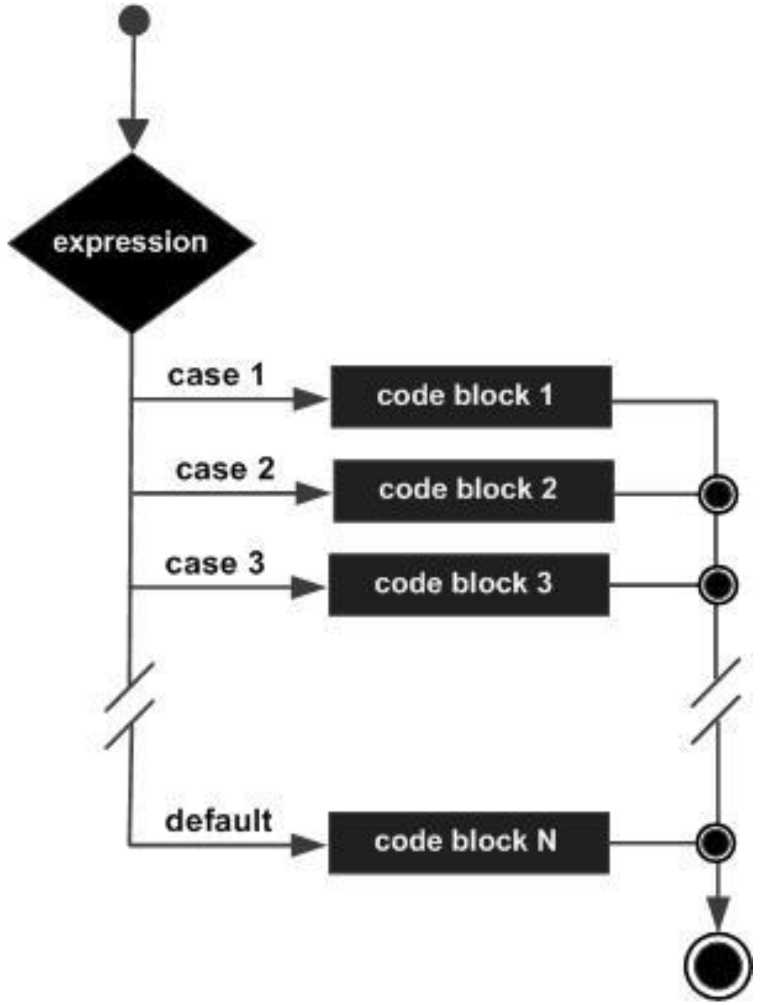
When the above code is compiled and executed, it produces the following result:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

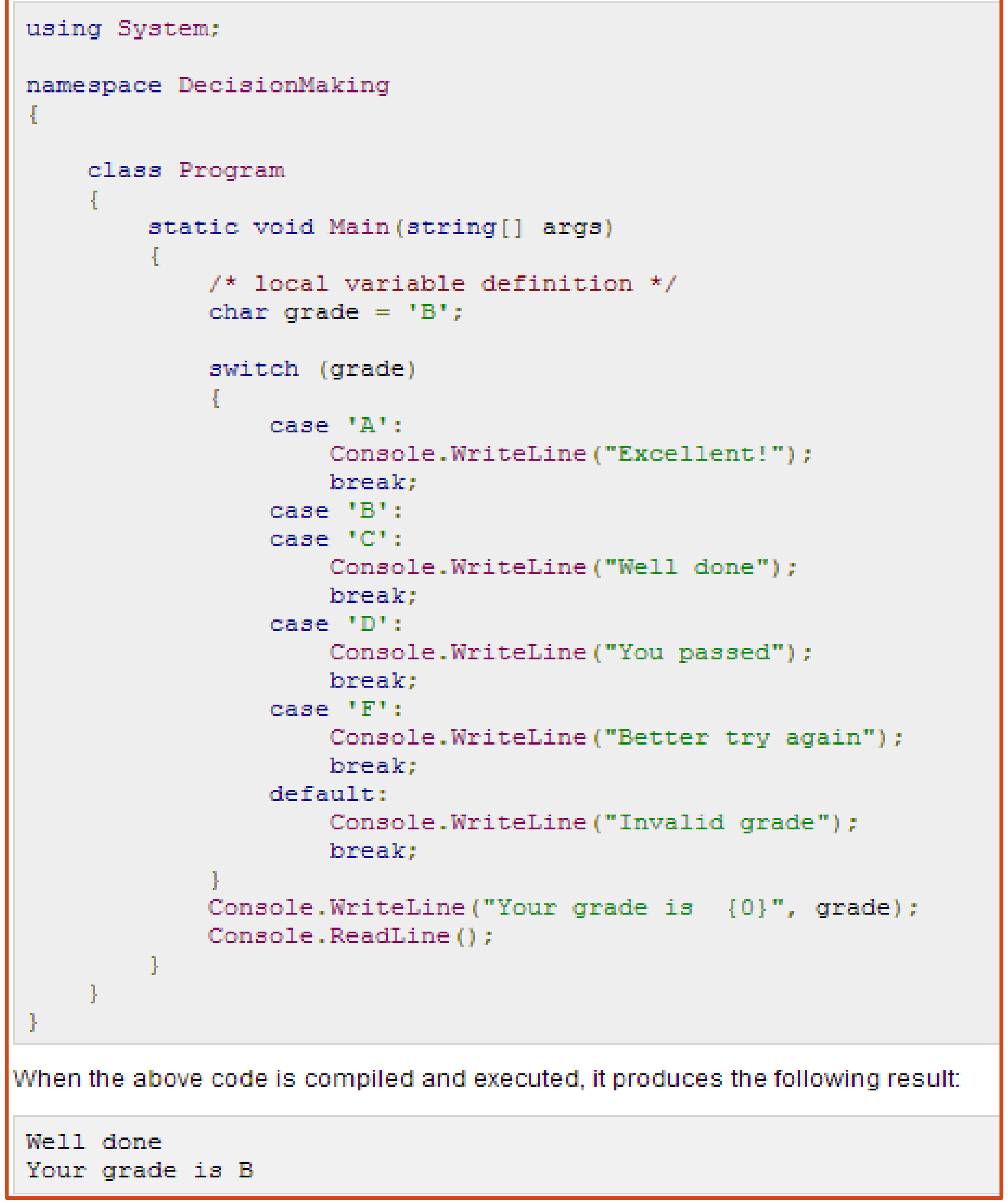
C# - SWITCH STATEMENT

- 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);
}
```



C# - SWITCH STATEMENT



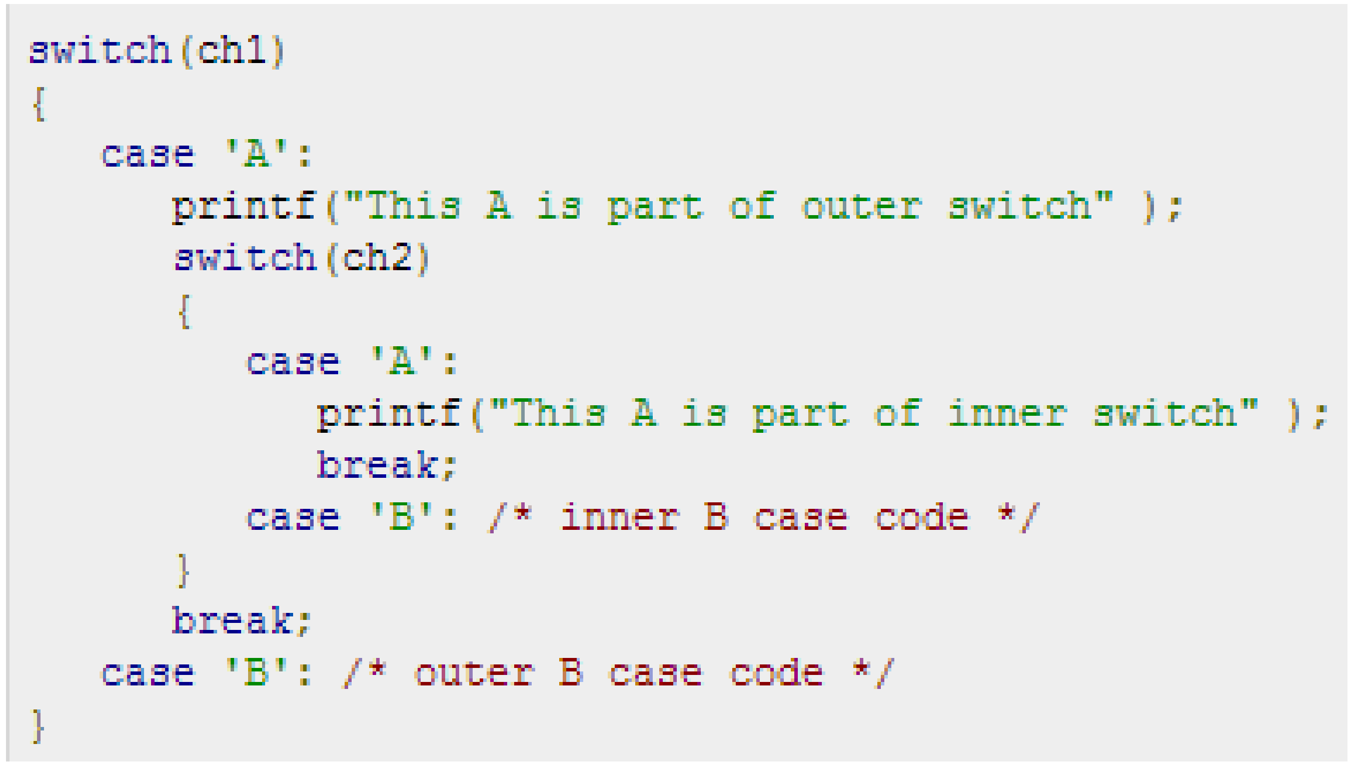
```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            char grade = 'B';

            switch (grade)
            {
                case 'A':
                    Console.WriteLine("Excellent!");
                    break;
                case 'B':
                case 'C':
                    Console.WriteLine("Well done");
                    break;
                case 'D':
                    Console.WriteLine("You passed");
                    break;
                case 'F':
                    Console.WriteLine("Better try again");
                    break;
                default:
                    Console.WriteLine("Invalid grade");
                    break;
            }
            Console.WriteLine("Your grade is  {0}", grade);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Well done
Your grade is B
```

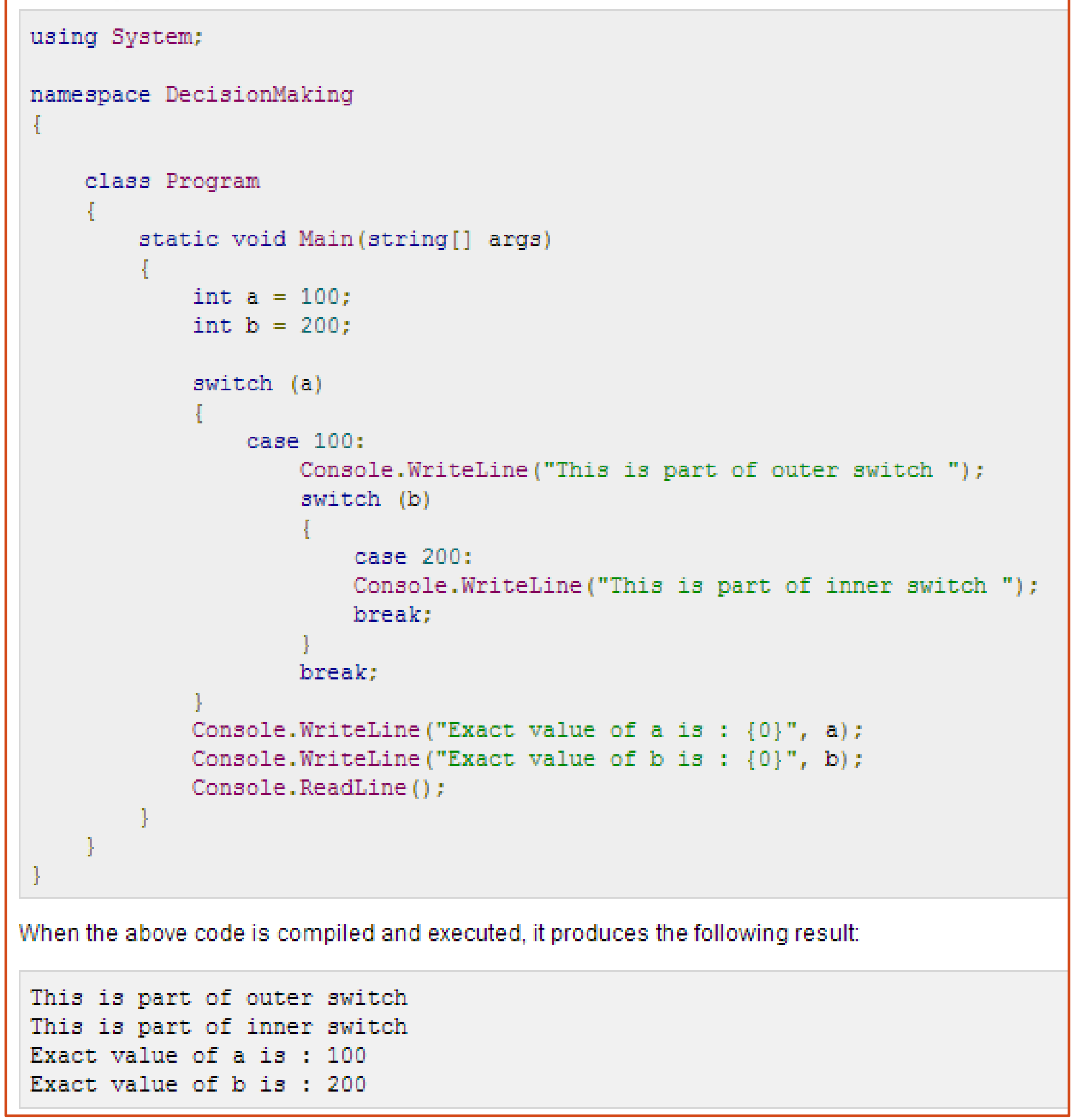
C# - NESTED SWITCH STATEMENTS

- The syntax for a **nested switch** statement in C# is as follows:



```
switch(ch1)
{
   case 'A':
      printf("This A is part of outer switch" );
      switch(ch2)
      {
         case 'A':
            printf("This A is part of inner switch" );
            break;
         case 'B': /* inner B case code */
      }
      break;
   case 'B': /* outer B case code */
}
```

C# - NESTED SWITCH STATEMENT



```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {
            int a = 100;
            int b = 200;

            switch (a)
            {
                case 100:
                    Console.WriteLine("This is part of outer switch ");
                    switch (b)
                    {
                        case 200:
                        Console.WriteLine("This is part of inner switch ");
                        break;
                    }
                    break;
            }
            Console.WriteLine("Exact value of a is : {0}", a);
            Console.WriteLine("Exact value of b is : {0}", b);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200
```

THE ? : OPERATOR

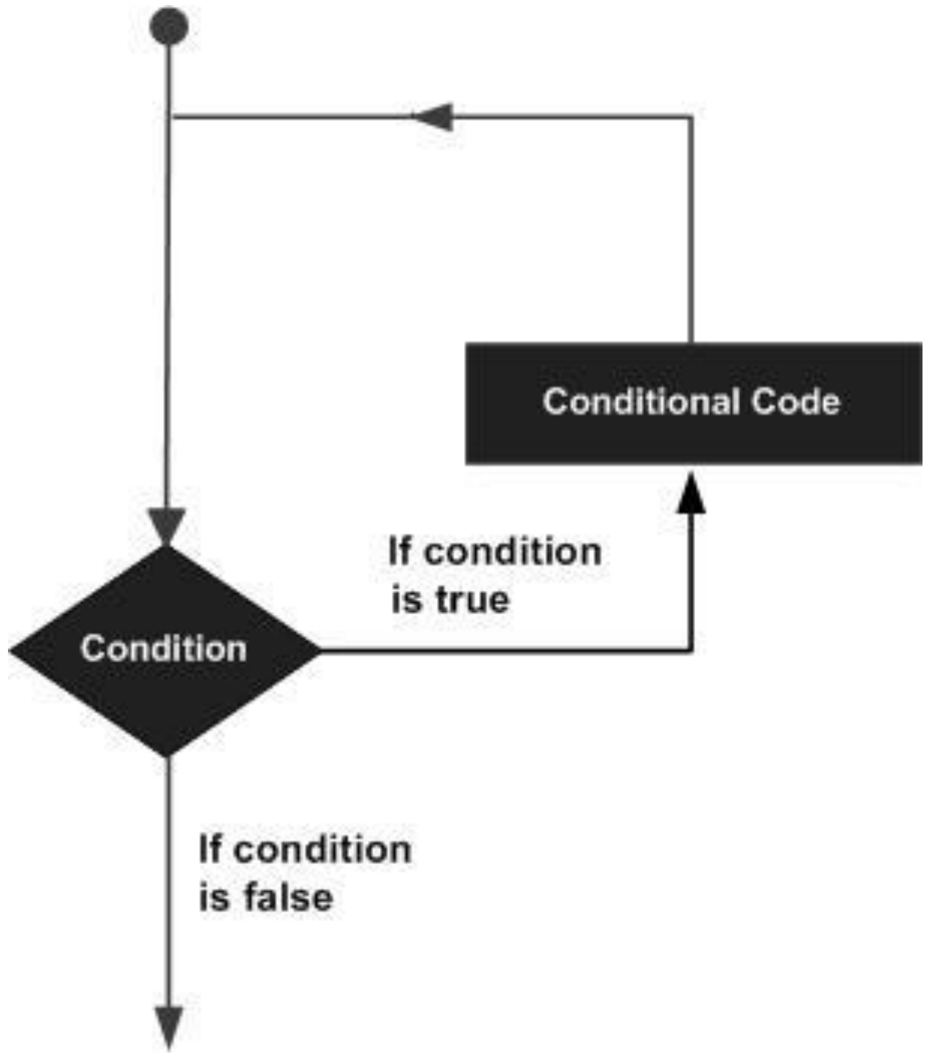
- We have covered conditional operator ? : in previous chapter which can be used to replace if...else statements. It has the following general form:
  - **Exp1 ? Exp2 : Exp3;**
- Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.
- The value of a ? expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

C# - LOOPS

There may be a situation, when you need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

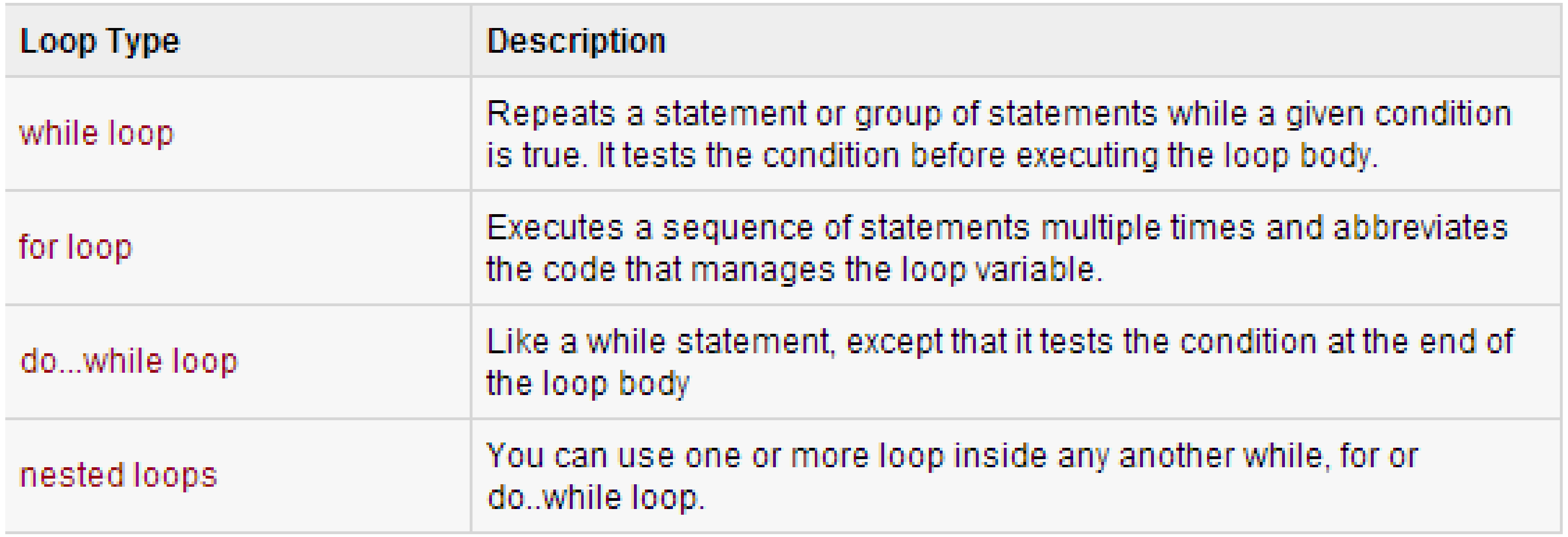
C# - LOOPS

- Programming languages provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



C# - LOOPS

- C# provides following types of loop to handle looping requirements. Click the following links to check their detail.

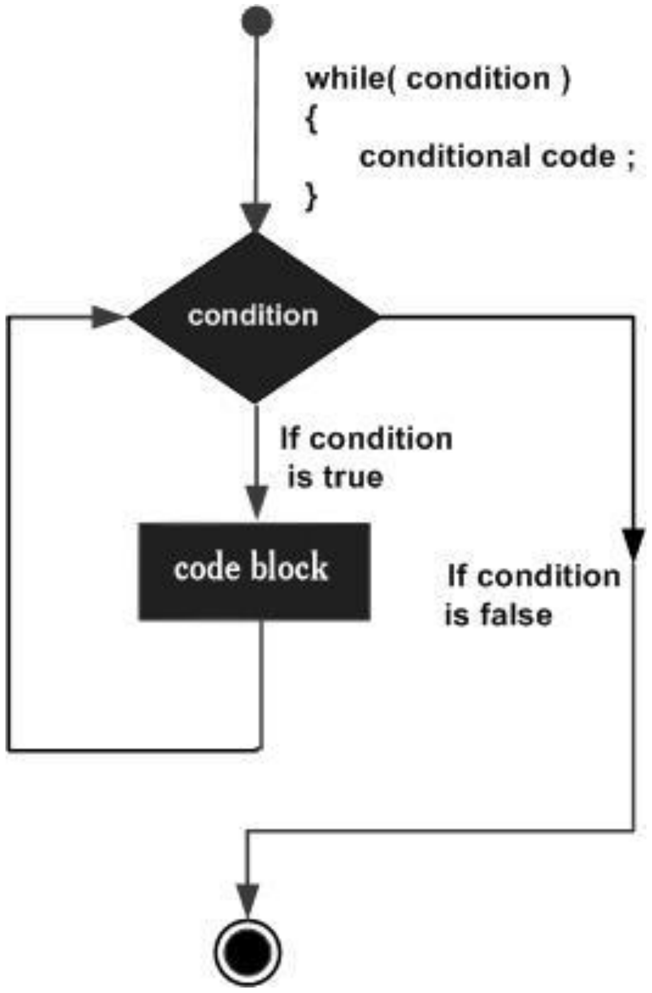


Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a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

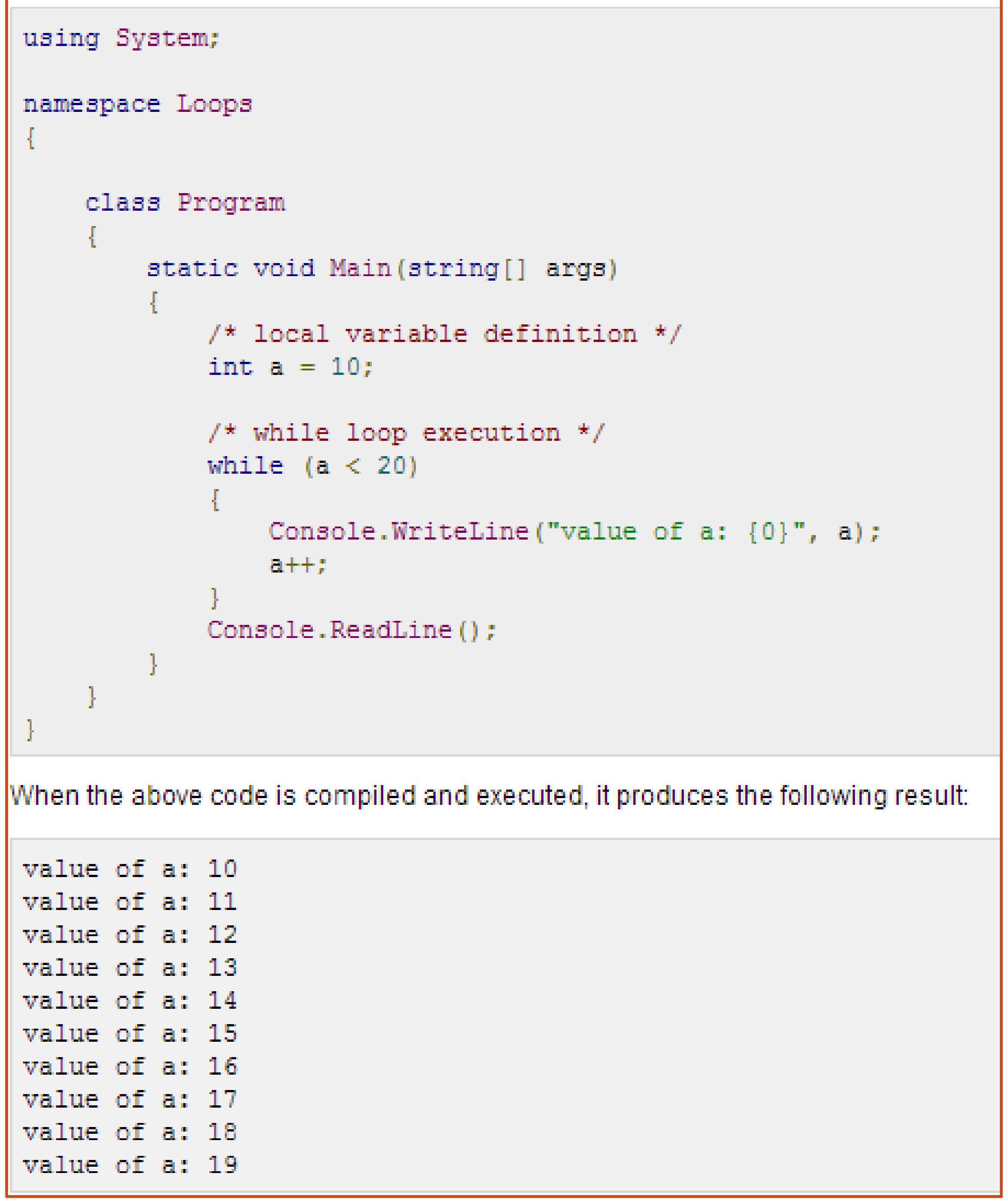
C# - WHILE LOOP

- A **while** loop statement in C# repeatedly executes a target statement as long as a given condition is true.
- The syntax of a while loop in C# is:

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



C# - WHILE LOOP



```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 10;

            /* while loop execution */
            while (a < 20)
            {
                Console.WriteLine("value of a: {0}", a);
                a++;
            }
            Console.ReadLine();
        }
    }
}
```

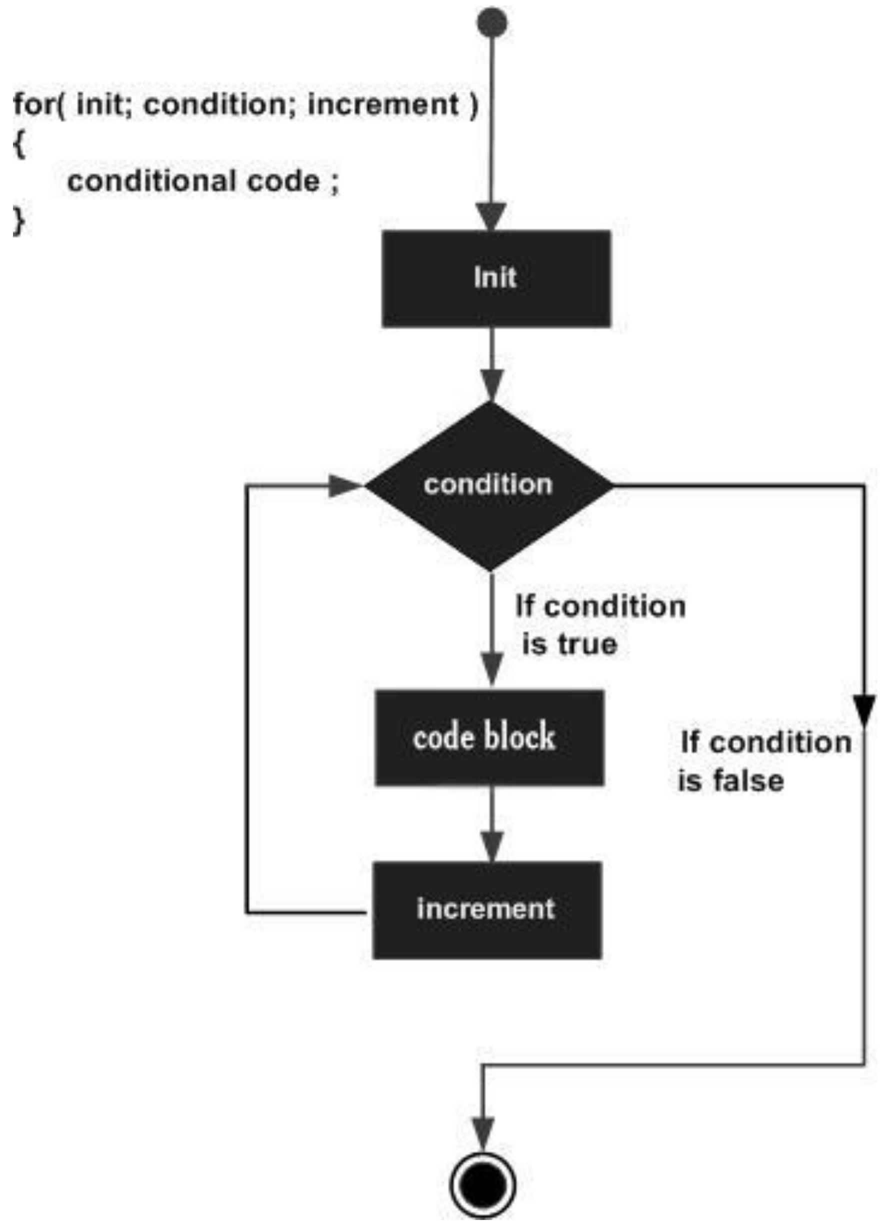
When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

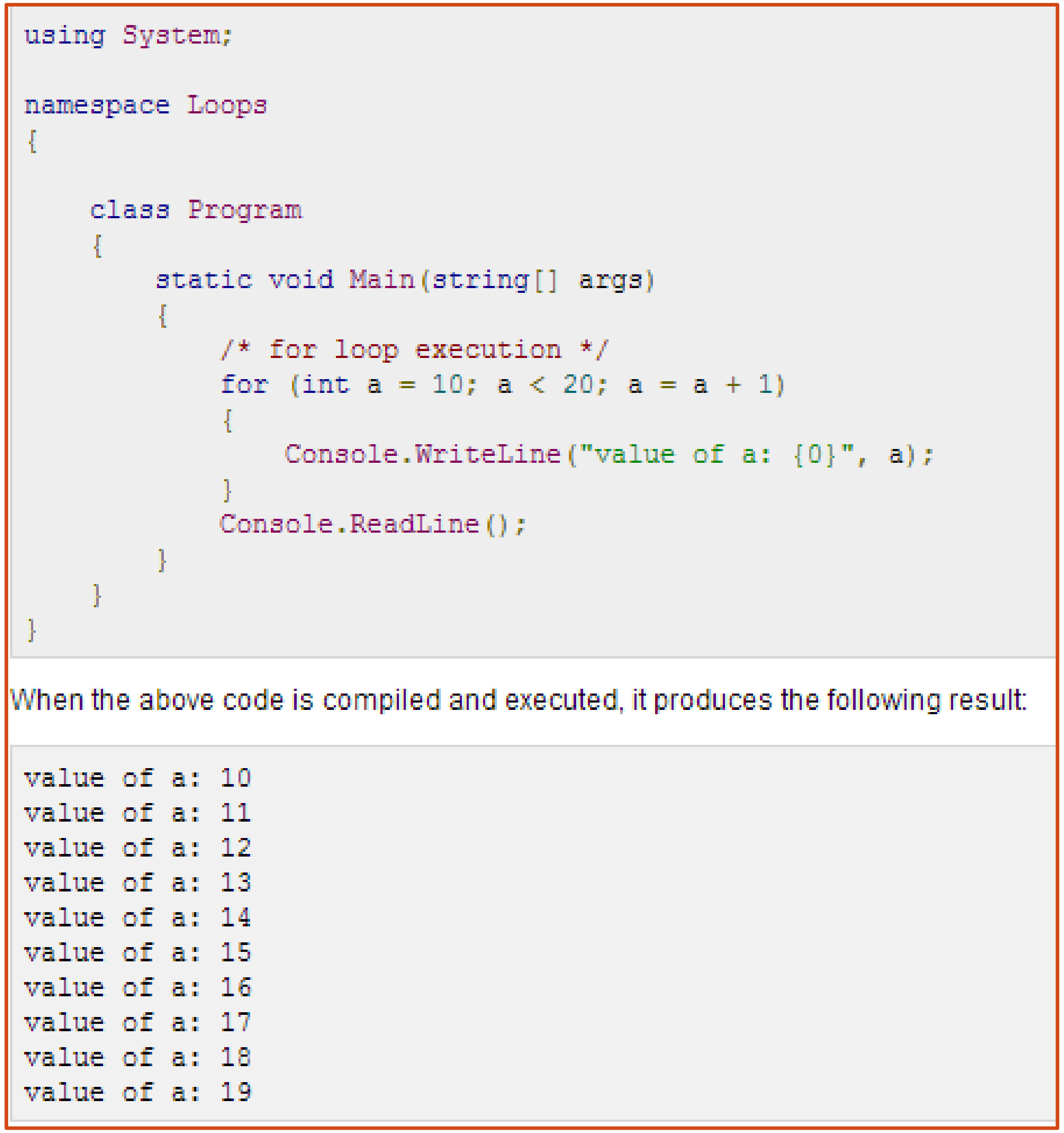
C# - FOR LOOP

- 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.
- The syntax of a for loop in C# is:

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



C# - FOR LOOP



```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* for loop execution */
            for (int a = 10; a < 20; a = a + 1)
            {
                Console.WriteLine("value of a: {0}", a);
            }
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

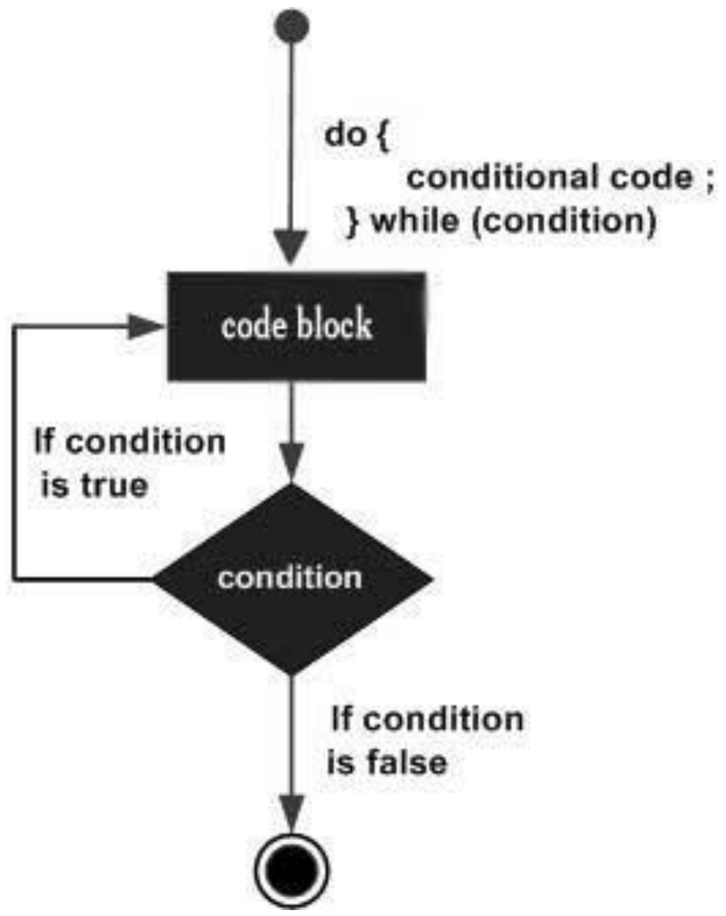
```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

C# - DO...WHILE LOOP

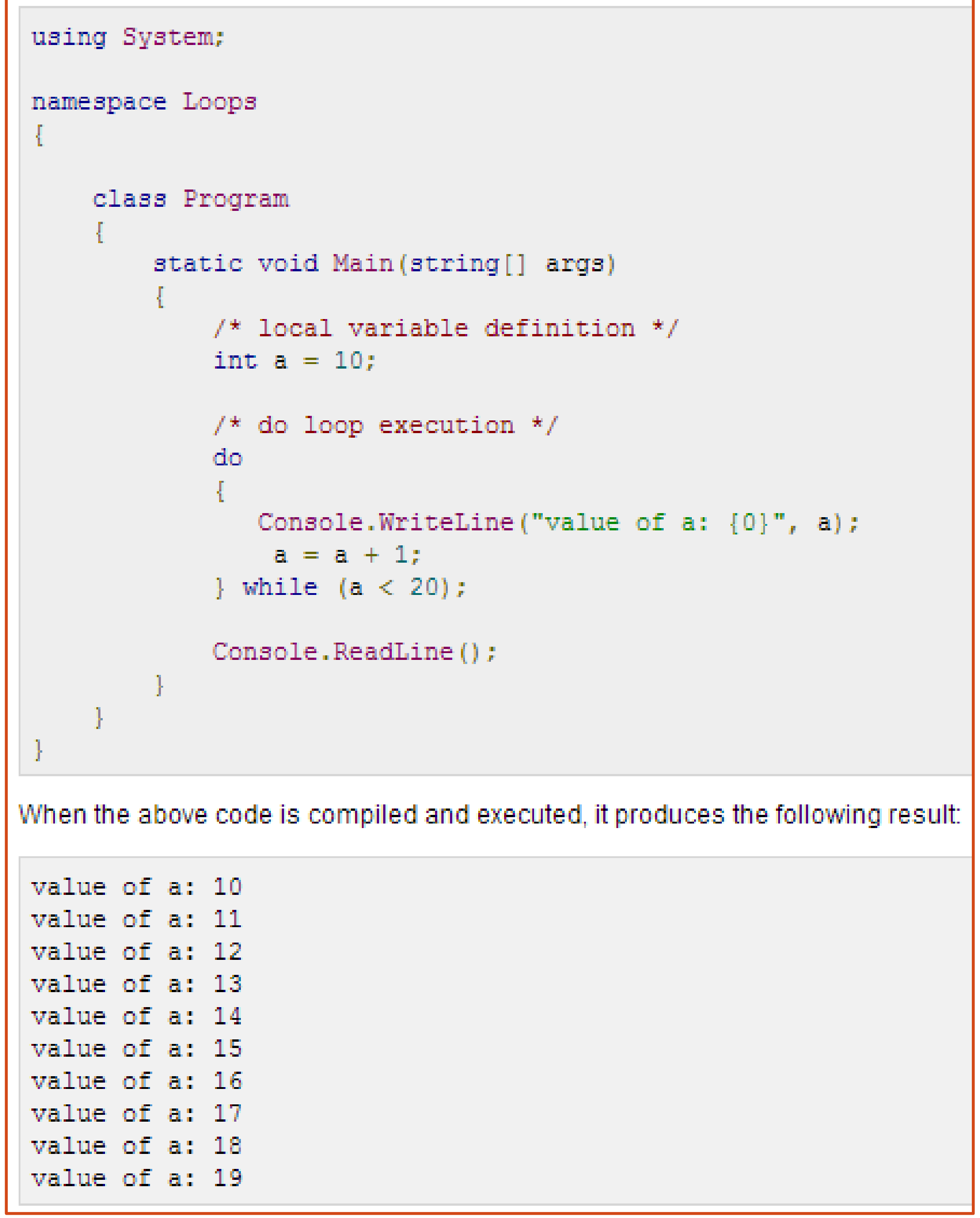
- Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.
- A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.
- The syntax of a do...while loop in C# is:

```
do
{
  statement(s);

}while( condition );
```

C# - DO...WHILE LOOP



C# - DO..WHILE LOOP

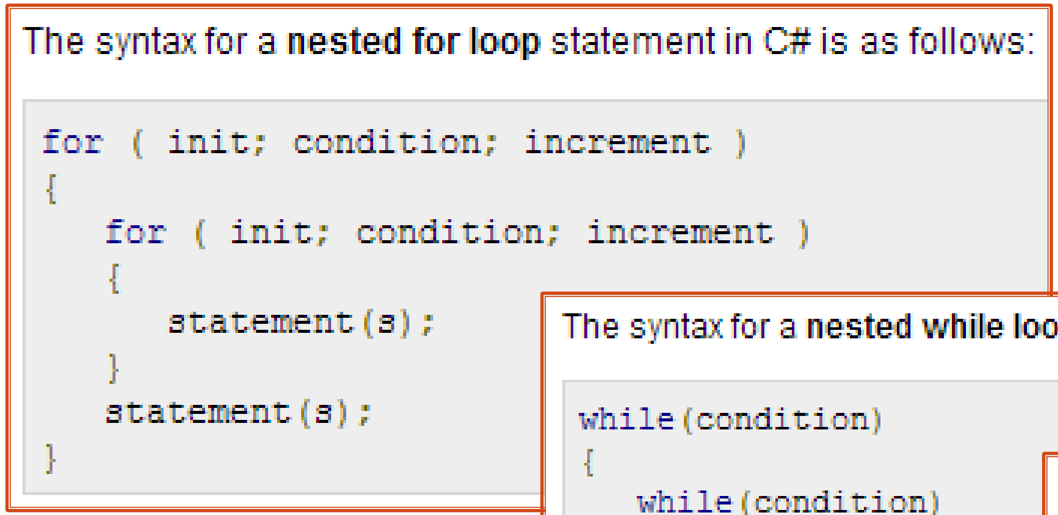


```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 10;

            /* do loop execution */
            do
            {
               Console.WriteLine("value of a: {0}", a);
                a = a + 1;
            } while (a < 20);

            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

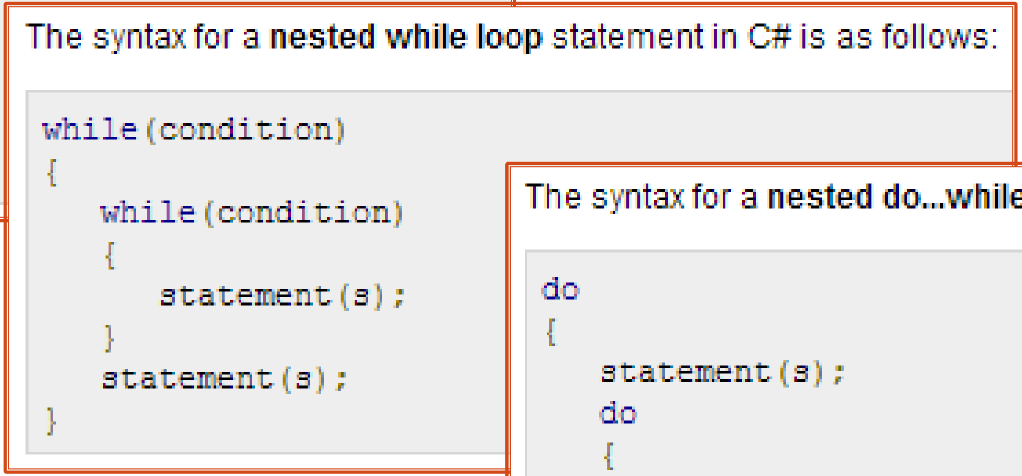
```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

C# - NESTED LOOPS



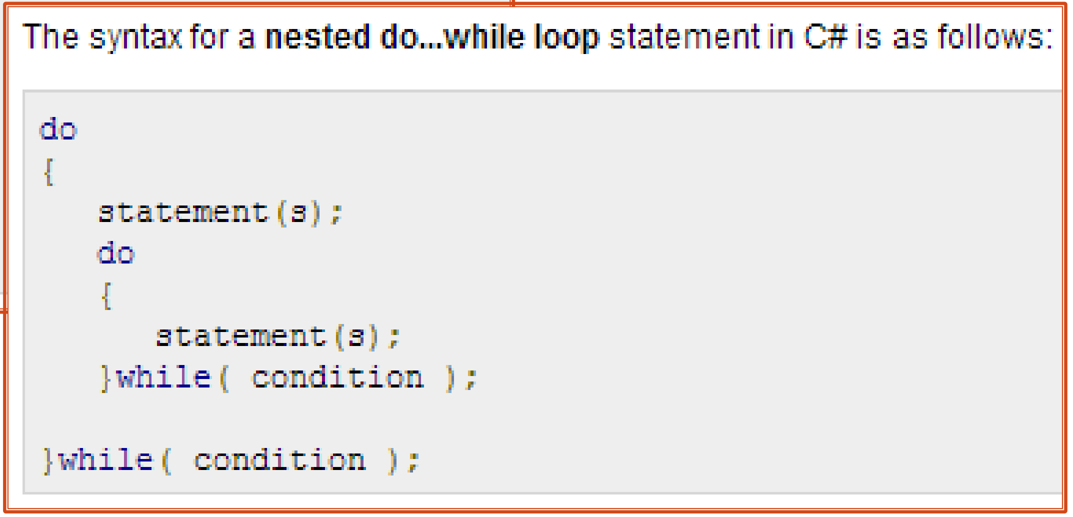
The syntax for a **nested for loop** statement in C# is as follows:

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



The syntax for a **nested while loop** statement in C# is as follows:

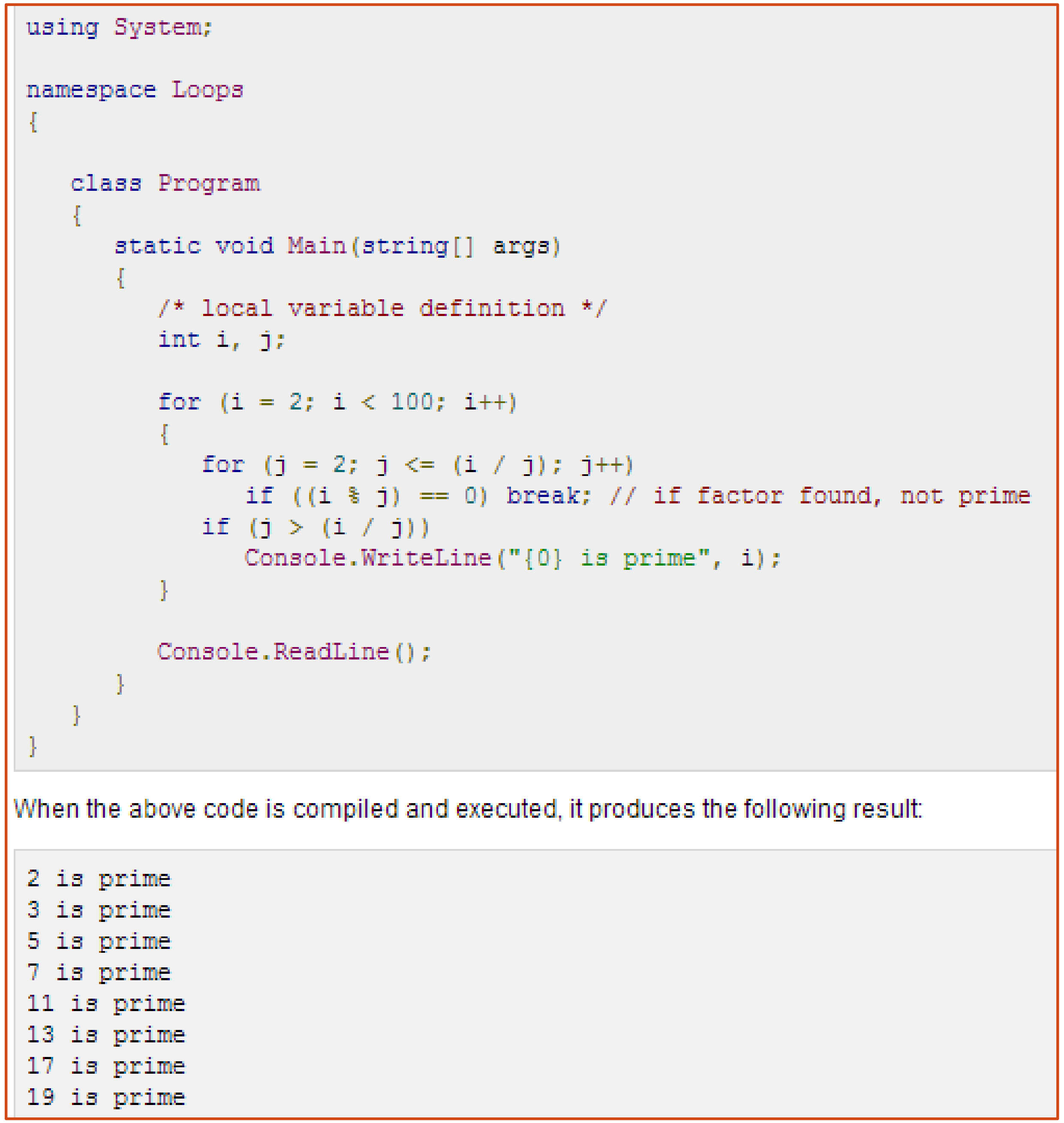
```
while(condition)
{
   while(condition)
   {
      statement(s);
   }
   statement(s);
}
```



The syntax for a **nested do...while loop** statement in C# is as follows:

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

}while( condition );
```

C# - NESTED LOOPS



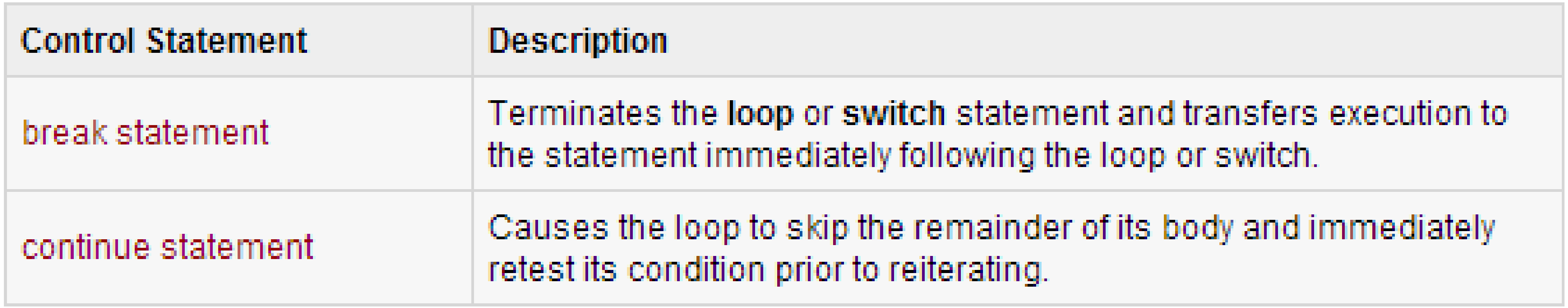
```
using System;

namespace Loops
{

   class Program
   {
      static void Main(string[] args)
      {
         /* local variable definition */
         int i, j;

         for (i = 2; i < 100; i++)
         {
            for (j = 2; j <= (i / j); j++)
               if ((i % j) == 0) break; // if factor found, not prime
            if (j > (i / j))
               Console.WriteLine("{0} is prime", i);
         }

         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
```

LOOP CONTROL STATEMENTS

- Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- C# provides the following control statements. Click the following links to check their details.

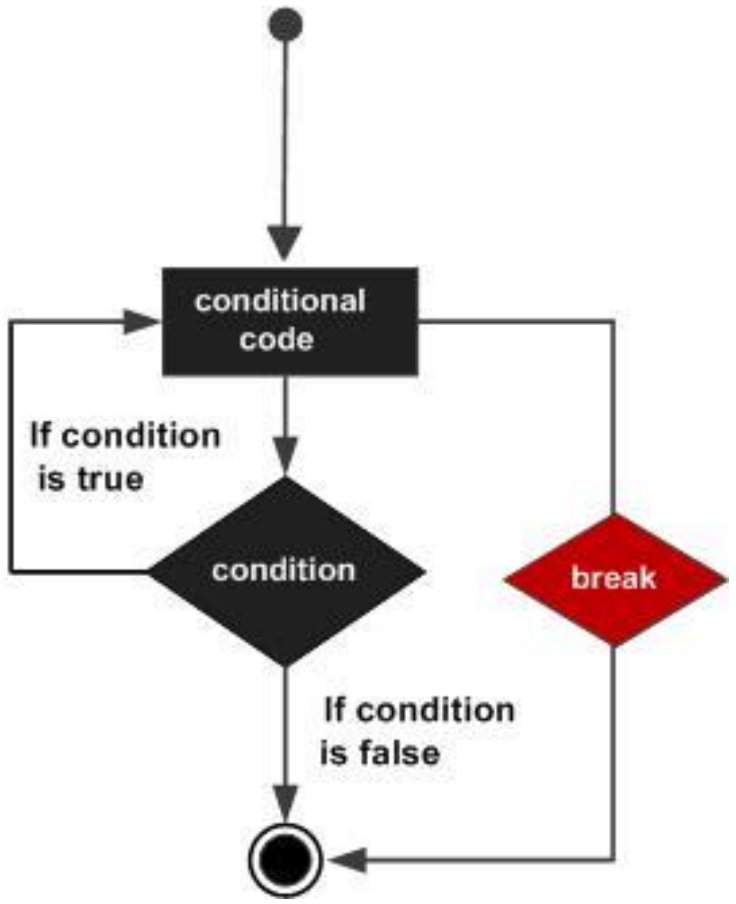


Control Statement	Description
break statement	Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

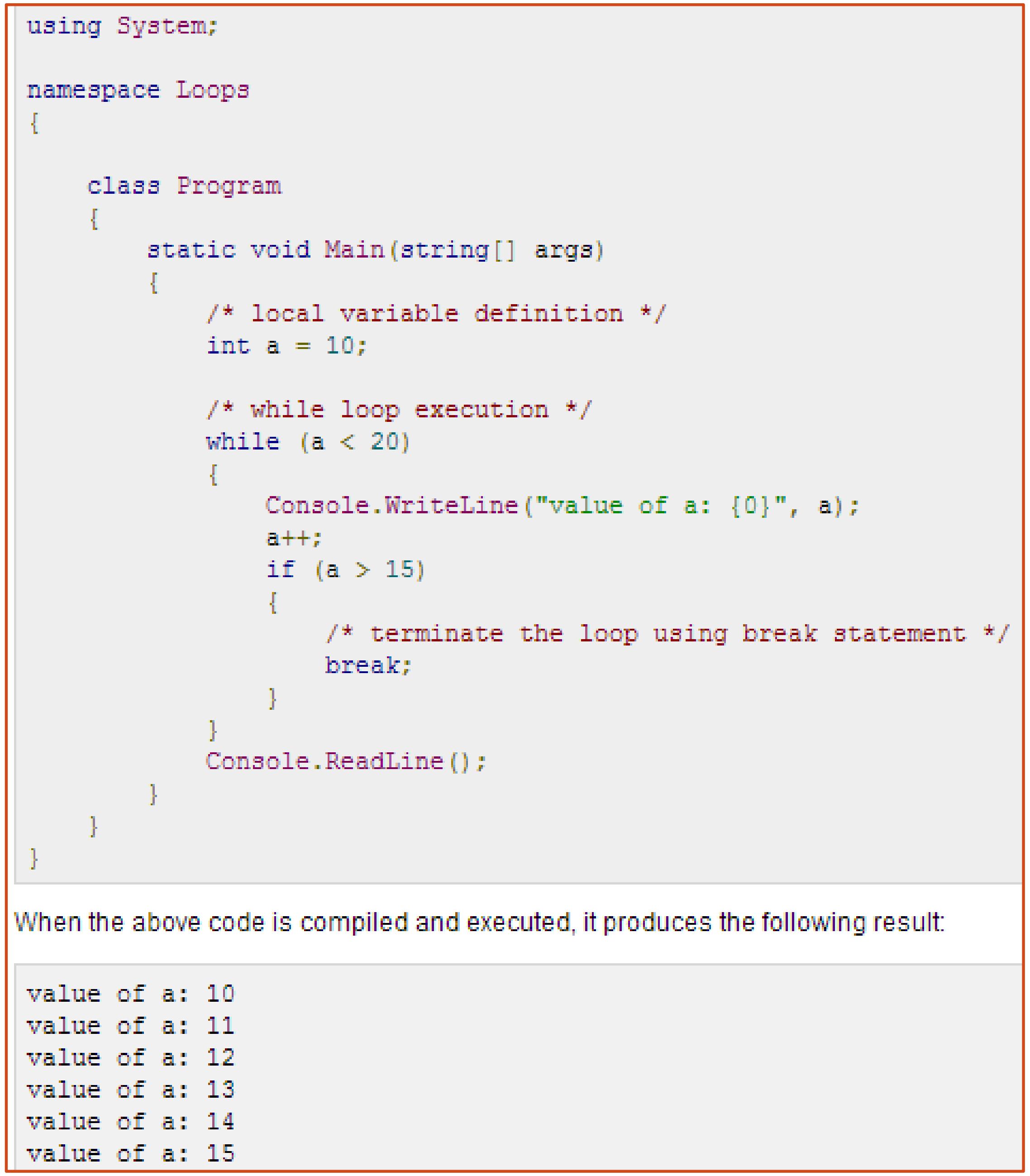
C# - BREAK STATEMENT

- The **break** statement in C# has following two usage:
    1. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
    2. It can be used to terminate a case in the **switch** statement.
- If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

C# - BREAK STATEMENT



C# - BREAK STATEMENT



```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 10;

            /* while loop execution */
            while (a < 20)
            {
                Console.WriteLine("value of a: {0}", a);
                a++;
                if (a > 15)
                {
                    /* terminate the loop using break statement */
                    break;
                }
            }
            Console.ReadLine();
        }
    }
}
```

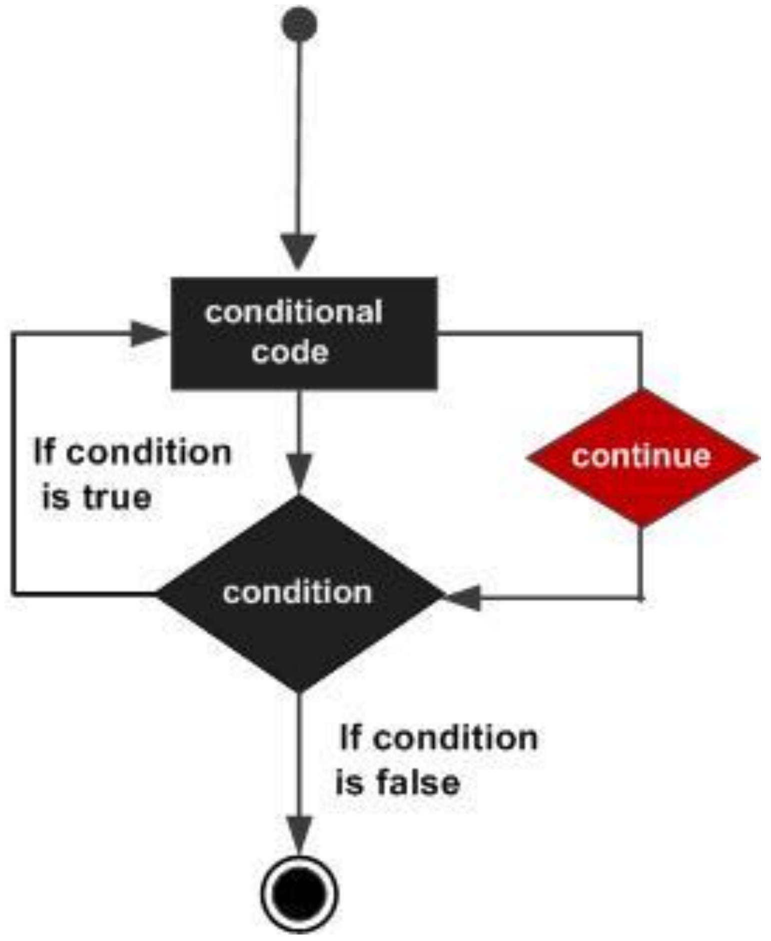
When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

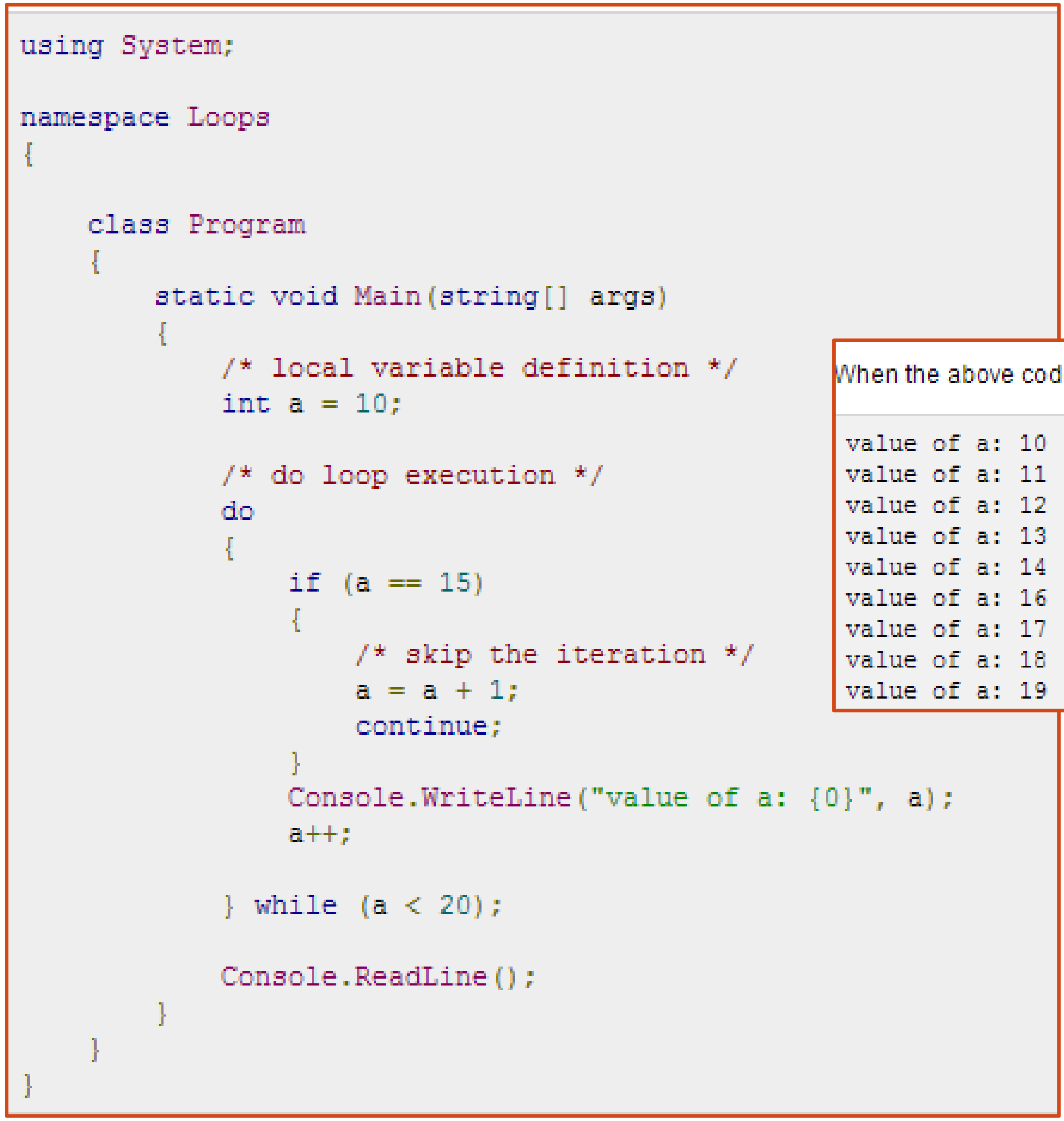
C# - CONTINUE STATEMENT

- The **continue** statement in C# works somewhat like the **break** statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.
- For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control passes to the conditional tests.

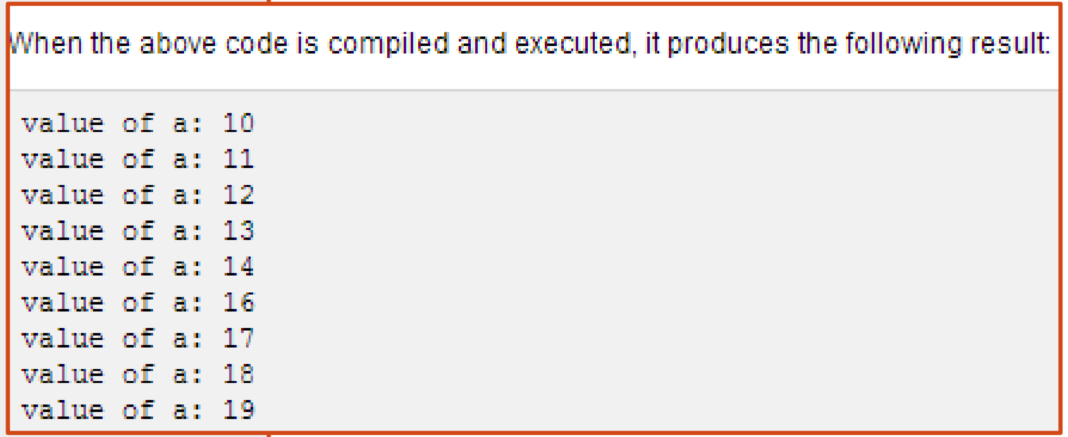
C# - CONTINUE STATEMENT



C# - CONTINUE STATEMENT



```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 10;

            /* do loop execution */
            do
            {
                if (a == 15)
                {
                    /* skip the iteration */
                    a = a + 1;
                    continue;
                }
                Console.WriteLine("value of a: {0}", a);
                a++;

            } while (a < 20);

            Console.ReadLine();
        }
    }
}
```

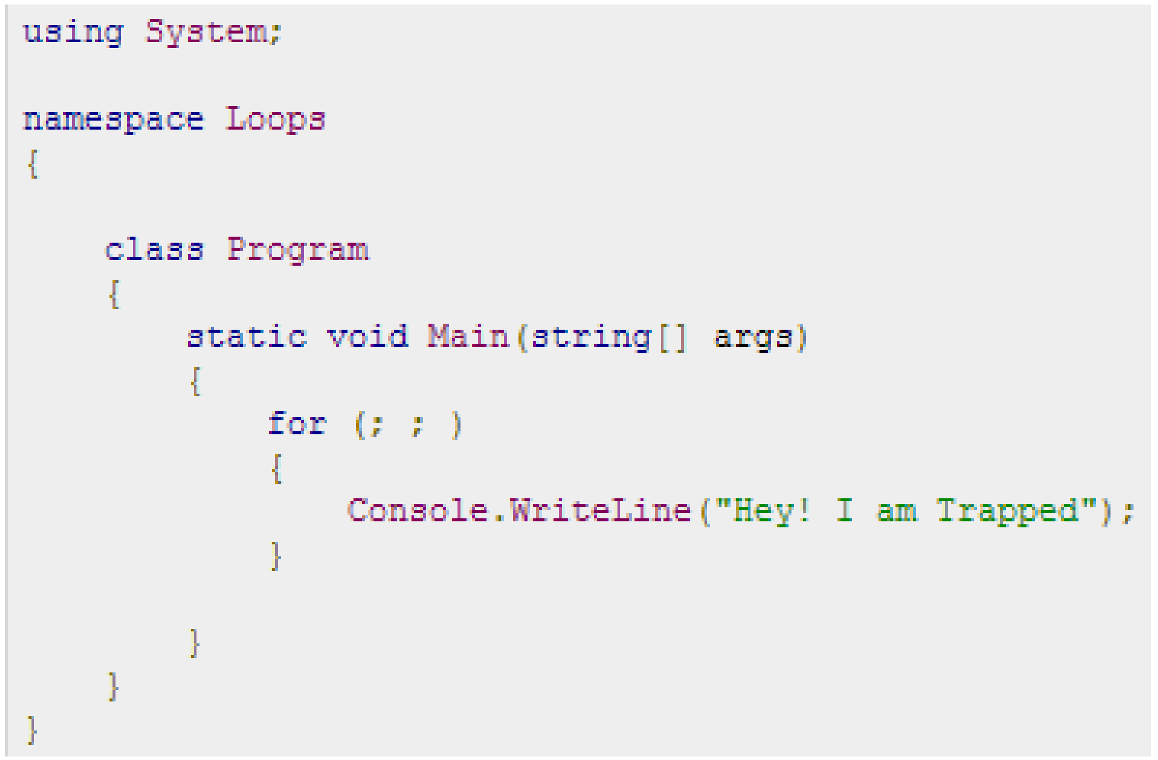


When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

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.



```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            for (; ; )
            {
                Console.WriteLine("Hey! I am Trapped");
            }

        }
    }
}
```

Q
A

Thank You!