P.G.DEPARTMENT OF COMPUTER SCIENCE

THE NEW COLLEGE(AUTONOMOUS)

PROGRAMMING IN ASP.NET

E-NOTES – UNIT I

C# 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, are said to be in the same class.

For example, let us consider a Rectangle object. It has attributes such as length and width. Depending upon the design, it may need ways for accepting the values of these attributes, calculating the area, and displaying details.

Let us look at implementation of a Rectangle class and discuss C# basic syntax

```
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.

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*.

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*.

Instantiating a Class

In the preceding program, the class *ExecuteRectangle* contains the *Main()* method and instantiates the *Rectangle* class.

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 such as? - + ! @ # % ^ & * ( ) [ ] { } . ; : " ' / 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 are called contextual keywords.

The following table lists the reserved keywords and contextual keywords in C# −

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	override	params

				modifier)		
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			

Variable

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.

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

C# also allows defining other value types of variable such as **enum** and reference types of variables such as **class**, which we will cover in subsequent chapters.

Defining Variables

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, 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;
```

Initializing Variables

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 in their declaration. The initializer consists of an equal sign followed by a constant expression as −

<data_type> <variable_name> = value;

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 may produce unexpected result.

The following example uses various types of variables −

```
using System;

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.

Lvalue and Rvalue Expressions 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 hence they may appear on the left-hand side of an assignment. Numeric literals are rvalues and hence they may not be assigned and can not appear on the left-hand side. Following is a valid C# statement −

```
int g = 20;
```

But following is not a valid statement and would generate compile-time error −

```
10 = 20;
```

Data types:

The variables in C#, are categorized into the following types −

- Value types
- Reference types
- Pointer types

Value Type

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, and float**, which stores numbers, alphabets, and 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 −

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	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / 10^{0}$ to 28	0.0M
double	64-bit double-precision floating point type	$(+/-)5.0 \times 10^{-324}$ to $(+/-)1.7 \times 10^{308}$	0.0D
float	32-bit single-precision floating point type	-3.4×10^{38} to $+ 3.4 \times 10^{38}$	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-9,223,372,036,854,775,808 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

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 −

```
using System;

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 Type

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 multiple variables, 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**.

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. The 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**.

```
object obj;
obj = 100; // this is boxing
```

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 takes place at run time.

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 as follows −

```
@"Tutorials Point";
```

The user-defined reference types are: class, interface, or delegate. We will discuss these types in later chapter.

Pointer Type

Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.

Syntax for declaring a pointer type is −

```
type* identifier;
```

For example,

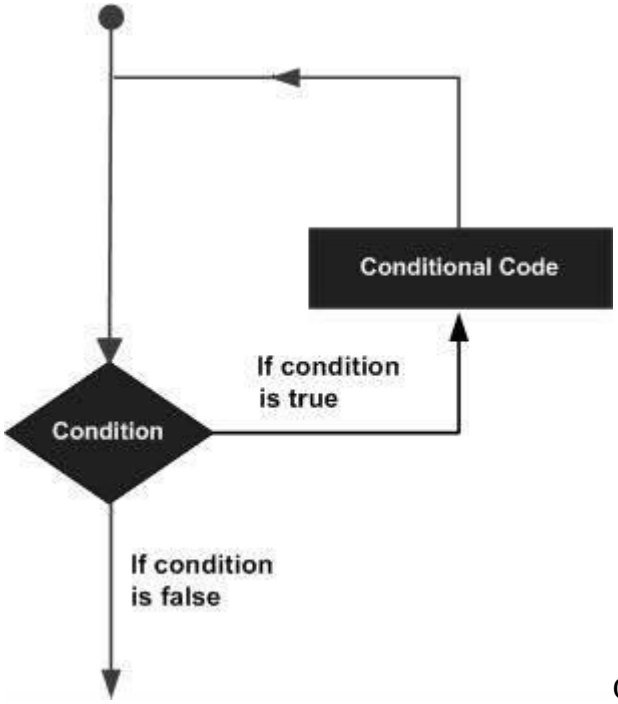
```
char* cptr;
int* iptr;
```

Looping Structure

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

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or a group of statements multiple times and following is the general from of a loop statement in most of the programming languages



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

Loop Type & Description	
1	while loop It repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop

	It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop It is similar to a while statement, except that it tests the condition at the end of the loop body
4	nested loops You can use one or more loop inside any another while, for or do..while loop.

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.

Sr.No.	Control Statement & Description
1	break statement Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

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.

Example

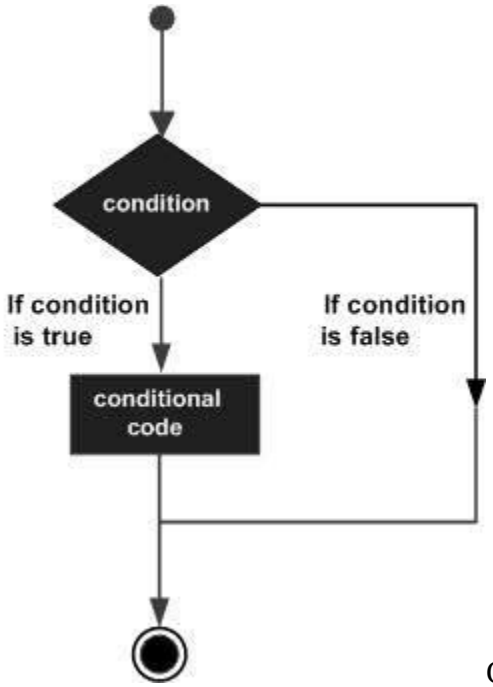
```
using System;

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

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but programmers more commonly use the for(;;) construct to signify an infinite loop.

Decision Structure:

Decision making structures requires the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



C# provides following types of decision making statements. Click the following links to check their detail..

Sr.No.	Statement & Description
1	if statement An **if statement** consists of a boolean expression followed by one or more statements.
2	if...else statement An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
3	nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
4	switch statement A **switch** statement allows a variable to be tested for equality against a list of values.
5	nested switch statements You can use one **switch** statement inside another **switch** statement(s).

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 as follows: 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.

Switch Case Statement:

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

Syntax

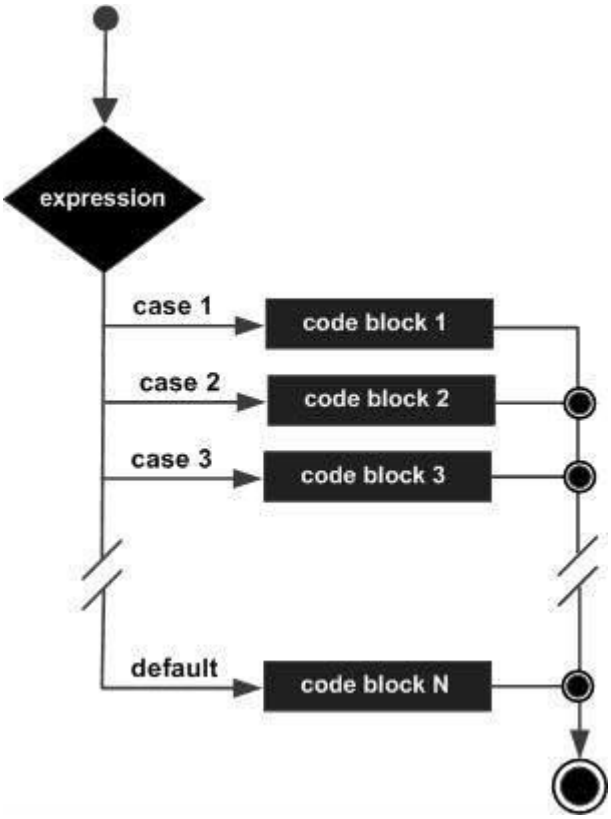
The syntax for a **switch** statement in C# is as follows −

```
switch(expression) {
  case constant-expression1  :
    statement(s);
    break;
  case constant-expression2  :
  case constant-expression3  :
    statement(s);
    break;

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

The following rules apply to a **switch** statement −

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

Flow Diagram



Example

```
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
```

Exception Handling

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

- **try** − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.

Syntax

Assuming a block raises an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

```
try {
   // statements causing exception
} catch( ExceptionName e1 ) {
   // error handling code
} catch( ExceptionName e2 ) {
   // error handling code
} catch( ExceptionName eN ) {
   // error handling code
} finally {
   // statements to be executed
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Exception Classes in C#

C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.

The **System.ApplicationException** class supports exceptions generated by application programs. Hence the exceptions defined by the programmers should derive from this class.

The **System.SystemException** class is the base class for all predefined system exception.

The following table provides some of the predefined exception classes derived from the Sytem.SystemException class −

Sr.No.	Exception Class & Description
1	**System.IO.IOException** Handles I/O errors.
2	**System.IndexOutOfRangeException** Handles errors generated when a method refers to an array index out of range.
3	**System.ArrayTypeMismatchException** Handles errors generated when type is mismatched with the array type.

4	**System.NullReferenceException** Handles errors generated from referencing a null object.
5	**System.DivideByZeroException** Handles errors generated from dividing a dividend with zero.
6	**System.InvalidCastException** Handles errors generated during typecasting.
7	**System.OutOfMemoryException** Handles errors generated from insufficient free memory.
8	**System.StackOverflowException** Handles errors generated from stack overflow.

Handling Exceptions

C# provides a structured solution to the exception handling in the form of try and catch blocks. Using these blocks the core program statements are separated from the error-handling statements.

These error handling blocks are implemented using the **try**, **catch**, and **finally** keywords. Following is an example of throwing an exception when dividing by zero condition occurs −

```
using System;

namespace ErrorHandlingApplication {
  class DivNumbers {
    int result;

    DivNumbers() {
      result = 0;
    }
    public void division(int num1, int num2) {
      try {
        result = num1 / num2;
      } catch (DivideByZeroException e) {
        Console.WriteLine("Exception caught: {0}", e);
      } finally {
```

```
         Console.WriteLine("Result: {0}", result);
      }
   }
   static void Main(string[] args) {
      DivNumbers d = new DivNumbers();
      d.division(25, 0);
      Console.ReadKey();
   }
  }
}
```

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

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at ...
Result: 0
```

Creating User-Defined Exceptions

You can also define your own exception. User-defined exception classes are derived from the **Exception** class. The following example demonstrates this −

```
using System;

namespace UserDefinedException {
  class TestTemperature {
    static void Main(string[] args) {
      Temperature temp = new Temperature();
      try {
        temp.showTemp();
      } catch(TempIsZeroException e) {
        Console.WriteLine("TempIsZeroException: {0}", e.Message);
      }
      Console.ReadKey();
    }
  }
}
public class TempIsZeroException: Exception {
  public TempIsZeroException(string message): base(message) {
  }
}
public class Temperature {
  int temperature = 0;

  public void showTemp() {

    if(temperature == 0) {
      throw (new TempIsZeroException("Zero Temperature found"));
```

```
    } else {
      Console.WriteLine("Temperature: {0}", temperature);
    }
  }
}
```

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

```
TempIsZeroException: Zero Temperature found
```

Throwing Objects

You can throw an object if it is either directly or indirectly derived from the **System.Exception** class. You can use a throw statement in the catch block to throw the present object as −

```
Catch(Exception e) {
  ...
  Throw e
}
```

Constants:

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.

The constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, and there is no prefix id 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.

Here are some examples of integer literals −

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
```

Following are other examples of various types of Integer literals −

```
85         /* decimal */
0x4b       /* hexadecimal */
30         /* int */
```

```
30u       /* unsigned int */
30l       /* long */
30ul      /* 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 */
314159E-5F   /* Legal */
510E         /* Illegal: incomplete exponent */
210f         /* Illegal: no decimal or exponent */
.e55         /* Illegal: missing integer or fraction */
```

While representing in 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. For example, 'x' and can be stored in a simple variable of char type. A character literal can be a plain character (such as 'x'), an escape sequence (such as '\t'), or a universal character (such as '\u02C0').

There are certain characters in C# when they are preceded by a backslash. They have special meaning and they are used to represent like newline (\n) or tab (\t). Here, is a list of some of such escape sequence codes −

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
\xhh . . .	Hexadecimal number of one or more digits

```
using System;

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

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.

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();
    }
  }
}
```

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

```
Enter Radius:
3
Radius: 3, Area: 28.27431
```

Properties:

Properties are named members of classes, structures, and interfaces. Member variables or methods in a class or structures are called **Fields**. Properties are an extension of fields and are accessed using the same syntax. They use **accessors** through which the values of the private fields can be read, written or manipulated.

Properties do not name the storage locations. Instead, they have **accessors** that read, write, or compute their values.

For example, let us have a class named Student, with private fields for age, name, and code. We cannot directly access these fields from outside the class scope, but we can have properties for accessing these private fields.

Accessors

The **accessor** of a property contains the executable statements that helps in getting (reading or computing) or setting (writing) the property. The accessor declarations can contain a get accessor, a set accessor, or both. For example

```
// Declare a Code property of type string:
public string Code {
   get {
      return code;
   }
   set {
      code = value;
   }
}

// Declare a Name property of type string:
public string Name {
   get {
      return name;
   }
   set {
      name = value;
   }
}

// Declare a Age property of type int:
public int Age {
   get {
      return age;
   }
   set {
      age = value;
   }
}
```

Abstract Properties

An abstract class may have an abstract property, which should be implemented in the derived class

Methods:

A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main.

To use a method, you need to −

- Define the method
- Call the method

Defining Methods in C#

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

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

Following are the various elements of a method −

- **Access Specifier** − This determines the visibility of a variable or a method from another class.
- **Return type** − A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is **void**.
- **Method name** − Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
- **Parameter list** − Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
- **Method body** − This contains the set of instructions needed to complete the required activity.

Example

Following code snippet shows a function *FindMax* that takes two integer values and returns the larger of the two. It has public access specifier, so it can be accessed from outside the class using an instance of the class.

```
class NumberManipulator {

  public int FindMax(int num1, int num2) {
    /* local variable declaration */
    int result;

    if (num1 > num2)
      result = num1;
    else
      result = num2;

    return result;
  }
  ...
}
```

Calling Methods in C#

You can call a method using the name of the method. The following example illustrates this −

```
using System;

namespace CalculatorApplication {
  class NumberManipulator {
    public int FindMax(int num1, int num2) {
      /* local variable declaration */
      int result;

      if (num1 > num2)
        result = num1;
      else
        result = num2;
      return result;
    }

    static void Main(string[] args) {
      /* local variable definition */
      int a = 100;
      int b = 200;
      int ret;
      NumberManipulator n = new NumberManipulator();
```

```
      //calling the FindMax method
      ret = n.FindMax(a, b);
      Console.WriteLine("Max value is : {0}", ret );
      Console.ReadLine();
    }
  }
}
```

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

```
Max value is : 200
```

You can also call public method from other classes by using the instance of the class. For example, the method *FindMax* belongs to the *NumberManipulator* class, you can call it from another class *Test*.

```
using System;

namespace CalculatorApplication {
  class NumberManipulator {
    public int FindMax(int num1, int num2) {
      /* local variable declaration */
      int result;

      if(num1 > num2)
        result = num1;
      else
        result = num2;

      return result;
    }
  }
  class Test {
    static void Main(string[] args) {
      /* local variable definition */
      int a = 100;
      int b = 200;
      int ret;
      NumberManipulator n = new NumberManipulator();

      //calling the FindMax method
      ret = n.FindMax(a, b);
      Console.WriteLine("Max value is : {0}", ret );
      Console.ReadLine();
    }
  }
}
```

Recursive Method Call

A method can call itself. This is known as **recursion**. Following is an example that calculates factorial for a given number using a recursive function −

Passing Parameters to a Method

When method with parameters is called, you need to pass the parameters to the method. There are three ways that parameters can be passed to a method −

Sr.No.	Mechanism & Description
1	Value parameters This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Reference parameters This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument.
3	Output parameters This method helps in returning more than one value.

C# Constructors

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor has exactly the same name as that of class and it does not have any return type. Following example explains the concept of constructor −

```
using System;

namespace LineApplication {
  class Line {
    private double length;   // Length of a line

    public Line() {
      Console.WriteLine("Object is being created");
    }
    public void setLength( double len ) {
```

```
      length = len;
    }
    public double getLength() {
      return length;
    }

    static void Main(string[] args) {
      Line line = new Line();

      // set line length
      line.setLength(6.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
      Console.ReadKey();
    }
  }
}
```

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

```
Object is being created
Length of line : 6
```

A **default constructor** does not have any parameter but if you need, a constructor can have parameters. Such constructors are called **parameterized constructors**. This technique helps you to assign initial value to an object at the time of its creation as shown in the following example

```
using System;

namespace LineApplication {
  class Line {
    private double length;   // Length of a line

    public Line(double len) {  //Parameterized constructor
      Console.WriteLine("Object is being created, length = {0}", len);
      length = len;
    }
    public void setLength( double len ) {
      length = len;
    }
    public double getLength() {
      return length;
    }
    static void Main(string[] args) {
      Line line = new Line(10.0);
      Console.WriteLine("Length of line : {0}", line.getLength());

      // set line length
```

```
      line.setLength(6.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
      Console.ReadKey();
    }
  }
}
```

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

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

C# Destructors

A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope. A **destructor** has exactly the same name as that of the class with a prefixed tilde (~) and it can neither return a value nor can it take any parameters.

Destructor can be very useful for releasing memory resources before exiting the program. Destructors cannot be inherited or overloaded.

Following example explains the concept of destructor −

```
using System;

namespace LineApplication {
  class Line {
    private double length;   // Length of a line

    public Line() {   // constructor
      Console.WriteLine("Object is being created");
    }
    ~Line() {   //destructor
      Console.WriteLine("Object is being deleted");
    }
    public void setLength( double len ) {
      length = len;
    }
    public double getLength() {
      return length;
    }
    static void Main(string[] args) {
      Line line = new Line();

      // set line length
      line.setLength(6.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
```

```
    }
  }
}
```

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

```
Object is being created
Length of line : 6
Object is being deleted
```