1. OVERVIEW

C# is a modern, general-purpose, object-oriented programming language developed by Microsoft and approved by European Computer Manufacturers Association (ECMA) and International Standards Organization (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 on different computer platforms and architectures.

The following reasons make C# a widely used professional language:

- It is a modern, general-purpose programming language
- It is object oriented.
- It is component oriented.
- It is easy to learn.
- It is a tructured language.
- It produces efficient programs.
- It can be compiled on a variety of computer platforms.
- It is a 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 a number of programmers worldwide.

Following is the list of few important features of C#:

- Boolean Conditions
- Automatic Garbage Collection
- Standard Library
- Assembly Versioning
- Properties and Events
- Delegates and Events Management
- Easy-to-use Generics
- Indexers

```
            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

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. Compiler ignores 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. They are 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* is used as a class, which contains the *Main()* method and instantiates the *Rectangle* class.

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:

```
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

6. TYPE CONVERSION

Type conversion is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms:

- **Implicit type conversion** - These conversions are performed by C# in a type-safe manner. For example, 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.

The following example shows an explicit type conversion:

```
using System;

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:

8. 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.

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, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and 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 */

078         /* Illegal: 8 is not an octal digit */

032UU       /* Illegal: cannot repeat a suffix */
```

Following are other examples of various types of Integer literals:

```
85         /* decimal */

0213       /* octal */

0x4b       /* hexadecimal */

30         /* int */

30u        /* unsigned int */

30l        /* long */

30ul       /* unsigned long */
```

\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

Following is the example to show few escape sequence characters:

```
using System;

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.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"

"hello, \
```

```
      else
      {
          Console.WriteLine("Line 2 - a is not less than b");
      }
      if (a > b)
      {
          Console.WriteLine("Line 3 - a is greater than b");
      }
      else
      {
          Console.WriteLine("Line 3 - a is not greater than b");
      }
      /* Lets change value of a and b */
      a = 5;
      b = 20;
      if (a <= b)
      {
          Console.WriteLine("Line 4 - a is either less than or equal to  b");
      }
      if (b >= a)
      {
         Console.WriteLine("Line 5-b is either greater than or equal to b");
      }
  }
}
```


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

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
```

```
Line 3 - a is greater than b

Line 4 - a is either less than or equal to b

Line 5 - b is either greater than or equal to b
```

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.

Example

The following example demonstrates all the logical operators available in C#:

```
using System;


namespace OperatorsAppl

{

    class Program

    {

        static void Main(string[] args)

        {

            bool a = true;

            bool b = true;
```

&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = 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 = 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 = 15, which is 0000 1111

Example

The following example demonstrates all the bitwise operators available in C#:

```
using System;

namespace OperatorsAppl

{

    class Program

    {

        static void Main(string[] args)

        {

            int a = 60;            /* 60 = 0011 1100 */

            int b = 13;            /* 13 = 0000 1101 */

            int c = 0;
```

```
            c = a & b;           /* 12 = 0000 1100 */
            Console.WriteLine("Line 1 - Value of c is {0}", c );

            c = a | b;           /* 61 = 0011 1101 */
            Console.WriteLine("Line 2 - Value of c is {0}", c);

            c = a ^ b;           /* 49 = 0011 0001 */
            Console.WriteLine("Line 3 - Value of c is {0}", c);

            c = ~a;               /*-61 = 1100 0011 */
            Console.WriteLine("Line 4 - Value of c is {0}", c);

            c = a << 2;     /* 240 = 1111 0000 */
            Console.WriteLine("Line 5 - Value of c is {0}", c);

            c = a >> 2;     /* 15 = 0000 1111 */
            Console.WriteLine("Line 6 - Value of c is {0}", c);
            Console.ReadLine();
        }
    }
}
```


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

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
```

as	Cast without raising an exception if the cast fails.	Object obj = new StringReader("Hello"); StringReader r = obj as StringReader;

Example

```
using System;


namespace OperatorsAppl

{


   class Program

   {

      static void Main(string[] args)

      {


         /* example of sizeof operator */

         Console.WriteLine("The size of int is {0}", sizeof(int));

         Console.WriteLine("The size of short is {0}", sizeof(short));

         Console.WriteLine("The size of double is {0}", sizeof(double));


         /* example of ternary operator */

         int a, b;

         a = 10;

         b = (a == 1) ? 20 : 30;

         Console.WriteLine("Value of b is {0}", b);


         b = (a == 10) ? 20 : 30;

         Console.WriteLine("Value of b is {0}", b);
```

if...else Statement

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

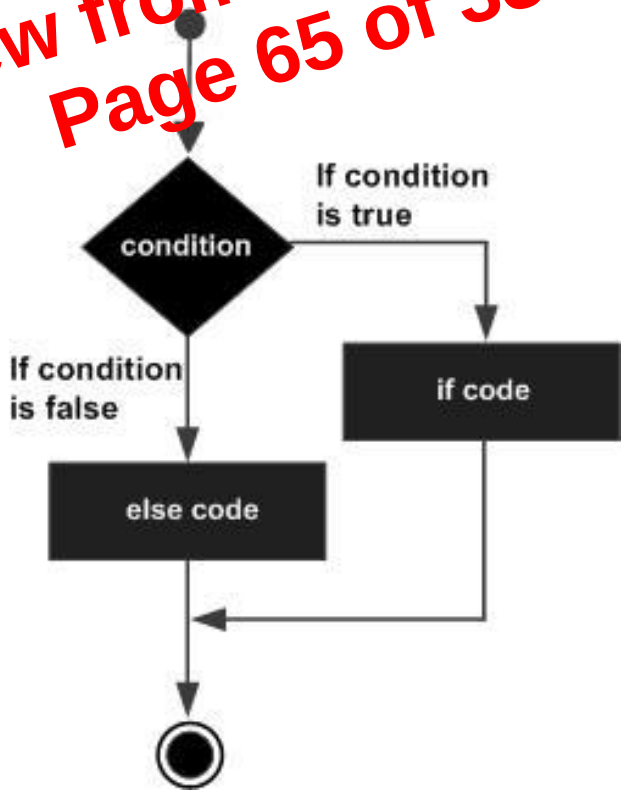
Syntax

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

```
if(boolean_expression)

{

   /* statement(s) will execute if the boolean expression is true */

}

else

{

  /* statement(s) will execute if the boolean expression is false */

}
```

If the boolean expression evaluates to **true**, then the **if block** of code is executed, otherwise **else block** of code is executed.

Flow Diagram



Nested Loops

C# allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax

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 );
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100:

```
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);

         }
```

```
            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
```

In the preceding example, the member variables length and width are declared **public**, so they can be accessed from the function Main() using an instance of the Rectangle class, named **r**.

The member function *Display()* and *GetArea()* can also access these variables directly without using any instance of the class.

The member functions *Display()* is also declared **public**, so it can also be accessed from*Main()* using an instance of the Rectangle class, named **r**.

Private Access Specifier

Private access specifier allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.

The following example illustrates this:

```
using System;


namespace RectangleApplication
{
    class Rectangle
    {
        //member variables
        private double length;
        private double width;


        public void Acceptdetails()
        {
            Console.WriteLine("Enter Length: ");
            length = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Enter Width: ");
            width = Convert.ToDouble(Console.ReadLine());
        }
```

```
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.length = 4.5;
            r.width = 3.5;
            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
```

In the preceding example, notice that the member function *GetArea()* is not declared with any access specifier. Then what would be the default access specifier of a class member if we don't mention any? It is **private**.

Passing Parameters by Reference

A reference parameter is a **reference to a memory location** of a variable. When you pass parameters by reference, unlike value parameters, a new storage location is not created for these parameters. The reference parameters represent the same memory location as the actual parameters that are supplied to the method.

You can declare the reference parameters using the **ref** keyword. The following example demonstrates this:

```
using System;

namespace CalculatorApplication

{

   class NumberManipulator

   {

      public void swap(ref int x, ref int y)

      {

         int temp;



         temp = x; /* save the value of x */

         x = y;    /* put y into x */

         y = temp; /* put temp into y */

       }



      static void Main(string[] args)

      {

         NumberManipulator n = new NumberManipulator();

         /* local variable definition */

         int a = 100;

         int b = 200;



         Console.WriteLine("Before swap, value of a : {0}", a);

         Console.WriteLine("Before swap, value of b : {0}", b);
```

```
namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void getValue(out int x )
      {
         int temp = 5;
         x = temp;
      }

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

         Console.WriteLine("Before method call, value of a : {0}", a);

         /* calling a function to get the value */
         n.getValue(out a);

         Console.WriteLine("After method call, value of a : {0}", a);
         Console.ReadLine();

      }
   }
}
```


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

```
Before method call, value of a : 100

After method call, value of a : 5
```

The variable supplied for the output parameter need not be assigned a value. Output parameters are particularly useful when you need to return values from a method through the parameters without assigning an initial value to the parameter. Go through the following example, to understand this:

```
using System;


namespace CalculatorApplication

{

   class NumberManipulator

   {

      public void getValues(out int x, out int y )

      {

          Console.WriteLine("Enter the first value: ");

          x = Convert.ToInt32(Console.ReadLine());

          Console.WriteLine("Enter the second value: ");

          y = Convert.ToInt32(Console.ReadLine());

      }


      static void Main(string[] args)

      {

         NumberManipulator n = new NumberManipulator();

         /* local variable definition */

         int a , b;


         /* calling a function to get the values */

         n.getValues(out a, out b);


               Console.WriteLine("After method call, value of a : {0}", a);
```

```
using System;

namespace ArrayApplication

{

   class MyArray

   {

      static void Main(string[] args)

      {

         int []  n = new int[10]; /* n is an array of 10 integers */

         int i,j;



         /* initialize elements of array n */

         for ( i = 0; i < 10; i++ )

         {

            n[ i ] = i + 100;

         }

         /* output each array element's value */

         for (j = 0; j < 10; j++ )

         {

            Console.WriteLine("Element[{0}] = {1}", j, n[j]);

         }

         Console.ReadKey();

      }

   }

}
```


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

```
Element[0] = 100
```

```
Element[1] = 101

Element[2] = 102

Element[3] = 103

Element[4] = 104

Element[5] = 105

Element[6] = 106

Element[7] = 107

Element[8] = 108

Element[9] = 109
```

Using the *foreach* Loop

In the previous example, we used a for loop for accessing each array element. You can also use a **foreach** statement to iterate through an array.


```
using System;


namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         int []  n = new int[10]; /* n is an array of 10 integers */



         /* initialize elements of array n */
         for ( int i = 0; i < 10; i++ )
         {
            n[i] = i + 100;
         }
```

Passing Arrays as Function Arguments

You can pass an array as a function argument in C#. The following example demonstrates this:

```
using System;


namespace ArrayApplication
{
   class MyArray
   {
      double getAverage(int[] arr, int size)
      {
         int i;
         double avg;
         int sum = 0;


         for (i = 0; i < size; ++i)
         {
            sum += arr[i];
         }


         avg = (double)sum / size;
         return avg;
      }
      static void Main(string[] args)
      {
         MyArray app = new MyArray();
         /* an int array with 5 elements */
         int [] balance = new int[]{1000, 2, 3, 17, 50};
         double avg;
```

```
            sum += i;
         }
         return sum;
      }
   }

   class TestClass
   {
      static void Main(string[] args)
      {
         ParamArray app = new ParamArray();
         int sum = app.AddElements(512, 720, 250, 567, 889);
         Console.WriteLine("The sum is: {0}", sum);
         Console.ReadKey();
      }
   }
}
```


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

```
The sum is: 2938
```

Array Class

The Array class is the base class for all the arrays in C#. It is defined in the System namespace. The Array class provides various properties and methods to work with arrays.

Properties of the Array Class

The following table describes some of the most commonly used properties of the Array class:

	Concatenates four string objects.
6	**public bool Contains(string value)** Returns a value indicating whether the specified String object occurs within this string.
7	**public static string Copy(string str)** Creates a new String object with the same value as the specified string.
8	**public void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count)** Copies a specified number of characters from a specified position of the String object to a specified position in an array of Unicode characters.
9	**public bool EndsWith(string value)** Determines whether the end of the string object matches the specified string.
10	**public bool Equals(string value)** Determines whether the current String object and the specified String object have the same value.
11	**public static bool Equals(string a, string b)** Determines whether two specified String objects have the same value.
12	**public static string Format(string format, Object arg0)** Replaces one or more format items in a specified string with the string representation of a specified object.
13	**public int IndexOf(char value)** Returns the zero-based index of the first occurrence of the specified Unicode character in the current string.
14	**public int IndexOf(string value)**

```
San Pedro
```

Joining Strings:

```
using System;


namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string[] starray = new string[]{"Down the way nights are dark",
         "And the sun shines daily on the mountain top",
         "I took a trip on a sailing ship",
         "And when I reached Jamaica",
         "I made a stop"};

         string str = String.Join("\n", starray);
         Console.WriteLine(str);
      }
      Console.ReadKey() ;
   }
}
```


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

```
Down the way nights are dark
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
```

```
    {
        // method body
    }
}
```

Note:

- Access specifiers specify the access rules for the members as well as the class itself. If not mentioned, then the default access specifier for a class type is **internal**. Default access for the members is **private**.
- Data type specifies the type of variable, and return type specifies the data type of the data the method returns, if any.
- To access the class members, you use the dot (.) operator.
- The dot operator links the name of an object with the name of a member.

The following example illustrates the concepts discussed so far:

```
using System;

namespace BoxApplication

{

    class Box

    {

       public double length;   // Length of a box

       public double breadth;  // Breadth of a box

       public double height;   // Height of a box

    }

    class Boxtester

    {

        static void Main(string[] args)

        {

            Box Box1 = new Box();        // Declare Box1 of type Box

            Box Box2 = new Box();        // Declare Box2 of type Box

            double volume = 0.0;         // Store the volume of a box here


                    // box 1 specification
```

```
            Box1.height = 5.0;

            Box1.length = 6.0;

            Box1.breadth = 7.0;


            // box 2 specification

            Box2.height = 10.0;

            Box2.length = 12.0;

            Box2.breadth = 13.0;


            // volume of box 1

            volume = Box1.height * Box1.length * Box1.breadth;

            Console.WriteLine("Volume of Box1 : {0}",  volume);


            // volume of box 2

            volume = Box2.height * Box2.length * Box2.breadth;

            Console.WriteLine("Volume of Box2 : {0}", volume);

            Console.ReadKey();
        }
    }
```


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

```
Volume of Box1 : 210

Volume of Box2 : 1560
```

Member Functions and Encapsulation

A member function of a class is a function that has its definition or its prototype within the class definition similar to any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

20. INHERITANCE

One of the most important concepts in object-oriented programming is inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and speeds up implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **IS-A** relationship. For example, mammal **IS A**animal, dog **IS-A** mammal hence dog **IS-A** animal as well, and so on.

Base and Derived Classes

A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base classes or interfaces.

The syntax used in C# for creating derived classes is as follows:

```
<acess-specifier> class <base_class>

{

 ...

}

class <derived_class> : <base_class>

{

 ...

}
```


Consider a base class Shape and its derived class Rectangle:

```
using System;

namespace InheritanceApplication

{

   class Shape

      {
```

```
    {
        return length * width;
    }
    public void Display()
    {
        Console.WriteLine("Length: {0}", length);
        Console.WriteLine("Width: {0}", width);
        Console.WriteLine("Area: {0}", GetArea());
    }
}//end class Rectangle
class Tabletop : Rectangle
{
    private double cost;
    public Tabletop(double l, double w) : base(l, w)
    { }
    public double GetCost()
    {
        double cost;
        cost = GetArea() * 70;
        return cost;
    }
    public void Display()
    {
        base.Display();
        Console.WriteLine("Cost: {0}", GetCost());
    }
}
class ExecuteRectangle
{
```

Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

Here are the rules about abstract classes:

- You cannot create an instance of an abstract class
- You cannot declare an abstract method outside an abstract class
- When a class is declared **sealed**, it cannot be inherited, abstract classes cannot be declared sealed.

The following program demonstrates an abstract class:

```
using System;

namespace PolymorphismApplication

{

   abstract class Shape

   {

      public abstract int area();

   }

   class Rectangle:  Shape

   {

      private int length;

      private int width;

      public Rectangle( int a=0, int b=0)

      {

         length = a;

         width = b;

      }

      public override int area ()

      {

         Console.WriteLine("Rectangle class area :");

         return (width * length);
```

```
}

class Tester
{
   static void Main(string[] args)
   {
      Box Box1 = new Box();          // Declare Box1 of type Box
      Box Box2 = new Box();          // Declare Box2 of type Box
      Box Box3 = new Box();          // Declare Box3 of type Box
      double volume = 0.0;           // Store the volume of a box here

      // box 1 specification
      Box1.setLength(6.0);
      Box1.setBreadth(7.0);
      Box1.setHeight(5.0);

      // box 2 specification
      Box2.setLength(12.0);
      Box2.setBreadth(13.0);
      Box2.setHeight(10.0);

      // volume of box 1
      volume = Box1.getVolume();
      Console.WriteLine("Volume of Box1 : {0}", volume);

      // volume of box 2
      volume = Box2.getVolume();
      Console.WriteLine("Volume of Box2 : {0}", volume);
```

```
    }

    public void setHeight( double hei )
    {
        height = hei;
    }
    // Overload + operator to add two Box objects.
    public static Box operator+ (Box b, Box c)
    {
        Box box = new Box();
        box.length = b.length + c.length;
        box.breadth = b.breadth + c.breadth;
        box.height = b.height + c.height;
        return box;
    }

    public static bool operator ==(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length == rhs.length && lhs.height == rhs.height
           && lhs.breadth == rhs.breadth)
        {
            status = true;
        }
        return status;
    }
    public static bool operator !=(Box lhs, Box rhs)
    {
        bool status = false;
```

{ n }	Matches the previous element exactly n times.	",\d{3}"	",043" in "1,043.6", ",876", ",543", and ",210" in "9,876,543,210"
{ n ,}	Matches the previous element at least n times.	"\d{2,}"	"166", "29", "1930"
{ n , m }	Matches the previous element at least n times, but no more than m times.	"\d{3,5}"	"166", "17668" "19302" in "193024"
***?**	Matches the previous element zero or more times, but as few times as possible.	\d*?\.\d	".0", "19.9", "219.9"
+?	Matches the previous element one or more times, but as few times as possible.	"be+?"	"be" in "been", "be" in "bent"
??	Matches the previous element zero or one time, but as few times as possible.	"ra[in]?"	"ran", "rain"
{ n }?	Matches the preceding element exactly n times.	",\d{3}?"	",043" in "1,043.6", ",876", ",543", and ",210" in "9,876,543,210"
{ n ,}?	Matches the previous element at least n times, but as few times as possible.	"\d{2,}?"	"166", "29", "1930"
{ n , m }?	Matches the previous element between n and m times, but as few times as possible.	"\d{3,5}?"	"166", "17668" "193", "024" in "193024"

Backreference Constructs

Backreference constructs allow a previously matched sub-expression to be identified subsequently in the same regular expression.

The following table lists these constructs:

Backreference construct	Description	Pattern	Matches
\ number	Backreference. Matches the value of a numbered subexpression.	(\w)\1	"ee" in "seek"
\k< name >	Named backreference. Matches the value of a named expression.	(?< char>\w)\k< char>	"ee" in "seek"

Alternation Constructs

Alternation constructs modify a regular expression to enable either/or matching. The following table lists the alternation constructs:

Alternation construct	Description	Pattern	Matches
|	Matches any one element separated by the vertical bar (|) character.	th(e|is|at)	"the", "this" in "this is the day. "
(?(expression)yes | no)	Matches *yes* if expression matches; otherwise, matches the optional *no* part. Expression is interpreted as a zero-width assertion.	(?(A)A\d{2}\b|\b\d{3}\b)	"A10", "910" in "A10 C103 910"
(?(name)yes | no)	Matches *yes* if the named capture name has a match; otherwise, matches the optional *no*.	(?< quoted>")?(?(quoted).+?" |\S+\s)	Dogs.jpg, "Yiska playing.jpg" in "Dogs.jpg "Yiska playing.jpg" "

```
         string str = "make maze and manage to measure it";


         Console.WriteLine("Matching words start with 'm' and ends with 'e':");
         showMatch(str, @"\bm\S*e\b");
         Console.ReadKey();
      }
   }
}
```

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

```
Matching words start with 'm' and ends with 'e':
The Expression: \bm\S*e\b
make
maze
manage
measure
```


Example 3

This example replaces extra white space:

```
using System;
using System.Text.RegularExpressions;


namespace RegExApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         string input = "Hello   World   ";
         string pattern = "\\s+";
```

Manipulating the Windows file system
It gives a C# programamer the ability to browse and locate Windows files and directories.

Reading from and Writing to Text Files

The **StreamReader** and **StreamWriter** classes are used for reading from and writing data to text files. These classes inherit from the abstract base class Stream, which supports reading and writing bytes into a file stream.

The StreamReader Class

The **StreamReader** class also inherits from the abstract base class TextReader that represents a reader for reading series of characters. The following table describes some of the commonly used **methods** of the StreamReader class:

Sr. No.	Methods
1	**public override void Close()** It closes the StreamReader object and the underlying stream, and releases any system resources associated with the reader.
2	**public override int Peek()** Returns the next available character but does not consume it.
3	**public override int Read()** Reads the next character from the input stream and advances the character position by one.

Example

The following example demonstrates reading a text file named Jamaica.txt. The file reads:

```
Down the way where the nights are gay

And the sun shines daily on the mountain top

I took a trip on a sailing ship

And when I reached Jamaica

I made a stop
```

```
                Console.WriteLine(e.Message);
            }
            Console.ReadKey();
        }
    }
}
```

Guess what it displays when you compile and run the program!

The StreamWriter Class

The **StreamWriter** class inherits from the abstract class TextWriter that represents a writer, which can write a series of character.

The following table describes the most commonly used methods of this class:

Sr. No.	Methods
1	**public override void Close()** Closes the current StreamWriter object and the underlying stream.
2	**public override void Flush()** Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream.
3	**public virtual void Write(bool value)** Writes the text representation of a Boolean value to the text string or stream. (Inherited from TextWriter.)
4	**public override void Write(char value)** Writes a character to the stream.
5	**public virtual void Write(decimal value)** Writes the text representation of a decimal value to the text string or stream.

```
                Console.WriteLine(e.Message + "\n Cannot read from file.");
                return;
            }
            br.Close();
            Console.ReadKey();
        }
    }
}
```

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

```
Integer data: 25
Double data: 3.14157
Boolean data: True
String data: I am happy
```


Windows File System

C# allows you to work with the directories and files using various directory and file related classes such as the **DirectoryInfo** class and the **FileInfo** class.

The DirectoryInfo Class

The **DirectoryInfo** class is derived from the **FileSystemInfo** class. It has various methods for creating, moving, and browsing through directories and subdirectories. This class cannot be inherited.

Following are some commonly used **properties** of the **DirectoryInfo** class:

Sr. No.	Properties
1	**Attributes** Gets the attributes for the current file or directory.
2	**CreationTime** Gets the creation time of the current file or directory.

```
            name = value;
         }
      }

      // Declare a Age property of type int:
      public int Age
      {
         get
         {
            return age;
         }
         set
         {
            age = value;
         }
      }
      public override string ToString()
      {
         return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
      }
   }
   class ExampleDemo
   {
      public static void Main()
      {
         // Create a new Student object:
         Student s = new Student();

         // Setting code, name and the age of the student
```

```
            Console.WriteLine(names[i]);
         }
         //using the second indexer with the string parameter
         Console.WriteLine(names["Nuha"]);
         Console.ReadKey();
      }
   }
}
```

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

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A
N. A.
N. A.
2
```

```
      {
         num += p;
         return num;
      }

      public static int MultNum(int q)
      {
         num *= q;
         return num;
      }
      public static int getNum()
      {
         return num;
      }

      static void Main(string[] args)
      {
         //create delegate instances
         NumberChanger nc;
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultNum);
         nc = nc1;
         nc += nc2;
         //calling multicast
         nc(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
```

```
            Console.WriteLine("Adding some numbers:");
            al.Add(45);
            al.Add(78);
            al.Add(33);
            al.Add(56);
            al.Add(12);
            al.Add(23);
            al.Add(9);

            Console.WriteLine("Capacity: {0} ", al.Capacity);
            Console.WriteLine("Count: {0}", al.Count);

            Console.Write("Content: ");
            foreach (int i in al)
            {
                Console.Write(i + " ");
            }
            Console.WriteLine();
            Console.Write("Sorted Content: ");
            al.Sort();
            foreach (int i in al)
            {
                Console.Write(i + " ");
            }
            Console.WriteLine();
            Console.ReadKey();
        }
    }
```

```
   }
```

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

```
Adding some numbers:

Capacity: 8

Count: 7

Content: 45 78 33 56 12 23 9

Content: 9 12 23 33 45 56 78
```

Hashtable Class

```
            Console.WriteLine("The next poppable value in stack: {0}",
            st.Peek());
            Console.WriteLine("Current stack: ");
            foreach (char c in st)
            {
               Console.Write(c + " ");
            }
            Console.WriteLine();

            Console.WriteLine("Removing values ");
            st.Pop();
            st.Pop();
            st.Pop();

            Console.WriteLine("Current stack: ");
            foreach (char c in st)
            {
               Console.Write(c + " ");
            }
        }
    }
}
```


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

```
Current stack:
W G M A
The next poppable value in stack: H
Current stack:
H V W G M A
```

```
}
Console.WriteLine();

//content of ba2
Console.WriteLine("Bit array ba2: 13");
for (int i = 0; i < ba2.Count; i++)
{
    Console.Write("{0, -6} ", ba2[i]);
}
Console.WriteLine();

BitArray ba3 = new BitArray(8);
ba3 = ba1.And(ba2);

//content of ba3
Console.WriteLine("Bit array ba1 after AND operation: 12");
for (int i = 0; i < ba3.Count; i++)
{
    Console.Write("{0, -6} ", ba3[i]);
}
Console.WriteLine();

ba3 = ba1.Or(ba2);
//content of ba3
Console.WriteLine("Bit array ba3 after OR operation: 61");
for (int i = 0; i < ba3.Count; i++)
{
    Console.Write("{0, -6} ", ba3[i]);
```

37. GENERICS

Generics allow you to delay the specification of the data type of programming elements in a class or a method, until it is actually used in the program. In other words, generics allow you to write a class or method that can work with any data type.

You write the specifications for the class or the method, with substitute parameters for data types. When the compiler encounters a constructor for the class or a function call for the method, it generates code to handle the specific data type. A simple example would help understanding the concept:

```
using System;

using System.Collections.Generic;


namespace GenericApplication

{

    public class MyGenericArray<T>

    {

        private T[] array;

        public MyGenericArray(int size)

        {

            array = new T[size + 1];

        }

        public T getItem(int index)

        {

            return array[index];

        }

        public void setItem(int index, T value)

        {

            array[index] = value;

        }
```

```
            Console.WriteLine("Int values after calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values after calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);
            Console.ReadKey();
        }
    }
}
```

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

```
Int values before calling swap:
a = 10, b = 20
Char values before calling swap:
c = I, d = V
Int values after calling swap:
a = 20, b = 10
Char values after calling swap:
c = V, d = I
```


Generic Delegates

You can define a generic delegate with type parameters. For example:

```
delegate T NumberChanger<T>(T n);
```

The following example shows use of this delegate:

```
using System;
using System.Collections.Generic;

delegate T NumberChanger<T>(T n);
namespace GenericDelegateAppl
{
```

```
namespace DelegateAppl

{

    class TestDelegate

    {

        static int num = 10;

        public static void AddNum(int p)

        {

            num += p;

            Console.WriteLine("Named Method: {0}", num);

        }


        public static void MultNum(int q)

        {

            num *= q;

            Console.WriteLine("Named Method: {0}", num);

        }

        public static int getNum()

        {

            return num;

        }


        static void Main(string[] args)

        {

            //create delegate instances using anonymous method

            NumberChanger nc = delegate(int x)

            {

               Console.WriteLine("Anonymous Method: {0}", x);

            };
```

39. UNSAFE CODES

C# allows using pointer variables in a function of code block when it is marked by the **unsafe** modifier. The **unsafe code** or the unmanaged code is a code block that uses a **pointer** variable.

Pointers

A **pointer** is a variable whose value is the address of another variable i.e., the direct address of the memory location. Similar to any variable or constant, you must declare a pointer before you can use it to store any variable address.

The general form of a pointer declaration is:

```
type *var-name;
```

Following are valid pointer declarations:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```


The following example illustrates use of pointers in C#, using the unsafe modifier:

```
using System;

namespace UnsafeCodeApplication
{
    class Program
    {
        static unsafe void Main(string[] args)
        {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} ",  var);
```

2	**public static LocalDataStoreSlot AllocateDataSlot()** Allocates an unnamed data slot on all the threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
3	**public static LocalDataStoreSlot AllocateNamedDataSlot(string name)** Allocates a named data slot on all threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
4	**public static void BeginCriticalRegion()** Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception might jeopardize other tasks in the application domain.
5	**public static void BeginThreadAffinity()** Notifies a host that managed code is about to execute instructions that depend on the identity of the current physical operating system thread.
6	**public static void EndCriticalRegion()** Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception are limited to the current task.
7	**public static void EndThreadAffinity()** Notifies a host that managed code has finished executing instructions that depend on the identity of the current physical operating system thread.
8	**public static void FreeNamedDataSlot(string name)** Eliminates the association between a name and a slot, for all threads in the process. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
9	**public static Object GetData(LocalDataStoreSlot slot)** Retrieves the value from the specified slot on the current thread, within the current thread's current domain. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
10	**public static AppDomain GetDomain()**

```
    class ThreadCreationProgram

    {

        public static void CallToChildThread()

        {

            Console.WriteLine("Child thread starts");

            // the thread is paused for 5000 milliseconds

            int sleepfor = 5000;

            Console.WriteLine("Child Thread Paused for {0} seconds",

                              sleepfor / 1000);

            Thread.Sleep(sleepfor);

            Console.WriteLine("Child thread resumes");

        }


        static void Main(string[] args)

        {

            ThreadStart childref = new ThreadStart(CallToChildThread);

            Console.WriteLine("In Main: Creating the Child thread");

            Thread childThread = new Thread(childref);

            childThread.Start();

            Console.ReadKey();

        }

    }

}
```


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

```
In Main: Creating the Child thread

Child thread starts

Child Thread Paused for 5 seconds

Child thread resumes
```