unit-2 Basics of C#

2.1. Introduction

C# (pronounced "C-sharp") is a modern, object-oriented programming language developed by microsoft as part of the .Net framework. C# is widely used for developing windows applications, web applications, game development, and enterprise solutions. C# is strongly typed, versatile, and supports multiple programming paradigms including procedural, object-oriented, and functional programming.

→ C# programs are runs on the .Net framework.
→ C# is used to develop web apps, desktop apps, mobile apps, games and much more.

Basic structure of C# program

```
// importing necessary namespaces
using System; // provides basic functionalities like input & output.
// Namespace declaration
  namespace myfirstprogram
  {
     class program // class declaration - blueprint for objects.
     {
        // main method - entry point of the program
          static void Main (string[] args)
        {
          Console.WriteLine ("Hello world!");
        }
     }
  }
```

using system;
↳ includes the system namespace that contains classes like console.

namespace myfirstprogram
↳ Defines a namespace to organize code and avoid naming conflicts.
Namespace is a containers that contains classes, interfaces, enums, and other types together to organize code and avoid naming conflicts.

class program
↳ Declares a class named program. Every C# program must have at least one class.

static void main (string[] args)
Main method → entry point starts.
static → belongs to the class and can be run without creating an instance(objects).
void → it does not return a value (absence data type)
string[] args → array that holds command-line arguments.

Console.Writeline ("Helloworld");
↳ prints / display string Helloworld to the console

2.2 Data types, operators, variables

Data types :- In C#, Data types refers to the type of data that a variable can store.

C# has two main types of data types:

1. Value types - stores the actual data
2. Reference types - stores a reference to the memory location of the data.

1. Value types datatypes directly store the value in memory.

	size	example
int	- 4 bytes	10;
float	- 4 bytes	3.14f;
double	- 8 bytes	3.14159;
char	- 2 bytes	'A';
bool	- 1 byte	true;
byte	- 1 byte	255;
short	- 2 bytes	32000;
long	- 8 bytes	1000000000L;

2. Reference types datatypes store references to objects in memory.

	example	
string	string name = "Rajesh";	sequence of characters
object	object obj = 42;	Base type for all types in C#
dynamic	dynamic data = "Hello";	determined at runtime
var	var number = 10;	Compiler determines the type.

Example of Different data types

```
using system;
namespace Datatypesexample
{
   class program
   {
      static void main()
      {
         //value types
         int age = 25;
         float pi = 3.14f;
         double d = 2.71828;
         char c = 'R';
         bool is_csharpfun = true;
         decimal salary = 25000.75m;

         // Reference types
         string name = "Rajesh";
         object obj = 100; // can hold any type

         console.writeline ("integer: "+age);
         Console.writeLine ("float:" +pi);
         Console. writeLine ("double:"+d);
         console. writeLine ("char:"+c);
         Console. writeline("bool:"+is_csharpfun);
         Console. writeline ("Decimal:"+salary);
         Console. writeLine ("string:"+name);
         Console. writeLine ("object:"+obj);

         // wait for user input before closing
           Console. Readline ();
      }
   }
}
```

Variables in C#

variables are containers for storing data values.

To declare a variable in C#, you need to specify the type of the variable followed by its name.

Examples are:

```
int age = 25; // Integer variable
double Salary = 50000.50// Double variable
char L = 'R'; // Character variable
String name = "Rajesh"; // String variable
bool isTeacher = true; // Boolean variable
```

Syntax:

```
type variable_Name = value;
```

where type is a C# type such as int or float, and variable_Name is the name of the variable such as x or name. The equal (=) sign is used to assign values to the variable.

#Example program in C# to show variable.

```
using System;
class program
{
    static void main()
    {
        int age=28;
        String name = "Rajesh";
        Console.writeline("Name:"+name);
        Console.writeline("Age:"+age);
    }
}
```

output: Name: Rajesh
Age: 28

Rules to declare variable Name

1. Variable names must start with a letter (A-Z or a-z) or an underscore (_).
2. Variable names cannot start with a digit (0-9).
3. Variable names cannot contain spaces or special character (@, #, $, etc), except for _.
4. c# is case-sensitive (myvar and myVar are different).
5. Reserved keywords (eg int, class, new) cannot be used as variable names.
6. Multiple variables of the same type can be declared in a single line eg

   ```
   int x, y, z;
   ```

7. Variables can be initialized at the time of declaration eg int num = 10;

Operators in C#

Operators are symbols that specific operations on operands. operators are used to perform operations on variables and values.

There are various types of operators in C# that are :-

1. Arithmetic operators :- +, −, *, /, %.
2. Relational (comparison operators :- ==, !=, >, <, >=, <=
3. Logical operators :- &&, ||, !
4. Bitwise operators :- &, |, ^, >>, <<, ~
5. Assignment operators :- =, +=, -=, *=, /=, %=
6. Increment and decrement operators :- ++, --
7. conditional (Ternary) operator :- ?:

Example program :

```
using system;
namespace operators
{
  class program
  {
    static void main()
    {
      int a=10, b=5;
      Console.writeLine("Arithmetic(+):" +(a+b));
      Console.writeLine("Relational(>):"+(a>b));
       bool x= true, y=false;
       Console.writeLine("Logical(&&):" + (x&&y));
       int bitwise1 =5, bitwise2=3;
       Console.writeLine("Bitewise(&):"+(bitwise1 & bitwise2));
        int assign =5;
        int assign +=3;
        Console.writeLine("Assignment(+=):" +assign);
        int num =7;
        num++;
```

```
        Console.WriteLine("Increment (++) : " + num);
    }
}
}
```

output:

```
Arithmetic (+) : 15
Relational (>) : True
Logical (&&) : false
Bitwise (&) : 1
Assignment (+=) : 8
Increment (++) : 8
```

8. Null-Coalescing operators(?? and ??=)
   ↳ used to handle null values

operator	Description	Example	Result
??	Returns the first non-null value	name ?? "unknown"	"unknown"
??=	Assigns value if the variable is null	name ??= "Guest"	"Guest"

9. Type casting & Type checking operators
   ↳ used to check or convert types.

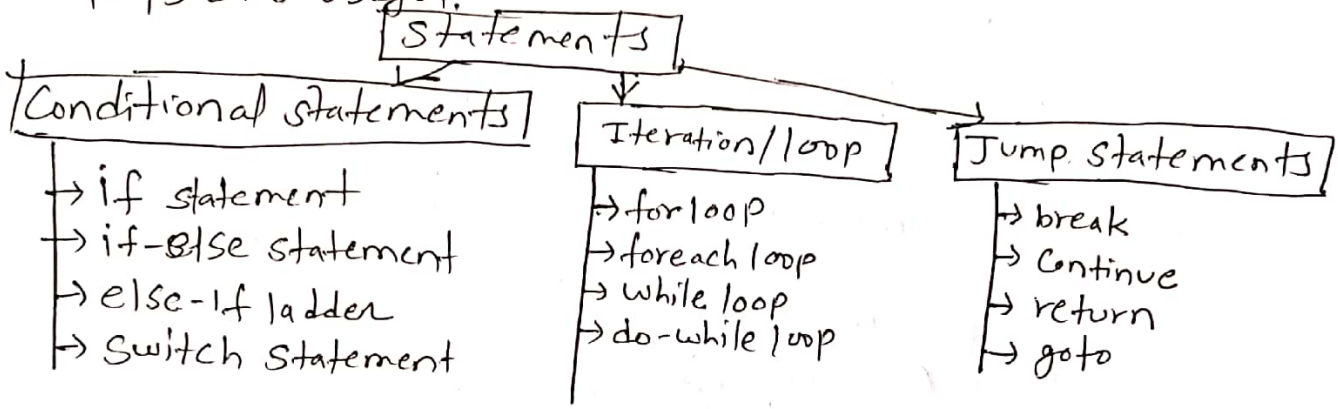
operator	Description	Example	Result
is	checks if an object is of a certain type	obj is int	true/false
as	Converts an object to a type if possible	obj as string	"Hello" or null
(type)	Explicit type casting	(int) 5.6	5

10. typeof → Gets the type of a data type — typeof(int) — System.Int32
11. sizeof → Gets the size of datatype — sizeof(int) — 4
12. nameof → Gets the name of variable — nameof(myvar) — "myvar"

2.3 Control statements in C#

The control statements are used to control the flow of execution of the program. If you want to execute a specific block of instructions only when a certain condition is true, then control statements are useful.

If you want to execute a block repeatedly, then loops are useful.



Conditional statements are also known as Decision making or selective statements.

a. if statement :- Executes a block of code if a specified condition is true.

Syntax:

```
if (condition)
{
  // code to execute if condition is true
}
```

Program Example:

```
using System;
Class program
{
  static void Main ( )
  {
    int num;
    num = Convert.ToInt32(Console.ReadLine( ));
    if (num > 5)
    { console.writeLine (" Number is greater than 5");
```

} } }

b. If-else statement

Executes one block if the condition is true, otherwise executes else block.

Syntax:

```
if (condition)
{
  // code to execute if condition is true
}
else
{
  // code to execute if condition is false
}
```

program example :-

```
using system;
namespace evenodd
{
  class program
  {
    static void Main (string args[])
    {
      Console.writeline ("Enter number:");
       int n = Convert.ToInT32 (console.Readline());
      if (n%.2 == 0)
      {
        Console.writeLine ("even number:"+n);
      }
      else
        console.writeline ("odd number:"+n);
  } } }
```

output: Enter number : 5
Odd number : 5

Example2: write a C# program to find Largest number among three numbers.

```
using System;
namespace largest
{
  class program
  {
     int num1, num2, num3;
     Console.WriteLine("Enter the first number:");
     num1 = convert.ToInt32(Console.ReadLine());
     Console.WriteLine("Enter the second number:");
     num2 = convert.ToInt32(Console.ReadLine());
     Console.WriteLine("Enter the third number:");
     num3 = Convert.ToInt32(Console.ReadLine());
     if (num1>=num2 && num1>=num3)
     {
       Console.WriteLine("Largest number is:"+num1);
     }
     else if(num2>=num1 && num2>=num3)
     {
       Console.WriteLine("Largest number is:"+num2);
     }
     else
     {
       Console.WriteLine("The Largest number is:"
         +num3);
     }
  }
}
```

output: Enter the first number: 10
Enter the second number: 15
Enter the third number: 20
The largest number is 20

c. else-if ladder statement: Checks multiple conditions sequentially.

Syntax:

```
if (condition1)
{
  //Code to execute if condition 1 is true
}
else if (condition2)
{
  //code to execute if condition2 is true
}
else
{
  //code to execute if none of the above conditions are true
}
```

Example 1 program

```
using System;
namespace elseifladder
{
  class program
  {
    static void main (string[] args)
    {
      int i=0;
      if(i>0)
      {
        console.writeLine("positive number");
      }
      else if(i<0)
      {
        console.writeLine("negative number");
      }
      else
      {
        console.writeLine("neither positive nor negative");
      }
    }
```

d. switch case statement :- selects one of many code blocks to execute.

Syntax:

```
switch (variable)
{
  case value1 :
  // code to execute if variable equals value1
  break;
  case value2 :
  // code to execute if variable equals value2
   break;
   :
   :
  case value n;
  // code to execute if variable equals value n
  default :
  // code to execute if variable does not match
    any case
    break;
}
```

Q. write a C# program to perform arithmetic operations on two numbers from the users using switch case statement.

```
using System;
namespace calc
{
  class program
  {
    static void Main()
    {
      double num1, num2, result;
      string choose_operator;
      Console.Writeline ("Enter the first number:");
      num1 = Convert.ToDouble (Console.Readline());
```

```
Console.WriteLine("Enter second number:");
num2 = Convert.ToDouble(Console.ReadLine());

Console.WriteLine("enter the operator (+, -, *, /): ");
Choose_operator = Console.ReadLine();
switch (choose_operator)
{
    case "+":
        result = num1 + num2;
        Console.WriteLine("Result: "+result);
        break;
    Case "-":
        result = num1 - num2;
        Console.writeLine("Result: "+result);
        break;
    Case "*":
        result = num1 * num2;
        Console.writeLine("Result: "+result);
        break;
    case "/":
        if(num!=0)
        {   result = num1/num2;
            Console.writeLine("Result: "+result);
        }
        else
        {Console.WriteLine("Error: Division by zero is
                            not allowed.");
        }
        break;
    default:
        Console.WriteLine("Invalid operator entered");
        break;
}
}
}
```

output:

```
Enter the first number:
12
Enter the second number:
8
Enter the operator (+, -, *, /):
+
Result: 20
```

2. Looping statements: Loop may be defined as block of statements that are repeatedly executed for a certain number of times or until a particular condition is satisfied.

Loop statements allow you to execute a block of code multiple times.

a. for Loop: Executes a block of code for a specified number of times.

Syntax:

```
for( initialize; condition; update)
{
    // codes to execute
}
```

Example: program to generate fibonacci series

```
using System
class program
{
    static void Main( )
    {
        int n, t1=0, t2=1, nextTerm = t1+t2;
        // Get the number of terms from the user
        Console.Write("Enter the number of terms:");
        n = Convert.ToInt32(Console.ReadLine());
```

```
// print the first two terms
 Console.Write("fibonacci series: {0}, {1},", t1, t2);

//print the remaining terms
  for(int i=3; i<=n; ++i)
  {
    Console.write(nextTerm + ", ");
      t1 = t2;
      t2 = nextTerm;
      nextTerm = t1+t2;
  }
} }
```

output:

enter the number of terms: 10
fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,

c# program to print Numbers from 1 to 10

```
using System;
class program
{
  static void Main()
  {
    // loop to print numbers from 1 to 10
    for(int i=1; i<=10; i++)
    {
      Console.WriteLine("Number: "+i);
    }
  } }
```

output: Number: 1
Number: 2
⋮
Number: 10

b. **foreach loop** :- In C#, foreach loop is used to iterate over elements in an array or collection without needing an index.

Syntax

```
foreach (datatype variable in collection)
{
    // Code to execute for each element
}
```

Example1

```
using System;
class program
{
    static void Main()
    {
        string[] fruits = { "Apple", "Banana", "Grapes" };
        foreach (string fruits in fruits)
        {
            Console.WriteLine("fruit: " + fruits);
        }
    }
}
```

output:

```
fruit: Apple
fruit : Banana
fruit: Grapes
```

Example 2

```
using System;
class program
{
    static void main()
    {
        string name = "Rajesh";
        foreach (char letter in name)
        {
            Console.WriteLine(letter);
        }
    }
}
```

output:

```
R
a
j
e
s
h
```

c. while loop in C# :-

The while loop executes a block of code repeatedly as long as the condition is true.

Syntax :-

```
while (condition)
{
    // code to execute
}
```

Example 1

C# Program to print Numbers from 1 to 3

```
using System;
class program
{
    static void Main()
    {
        int i = 1;
        while (i <= 3)
        {
            Console.WriteLine(i);
            i++;
        }
    }
}
```

output:

```
1
2
3
```

Example 2

c# program to countdown from 5 to 1

```
using System;
class program
{
    static void Main()
    {
        int i = 5;
        while (i >= 1)
        {
            Console.WriteLine(i);
            i--;
        }
    }
}
```

output:

```
5
4
3
2
1
```

d. do-while loop :- Executes a block of code at leat once, and then repeats as long as a condition is true. It is also called Exit Controlled and post test loop.

Syntax:

```
do
{
  // code to execute
} while (condition);
```

→ do-while loop is similar to the while loop, but it executes at least once before checking the condition.

C# program to ask for password until it's correct

```
using system;
namespace dowhileloop
{
  class program
  {
    static void Main()
    {
      string password;
      do
      {
        Console.write("Enter password:");
        password = Console.ReadLine();
      } while (password != "1234");
      Console.writeLine("Access granted!");
  } } }
```

Enter password: hello
Enter password: pass
Enter password: 1234
Access granted!

Example2: print Numbers from 1 to 5 using do-while loop

```
using system;
class program
{
   static void main()
   {
      int i=1;
      do
      {
         Console.WriteLine(i);
         i++;
      }while(i<=5);
   }
}
```

output:

```
1
2
3
4
5
```

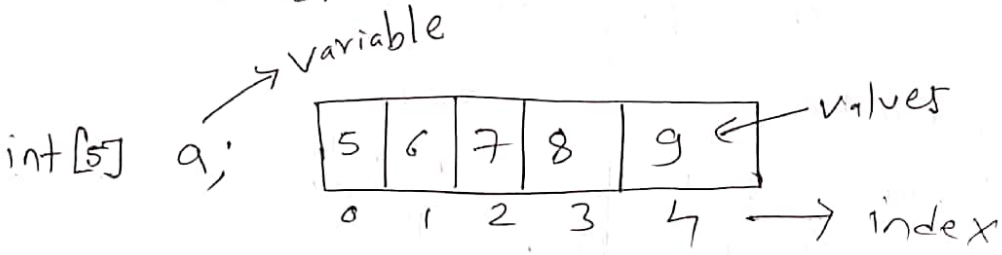
2.4 Arrays, Classes, Structures, Enumerations

Arrays in C#

→ An array is a collection of elements of the same type that are stored in a contiguous memory location.

→ Arrays allows us to store multiple values in a single variable.

→ Arrays have a fixed size that is determined when it is initialized.



properties of Array

(i) Arrays are indexed starting from 0.

(ii) Arrays can store data of any type (values or reference).

(iii) The size of an array is fixed after it is declared, meaning it cannot be resized.

(iv) Arrays can be single-dimensional or multi-dimensional

(1) Single Dimensional Arrays

Syntax

```
type[] arrayName;
type[] arrayName = new type[size];
```

eg `int[] numbers = new int[5];`

(2) Multi-Dimensional Arrays:

Syntax:

```
type[,] arrayName = new type[rows, columns];
```

eg `int[,] matrix = new int[3,3];`

(3) Jagged Arrays :- Array of arrays

Syntax

```
type[][] arrayName = new type[rows][];
```

eg `int[][] jaggedArray = new int[3][];`

Single dimensional Array :- Stores Elements in a single Line.

Program Example

```
using System;
namespace ArrayExample
{
  class program
  {
    static void Main (string[] args)
    {
      int[] numbers = { 10, 20, 30, 40, 50};
      Console.WriteLine("Array Elements:");
       for( i=0; i<numbers.Length; i++)
        {
          Console.WriteLine (numbers[i]);
        }
    }
  }
}
```

output: Array Elements:
10
20
30
40
50

Another way

```
using System;
 class program
  { static void Main( )
    {
      int[] numbers = new int[5];
      numbers[0] = 10;
      numbers[1] = 20;
      numbers[2] = 30;
      numbers[3] = 40;
      numbers[4] = 50;
      Console.WriteLine ("Array elements:");
      for (int i=0; i<numbers.
          Length; i++){
      Console.WriteLine(numbers[
                  i]);
      }
    }
}
```

output:
Array Elements:
10
20
30
40
50

Classes in C# :- A class is a blueprint for creating an objects. It consists of variables (properties) and methods (functions).

Syntax

```
class className
{
    // fields (variables)
    // Methods (functions)
}
```

Example program of class in C#

```
using System;
class Car
{
    public string brand = "Toyota";
    public void Horn()
    {
        Console.WriteLine("Beep Beep!");
    }
}
class Program
{
    static void Main()
    {
        Car myCar = new Car(); // Creating an obj
        Console.WriteLine(myCar.brand);
        myCar.Horn();
    }
}
```

output: Toyota
Beep Beep!

Class Example-2

```
using System;
namespace ClassExample
{
  class person
  {
    public string name;
    public int age;
    public void Display_Details()
    {
      Console.WriteLine("name: " + name);
      Console.WriteLine("Age: " + age);
    }
  }
  class program
  {
    static void Main(string[] args)
    {
      person p = new person();
      p.name = "Rajesh";
      p.age = 28;
      p.Display_Details();
    }
  }
}
```

output: Name: Rajesh
Age: 28

C# program to calculate simple interest using class

```
using System;
class CalculatorSI
{
  public double SI(double p, double t, double r)
   return (p*t*r)/100;
}
class program
{static void main()
 {CalculatorSI obj = new CalculatorSI();
  Console.Write("Enter principal Amount: ");
  double p = Convert.ToDouble(Console.ReadLine());
  Console.Write("Enter Annual Interest Rate : ");
  double r = Convert.ToDouble(Console.ReadLine());
  Console.Write("Enter Time period : ");
  double t = Convert.ToDouble(Console.ReadLine());
  double interest = obj.SI(p,t,r);
  Console.WriteLine("Simple Interest : "+ interest);
 }
}
```

output:

```
Enter principal Amount: 1000
Enter Annual Interest Rate: 5
Enter Time period : 2
Simple Interest: 100.0
```

Structure in C#

A Structure (struct) in C# is a value type that is used to encapsulate related variables of different data types. It is similar to a class but is stored on the stack instead of the heap, making it more efficient for small data types.

→ A structure is like a class but simpler.
→ It stores data directly in memory (stack).
→ No inheritance but can have methods.
→ It can be created instale without new keyword.

Syntax of Structure

```
struct structureName
{
    public datatype variable1;
    public datatype variable2;
    public void Display()
    {
        Console.WriteLine("variable1 : " + variable1);
        Console.WriteLine("variable2 : " + variable2);
    }
}
```

Example program to demonstrate Structure in C#

```
using System;
struct person
{
    public string name;
    public int age;
    public void Display()
    {
        Console.WriteLine("Name:" + name);
        Console.WriteLine("Age:" + age);
    }
}
class program
{
    static void Main()
    {
        person p1;
        p1.name = "Rajesh";
        p1.age = 28;
        p1.Display();
    }
}
```

output:
Name: Rajesh
Age: 28

Enumeration (enum) in c#

Enumeration is a value data type in c#. Enum allows you to define a set of named constant values. like colors, planets, Months of the year, Days of the week.

Syntax:

```
enum EnumName
{
  Value1,
  Value2,
  Value3
}
```

```
enum days
{
  sunday,
  monday,
  :
  saturday
}
```

By defaut, the first value starts from 0 and increases by 1 for each item. constants values means unchange able / read-only variables).

Example program of enum : Days of the week

```
using system;
enum Days
{ sunday = 0,
  monday = 1,
  Tuesday = 2,
  wednesday = 3,
  Thursday = 4,
  friday = 5,
  saturday = 6.
}
class program
{ static void main ( ){
  Days today = Days.monday;
  Console.writeLine ("Today is: "+ today);
  Console.writeline("Numeric value : "+ (int) today);
} }
```

output: Today is: Monday
Numeric value: 1

Consider an Example where we can use an enum to represent the status of a task (e.g. Not started, In progress, and Completed).

⇒
```
using System;
enum TaskStatus { Notstarted = 1, Inprogress = 2,
                  Completed = 3};

class program
{ static void Main()
  {
    TaskStatus ts = Taskstatus.Inprogress;
    Console.WriteLine("Task status:" + ts);
    Console.WriteLine("Numeic value:" + (int)ts);
} }
```

output: Task status: Inprogress
Numeic value: 2

Example 3
```
using System;
enum Level
{
  low,
  Medium
  High
}
// You can alters enum items with the dot.
class program { static void Main() {
  Level myVar = Level.Medium;
  Console.WriteLine(myVar);
} }
```

2.5 partial classes, static classes, sealed classes.

1. partial classes

A partial class allows the definition of a class to be split into multiple files. This is useful when working with large classes or when multiple developers are working on different parts of the same class.

Syntax:

```
//file1
public partial class myclass
{ public void method1()
  { // code statements
  } }
//file2
public partial class myclass
{ public void method2(){
    // code statements
  } }
```

Points to know

- → Allows a class to be split across multiple files.
- → Each file must have the partial keyword
- → useful for organizing large classes and when working in teams.

Example program of partial classes

```
using System;
public partial class Employee
{ public string name;
}
public partial class employee
{ public void employeeNameDisplay()
  { Console.WriteLine(Name);
  } }
class program
{ static void Main()
{
 Employee emp = new
                employee()
 emp.Name = "Rajesh";
emp.EmployeeNameDisplay
    ();
} }
```

2. Static classes

A static class is a class that cannot be instantia-ted (no objects can be created). It can only contain static members (methods, fields, properties).

Syntax:

```
public static class myclass
{
  public static void mymethod()
  {
  ---
  }
}
```

Example program of static class in C#

```
using System;
Public static class mathoperations
{ public static int Add(int a, int b)
  {
    return a+b;
  }}
class program
{
  static void Main()
  {
    int result = Mathoperations.Add(5,10);
    Console.writeLine("Addition is:" + result);
  }}
```

output: Addition is: 15

- → Cannot be instantiated (no new keyword).
- → All members must be static.
- → useful for methods that don't require object instances.

3. Sealed classes

A sealed class is a class that cannot be inherited. It is used when you want to prevent a class from being used as a base class.

Syntax:

```
public sealed class myclass
{
    public void mymethod()
    {
    }
}
```

Example of sealed class in c#

```
using system;
public sealed class Car
{
    public void Drive()
    {
        Console.writeline("Driving the car");
    }
}
Class program
{
    static void Main()
    {
        Car car = new Car();
        car.Drive();
    }
}
```

output: Driving the car

Points to Know

- Sealed classes cannot be inherited
- Used to prevent further subclassing when inheritance is not necessary
- Can still be used to create objects.

2.6 Constructors and Destructors

A constructor in C# is a special method used to initialize objects. It has the same name as the class and does not have a return type (not even void).

A constructor is a special method in C# that is automatically called when an object of a class is created. It is used to initialize objects.

Points to know

- → A constructor has the same name as the class.
- → It does not have a return type (not even void).
- → It is called automatically when an object is created.
- → It is mainly used to initialize variables or set up objects.

1. Default Constructor

A constructor with no parameters.
If no constructor is defined, C# provides a default constructor automatically.

Example1

```
using System;
class Car
{
    public Car()
    {
        Console.WriteLine("Constructor called!");
    }
    static void Main()
    {
        Car obj = new Car();
    }
}
```

output: Constructor called!

→ Constructor initializes numeric fields to 0 and string fields to null print empty in Console.WriteLine().

Example2

```
using System;
class Car{
    string brand;
    int year;
    public void Display()
    {
        Console.WriteLine("Car Brand: "+brand+"Year: "
        +year);
    }
    static void Main(){
        Car obj = new Car();
        obj.Display();
    }}
```

output:
Car Brand: Year: 0

2. Parameterized Constructor

A constructor that accepts parameters to initialize object properties.

```
using System;
Class Car {
        string brand;
        public Car(string b)
        {
          brand = b;
          Console.WriteLine("car brand: "+brand);
        }
        Static Void Main()
        {
          Car obj = new Car("BMW");
        }
}
```

output: car brand: BMW

Examp 2:

```
using System;
Class Student
{
  String name;
  int age;
  public Student(string studentName, int studentAge)
  {
    name = studentName;
    age = studentAge;
  }
  public void Display()
  {
    Console.WriteLine("student Name: "+name);
    Console.WriteLine($"student Age: {Age}");
  }
  static void Main()
  {
```

```
Student S1 = new Student("Rajesh", 28);
 S1.display();
}}
```

output: Student Name : Rajesh
Student Age : 20

Destructor in C# :- A destructor is used to clean up or free resources when an object is destroyed.

```
using System;
Class myclass
{  public myclass() //constructor
   {  Console.writeline("constructor called");
   }
~myclass()
  {  Console.writeLine("Destructor called!");
  }
}
Class program
{  static void Main(string args[])
   {
     myclass obj = new myclass();
   }
}
```

output: Constructor called!
Descructor called

⇒ When the object is no longer in use and the garbage collector runs, the destructor is called. Destructor is depends on the garbage collection process.

Inheritance and polymorphism

Inheritance :- A mechanism where child class inherits properties and methods from another class (parent class).

Syntax :

```
Class A
{
--
}
Class B : A
{
--
}
```

Types of Inheritance
- Single level inheritance
- multi level inheritance
-

Polymorphism :- It allows one method to do different things based on the object it is acting upon.

Runtime polymorphism
- Virtual functions
- Method overriding

Compile Time polymorphism
- method overloading
- operator overloading

Example program of inheritance & polymorphism in one program

method overriden program

```
using system;
class Animal
{ public virtual void Speak()
  {
    Console.WriteLine("Animal speaks");
} }

Class Dog : Animal
{
  public override void Speak()
    { Console.WriteLine("Dog barks");
  } }

Class program
  {
    Static Void Main(String[] args)
    {
      Animal animal = new Animal();
      animal.Speak();
      Dog dog = new Dog();
      dog.Speak();
    }
}
```

output:

Animal speaks

Dog barks

2.8. Concept and implementation of interfaces

in C#, an interface is a contract that defines a set of methods, properties, events that a class or struct must implement. It does not contain any implementation details - only declarations.

Syntax:

```
public interface Interfaceeg
{
    void method();
}
```

- → No implementation: only method signatures, no method bodies.
- → A class can implement multiple interfaces.
- → Interfaces allow different classes to be treated as same type.

A class or struct can implement multiple interfaces.

Example:

```
Using System;
public interface Animal
{
  void makesound();
}
public class Dog : Animal
{
  public void Makesound()
  {
    console.WriteLine("woof woof!");
  }
}
class program
{
  static void Main()
  {
    Animal Doganimals = new Dog(); //interface reference
    Doganimals.makesound(); //call the method in the dog cl
  }
}
```

output:
woof woof!

Virtual methods in C# :- A virtual method in C# is a method in the base class that can be overridden in a derived class. It allows runtime polymorphism (dynamic method dispath) / Late binding.

Syntax:

```
class Baseclass {
    public virtual void show()
    { Console.writeLine("Base class method");
    }}
class Derivedclass: Baseclass {
    public override void show()
    {
    Console.writeLine("Derived Class method");
    }}
```

Points To know

- → virtual keyword in the base class.
- → override keyword used in the derived class to override the virtual method
- → The base class reference can call the overriden method.

Method overriding

The process of redefining a virtual method in a child class.

Example of virtual methods

```
using system;
Class Base
{
  public virtual void displaymessage()
    { Console.WriteLine("I am Base class method");
    }
  }
  class child : Base
  {
    public override void displaymessage()
    { Console.writeLine("I am child class method");
  }}
  Class Main
  {
    static void Main()
    {
      Base obj = new child(); // overridden method (Late Binding)
    }}
```

output: I am child class method

Abstract classes and methods in C#

An abstract class is a class that cannot be instantiated and meant to be inherited by other classes. It can have abstract methods (without a body) and regular methods (with implementation).

An abstract method is a method without implementation in the abstract class. The derived class must override and implement it.

Syntax of Abstract class and method

```
abstract class Animal
{ public abstract void makesound();
  public void Eat() // Regular method
  {
    Console.WriteLine("this animal eatsfood");
  }
}
```

Example of Abstract class and Abstract Method in C#

```
using System;
abstract class Animal
{ public abstract void makesound();
  public void Eat() // Non abstract method means regular method
  { Console.WriteLine("This animal eats food");
  }
}
class Dog : Animal
{ public override void makesound()
  { Console.WriteLine("Dog barks");
  }
}
```

```
class main
{
    static void main()
    {
        Animal myDog = new Dog();
        myDog.MakeSound();
        myDog.Eat();
    }
}
```

Dog barks
This animal eats food.

Key points to know abstract classes

- abstract keyword used for abstract classes and methods.
- → cannot create an object of an abstract class new animal() X
- → must override abstract methods in derived classes using override.
- → Abstract classes can have both abstract and non abstract methods.

Exception Handling in C#

Exception handling in C# is used to manage runtime errors, so the program doesn't crash unexpectedly. C# provides a powerful way to handle exceptions using the try, catch, finally and throw keywords.

Exception Handling keywords:

1. try :- Defines a block of code that may throw an exception.
2. catch :- Catches and handles the exception thrown by the try block.
3. finally :- Executes code After try and catch, regardless of whether an exception occurred or not.

4. throw :- used to explicitly throw an exception.

Syntax of exception Handling

```
try
{
  // code that might be throw an exception
}
catch (exceptiontype ex)
{
  // code to handle the exception
}
finally
{
  // code that will always run
}
```

program :- Divide by zero exception Handling

```
using System
class program
{
  static void main()
  {
    try
    {
      Console.WriteLine("Enter numerator:");
      double numerator = double.Parse(Console.ReadLine());
      Console.WriteLine("Enter denominator:");
      double denominator = double.Parse(console.Readline
                                        ());
      double result = numerator/denominator;
       Console.WriteLine("Result:"+ result);
    }
```

```
catch (DivideByZeroException)
{
    Console.WriteLine("Error: Cannot divide by zero");
}
catch (FormatException)
{
    Console.WriteLine("Error: Invalid input.");
}
}
}
```

output:

```
Enter numerator:
10
Enter denominator:
0
Error: Cannot divide by zero.
```

Note:- FormatException handles invalid (non-numeric) input.

Delegates: A delegate is a pointer to a method. That means, a delegate holds the address of a method which can be called using that delegate.

A delegate is also called type safe pointer.
A delegate is a way to call a method indirectly using a reference.

Syntax:

```
delegate returntype DelegateName(parameters);
```

Steps to use a Delegate

step1: Declare a Delegate that matches the method signature you want to point to.

```
delegate void mydelegate(string name);
```

step2: Create a method that matches the Delegate's signature

```
static void ~~Rajesh~~Name(string name)
{
    Console.WriteLine("Hello "+name);
}
```

step3: Instantiate the Delegate

```
mydelegate md = new mydelegate(~~Rajesh~~name);
```

step4: Call the method using delegate

```
md("Rajesh");
```

```
using System;
class program{
delegate void mydelegate(string name);
 static void ~~md~~Name(string name)
  {Console.WriteLine("Hello "+name);
 }
```

```
static void main()
{
mydelegate md =
 new mydelegate
      (Name);
md("Rajesh");
}
```

Example

Delegate for Different Arithmetic operation

```
using System;
class program
{
  delegate int mathoperation(int a,
                             int b);
   static int Add (int a, int b)
    {return a+b;
    }
   static int mul(int a,int b)
    {return a*b;
    }
     static void main( )
    {
      mathoperation mo = new (Add);
       mo (5,3);
       mathoperation mo1 = new(mul);
        mo1(5,3);
        Console.writeLine("Add:"+mo(5,3));
        Console.writeLine("multiply:"+mo1(5,3));
    }
}
```

Output: Add: 8
multiply: 15

Key points about Delegate

- → A delegate stores a reference to a method.
- → It ensures type safety (the method signature must match).
- → used in event handling, callbacks, & functional programming.

Q. Delegate program example with no parameter

```
using system;
class program
{
  delegate void delegate_Name ( ); //declare delegate
  static void Main ( )
  {
    delegate_Name d = new delegate_name (rajesh);
  } d( ); // invoke the delegate
  static void rajesh ( )
  { Console.WriteLine ("Delegate first program");
  }
}
```

output:

Delegate first program.

Q.2 Delegate program to Add two numbers

```
using system;
class program
{  delegate int AddNumbers (int x, int y);
   static void Main ( )
   {
    AddNumbers a = new AddNumbers (sum );
    Console.WriteLine ("Enter first Number: ");
    int num1 = Convert.ToInt32 (Console.ReadLine( ));
    Console.WriteLine ("Enter second Number: ");
    int num2 = Convert.ToInt32 (Console.ReadLine( ));
    int result = a[redacted] (num1, num2);
```

```
Console.WriteLine("The sum is : "+result);
}
static int sum(int x, int y) // int a, int b
{
  return x+y;          // return a+b;
}
}
```

Note

method that matches the delegate signature but the parameter names can be changed with the same type.

output : Enter first Number :
5
Enter second Number :
7
The sum is 12.