C# Short Notes

Datatypes

Value type: Value type variables can be assigned a value directly **Eg**:int char float

Reference Type: The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables

Pointer Type: The pointer in C# language is a variable, it is also known as locator or indicator that points to an address of a value. **Eg**. int *a;

Type Conversion

Implicit type conversion: these conversions are performed by C# in a type-safe manner Eg: smaller to larger integral types

Explicit type conversion: these conversions are done explicitly by users using the pre-defined functions.

Eg: double d=545.45;

    int i

//Explicit type

i=(int)d;

Variables

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

Syntax: <data_type> <variable_list>; **Eg:** int i,j,k;

Variable Initialization Eg: int d = 3, f = 5; byte z = 22;

1. **Lvalue : :** An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
2. **Rvalue:** An expression that is an rvalue may appear on the right- but not left-hand side of an assignment

Encapsulation

Encapsulation is defined 'as the process of enclosing one or more items within a physical or logical package'.

Access Specifier

- Public (Allows member Func and Variables to be accessed outside the class)
- Private (Only functions of the same class can access its private members)
- Protected (Allows child class to access Base class varibles and func)
- Internal (Same as public)
- Protected Internal (Same as Protected)

Methods

A method is a group of statements that together perform a task.

Declare: <Access Specifier> <Return Type> <Method Name>(Parameter List)

Calling: variable=<obj.methodname>(parameter);

Recursive Method : A method can call itself. This is known as recursion

Parameter

- Values (When a method is called, a new storage location is created for each value parameter)
- Reference (A reference parameter is a **reference to a memory location** of a variable)
- Output (Return More than 1 values)

Nullables

Provides a special data types, the nullable types, to which you can assign normal range of values as well as null values.

Syntax < data_type> ? <variable_name> = null;

Arrays

An array stores a fixed-size sequential collection of elements of the same type.

Syntax: Datatype[] arrayName; **Eg:** int[] balance = new int[5] {1,2,3,4,5};

Multi-dimensional arrays

Syntax: string [,] names; **Eg:** int [,] a = int [3,4] = {{0, 1, 2, 3} ,{4, 5, 6, 7} ,{8, 9, 10, 11}}

String

String is an object of **System.String** class that represent sequence of characters.

Operations: concatenation, comparision, getting substring

Structure

A structure is a value type data type. It helps you to make a single variable hold related data of various data types. The **struct** keyword is used for creating a structure.

Syntax: struct <name>

Features

- Structures can have defined constructors, but not destructors.
- Unlike classes, structures cannot inherit other structures or classes.
- Structures cannot be used as a base for other structures or classes.
- A structure can implement one or more interfaces.

Enums

An Enumeration is a set of named integer constants. An enumerated type is declared using the enum keyword. C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.

Syntax: enum <enum_name>{

enumeration list

};

Eg: enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

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 will have exact same name as the class and it does not have any return type.
Eg: public line() //**Default Constructors ,** public line(double len) //**Parametrised Constructors**

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 will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters.

Eg: public Line()// **constructor** ~Line() **//destructor**

Static Members

When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the static member.

Eg: public static int num;

<u>OOPS Concept</u>

- Class
- Object
- Inheritance
- Abstraction
- Encapsulation
- Polymorphism

<u>Inheritance</u>

Its is consist of **class and Objects**. Used for **Code reusability** and **Relationship with another class**. Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.

Symbol: ":"
Types

- **Single**-one to one class
- **Multiple**-two to one class
- **Multilevel**-generation type class
- **Hierarchical**-one to two class
- **Hybrid**-combination of multiple and Hierarchical

<u>Polymorphism</u>
Ability to do many forms with Inheritance
Types
There are two types of polymorphism in C#: **static or compile time polymorphism and runtime polymorphism**. Compile time polymorphism is achieved by method overloading and operator overloading in C#. Runtime polymorphism in achieved by method overriding which is also known as dynamic binding or late binding.
Method Overloading: If we create two or more members having same name but different in number or type of parameter**(Same Function and Different Parameter)**

Eg:
```
public static int add(int a,int b){
    return a + b;
  }
  public static int add(int a, int b, int c)  {
    return a + b + c;
  }
public class TestMemberOverloading
{
  public static void Main() {
    Console.WriteLine(Cal.add(12, 23));
    Console.WriteLine(Cal.add(12, 23, 25));
 }
```

Output:
```
35
60
```

Method Overriding:

If derived class defines same method as defined in its base class. **Same Function and Parameter.** To perform method overriding in C#, you need to use **virtual** keyword with base class method and **override** keyword with derived class method.

Eg:
```
public class Animal{
   public virtual void eat(){
      Console.WriteLine("Eating...");
   }
}
public class Dog: Animal
{
   public override void eat()
   {
      Console.WriteLine("Eating bread...");
   }
}
public class TestOverriding
{
   public static void Main()
   {
      Dog d = new Dog();
      d.eat();
   }
```

OUTPUT:
```
Eating bread…
```

<u>**Abstraction**</u>

Abstraction in C# is the process to hide the internal details and showing functionality only.

To perform method overriding in C#, you need to use **abstract** keyword with base class method and **override** keyword with derived class method.

Uses

- Ability to hide irrelevant Content.
- Only Declarition no definition.
- Need Inheritance

Keyword: abstract

<u>**Encapsulation**</u> **(Need Inheritance)**

Encapsulation is the concept of wrapping data into a **single unit**. It collects data members and member functions into a single unit called class.Ability to hide irrelevant Content. Aquired by **Private access.**

Keyword: Final (Variable,function,class cannot be changed)

<u>**Interface**</u>

Interface is similar to a class. It is like abstract class because all the methods which are declared inside the interface are abstract methods. It cannot have method body and cannot be instantiated.It is used to achieve multiple inheritance. **Keyword: Interface**

Generics

Generics allow you to write a class or method that can work with any data type.

Uses
It helps you to maximize code reuse, type safety, and performance.
You can create your own generic interfaces, classes, methods, events and delegates
Symbol: <generic name >
Inizitate
Classname<**datatype**> obj   = **new** classname<datatype> ("Contents");

Delegates

A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.Mainly used to change value in the memory.
Eg: public delegate int MyDelegate (string s); or Number nc1 = new Number (AddNum);

File IO

File is a collection of data stored in a disk with a specific name and a directory path. When a fileis opened for reading or writing, it becomes a stream.The stream is basically the sequence of bytes passing through the communication path. There are two main streams: the **input stream and the output stream**. The input stream is used for reading data from file (read operation) and the output stream is used for writing into the file (write operation).

FileStream Class

Syntax: FileStream <object_name> = new FileStream(<file_name>,<FileMode Enumerator>, <FileAccess Enumerator>, <FileShare Enumerator>);

Eg: FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read, FileShare.Read);

StreamReader Class
Eg: (StreamReader sr = new StreamReader("names.txt")

StreamWriter Class
Eg: (StreamWriter sw = new StreamWriter("names.txt")

BinaryReader Class
Eg: (BinaryReader br = **new** BinaryReader(File.Open("e:\\binaryfile.dat", FileMode.Open)

BinaryWriter Class

Eg:(BinaryWriter writer = **new** BinaryWriter(File.Open("e:\\binaryfile.dat", FileMode.Create)

DirectoryInfo Class

Eg:

//creating a DirectoryInfo object

DirectoryInfo mydir = new DirectoryInfo(@"c:\Windows");

// getting the files in the directory, their names and size

FileInfo [] f = mydir.GetFiles();

<u>Collections</u>

Collection classes are specialized classes for **data storage and retrieval**. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces.

It Includes:

- ArrayList(It can have duplicate elements)

  Syntax: ArryList names = **new** ArrayList<**string**>();

```
 names.Add("Sonoo Jaiswal");
 names.Add("Ankit");
 names.Add("Peter");
 //Display
foreach(String name in names)
```

Operations

Capacity, count, sort

- Hashtable(It does not store duplicate elements)

  Syntax: hashtable names= new hashtable();

```
 names.Add("Sonoo Jaiswal");
 names.Add("Ankit");
 names.Add("Peter");
 //Display
foreach(String name in names)
```

- Stack (class is used to push and pop elements. It uses the concept of Stack that arranges elements in LIFO order. It can have duplicate elements)

  Syntax: Stack names = **new** Stack();

```
names.Push("Sonoo");
names.Push("Peter");
names.Push("James");
```

```
 names.Push("Ratan");
//Display use foreachloop
```

- Queue(lass is used to Enqueue and Dequeue elements. It uses the concept of Queue that arranges elements in FIFO order. It can have duplicate elements.)

  Syntax: Queue names = new Queue ();

  ```
      names.Enqueue("Sonoo");

      names.Enqueue("Peter");

      names.Enqueue("James");

      names.Enqueue("Ratan");
     //Display using foreach loop
  ```

- Bit array (It represents an array of the **binary representation** using the values 1 and 0)
  Syntax
  ```
  BitArray ba1 = new BitArray(8);  BitArray ba2 = new BitArray(8);
  byte[] a = { 60 };
  byte[] b = { 13 };
  ```

- Sorted (It uses a **key** as well as an **index** to access the items in a list.A sorted list is a combination of an array and a hash table. It contains a list of items that can be accessed using a key or an index.)
  Syntax:
  ```
  SortedList sl = new SortedList();
  sl.Add("001", "Zara Ali");
  sl.Add("002", "Abida Rehman");
  sl.Add("003", "Joe Holzner");  sl.Add("004", "Mausam Benazir Nur");  sl.Add("005",
  "M. Amlan");
  sl.Add("006", "M. Arif");
  sl.Add("007", "Ritesh Saikia");
  //Display using Foreach loop
  ```