1. **What is class?**

* A blueprint or template for creating objects.
* Defines the properties (attributes) and behaviors (methods) that objects of the class will have.

1. **What is object?**

* An instance of a class.
* Represents a real-world entity and encapsulates data and functionality.

1. **Define Encapsulation?**

Encapsulation is the process of binding/combining the data members and member functions into a single unit.

* We declare data members as private and use getter and setter methods to get the data from orivate variables

1. **How do you define Abstraction?**

It is known as the process of hiding the implementation details and highlighting the essential features.

* We only declare the implementation part of method and declaration is done by overriding the method in inherited class.

using System;

// Abstract class representing a shape

public abstract class Shape

{

// Abstract method to be implemented by derived classes

public abstract void Draw();

// Concrete method

public void DisplayArea()

{

Console.WriteLine("Area calculation is specific to each shape.");

}

}

// Concrete class 1: Circle

public class Circle : Shape

{

private double radius;

public Circle(double radius)

{

this.radius = radius;

}

// Implementing the abstract method from the base class

public override void Draw()

{

Console.WriteLine($"Drawing a circle with radius {radius}");

}

}

// Concrete class 2: Rectangle

public class Rectangle : Shape

{

private double length;

private double width;

public Rectangle(double length, double width)

{

this.length = length;

this.width = width;

}

// Implementing the abstract method from the base class

public override void Draw()

{

Console.WriteLine($"Drawing a rectangle with length {length} and width {width}");

}

}

class Program

{

static void Main()

{

// Creating instances of derived classes

Circle myCircle = new Circle(5);

Rectangle myRectangle = new Rectangle(4, 6);

// Calling the abstract method, which is implemented differently in each class

myCircle.Draw();

myRectangle.Draw();

// Calling the concrete method from the base class

myCircle.DisplayArea();

myRectangle.DisplayArea();

}

}

1. **Interface :**

Interfaces in C# are essential because they allow for **multiple inheritance**, enabling a class to implement multiple contracts.

Abstract classes support single inheritance and have implementation details, while interfaces provide a pure abstraction with no implementation, promoting flexibility, and facilitating the creation of more modular and loosely coupled code.

**using System;**

**public interface IDrawable { void Draw(); } // First Interface**

**public interface IResizable { void Resize(int width, int height); } // Second Interface**

**public class Rectangle : IDrawable, IResizable // Class implementing both interfaces**

**{**

**public void Draw()**

**{**

**Console.WriteLine("Drawing a rectangle");**

**}**

**public void Resize(int width, int height)**

**{**

**Console.WriteLine($"Resizing the rectangle to {width} x {height}");**

**}**

**}**

**public class Program {**

**public static void Main() {**

**Rectangle myRectangle = new Rectangle(); // Creating an instance of the class**

**myRectangle.Draw(); // Calling methods from both interfaces**

**myRectangle.Resize(100, 150);**

**} }**

1. **Inheritance.**

* In C#, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.
* In C#, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

1. **What do you understand by Polymorphism?**

Polymorphism means "many forms". This allows us to perform a single action in different ways.

* We use virtual keyword in base class and override keyword in inherited class to use the same action(method) in different child(inherited) classes.

**Example**:

using System;

public class Shape

{

public virtual void draw() { Console.WriteLine("drawing..."); }

}

public class Rectangle: Shape

{

public override void draw() { Console.WriteLine("drawing rectangle..."); }

}

public class Circle : Shape

{

public override void draw() { Console.WriteLine("drawing circle..."); }

}

public class TestPolymorphism

{

public static void Main()

{

Shape s= new Shape();

s.draw();

Shape s1 = new Rectangle();

s1.draw();

Shape s2 = new Circle();

s2.draw();

}

}

OUTPUT-

drawing...

drawing rectangle...

drawing circle...

* One can define two types of Polymorphism-
* Compile time Polymorphism, also known as Overloading
* Runtime polymorphism, also known as Overriding

1. **Method OverLoading:**

A class to have multiple methods having the same name, but with a different number or type of parameters.

1. **Method Overriding:**

A specific implementation for a method in derived class that is already defined in its superclass.

1. **Sealed**:

C# sealed keyword applies restrictions on the class and method.

* If you create a sealed class, it cannot be derived.
* If you create a sealed method, it cannot be overridden.
* Structs are implicitly sealed therefore they can't be inherited.

1. **Constructor:**

* Constructors are methods that are automatically executed every time you create an object.
* The purpose of a constructor is to construct an object and assign values to the object's members.
* A constructor takes the same name as the class to which it belongs, and does not return any values.

1. **C# Math**

* Inbuilt Math methods in c# methods that allows you to perform mathematical tasks on numbers.
* Math.Max(x,y) - Math.Max(5, 10); output: 10
* Math.Min(x,y) - Math.Min(5, 10); output: 5
* Math.Sqrt(x) - Math.Sqrt(64); output: 8
* Math.Abs(x) - Math.Abs(-4.7); output: 4.7
* Math.Round() - Math.Round(9.99); output: 10
* Math.Ceiling() -Math.Ceiling(9.2); output:10
* Math.Floor() -Math.Floor(9.6); output:9

1. **C# strings**
2. **Boxing Unboxing**

The C# Type System contains three data types:

* Value Types (int, char, etc),
* Reference Types (object) and
* Pointer Types.
* Basically, Boxing converts a Value Type variable into a Reference Type variable, and Unboxing achieves the vice-versa

**Boxing :**

The process of converting a Value Type variable (char, int etc.) to a Reference Type variable (object) is called Boxing.

* Boxing is an implicit conversion process in which object type (super type) is used.
* Value Type variables are always stored in Stack memory, while Reference Type variables are stored in Heap memory.

**Example** :

int num = 23; // 23 will assigned to num

Object Obj = num; // Boxing

**Unboxing:**

The process of converting a Reference Type variable into a Value Type variable is known as Unboxing.

* It is an explicit conversion process.

**Example** :

int num = 23; // value type is int and assigned value 23

Object Obj = num; // Boxing

int i = (int)Obj; // Unboxing

1. **Difference between Break and Continue Statement.**

* **Continue statement** - Used in jumping over a particular iteration and getting into the next iteration of the loop.
* **Break statement** - Used to skip the next statements of the current iteration and come out of the loop.

1. **C# access modifiers.**

The C# access modifiers are -

**Private Access Modifier** - A private attribute or method is one that can only be accessed from within the class.

**Public** **Access Modifier** - When an attribute or method is declared public, it can be accessed from anywhere in the code.

**Internal** **Access Modifier** - When a property or method is defined as internal, it can only be accessible from the current assembly point of that class.

**Protected** Access Modifier - When a user declares a method or attribute as protected, it can only be accessed by members of that class and those who inherit it.

1. **Type Conversion Methods**

* It is also possible to convert data types explicitly by using built-in methods, such as
* Convert.ToBoolean,
* Convert.ToDouble,
* Convert.ToString,
* Convert.ToInt32 (int) and
* Convert.ToInt64 (long)

**Example:**

int myInt = 10;

double myDouble = 5.25;

bool myBool = true;

Console.WriteLine(Convert.ToString(myInt)); // convert int to string

Console.WriteLine(Convert.ToDouble(myInt)); // convert int to double

Console.WriteLine(Convert.ToInt32(myDouble)); // convert double to int

Console.WriteLine(Convert.ToString(myBool)); // convert bool to string

1. **C# Typecasting**

Type casting is when you assign a value of one data type to another type.

In C#, there are two types of casting:

**Implicit Casting (automatically)** –

converting a smaller type to a larger type size

char -> int -> long -> float -> double

**Explicit Casting (manually)** –

converting a larger type to a smaller size type

double -> float -> long -> int -> char

* Explicit casting must be done manually by placing the type in parentheses in front of the value:

**Collections**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. **Generic classes in collections(System.Collections.Generic classes)**

* List -- List<object> list1 = new List<object>
* Stack
* Queue
* LinkedList
* HashSet
* SortedSet
* Dictionary - Dictionary<**string**, **string**> names = **new** Dictionary<**string**, **string**>();
* SortedDictionary -
* SortedList

1. **Non-Generic classes (System.Collections classes)**

These classes are legacy. The System.Collections namespace has following classes:

* ArrayList - ArrayList arrlist= new ArrayList();
* Stack
* Queue
* Hashtable

1. **List**

* It can store duplicate data.
* To get count of

1. **HashSet** - It only stores unique data
2. **SortedSet**  - It only store unique data in acsending order.
3. **Stack**

* It follows (LIFO)Last In First Out
* (Push,Pop,Peek).
* It does not use Collections Initializer

1. **Queue**

* It follows (FIFO) First In First Out
* (Enqueue,Dequeue,Peek).
* It does not use Collections Initializer.

1. **LinkedList**

* It can store duplicate data.
* It does not use  Add method or Collections Initializer.
* It uses (AddFirst, AddLast).
* To add one more way using LinkedListNode .Find()
* (AddBefore(),AddAfter()).
* Unlike List, you cannot create LinkedList using Collection initializer.

1. **Dictionary**

* It uses concept of <Key , value>. I stores values on the basis of unique keys.

Dictionary < int, string > dict = new Dictionary<int,string>();

* It cannot have duplicate keys
* But it can hace duplicate Values
* We can also pass list inside dictionary

Dictionary < int, List<T> > dict = new Dictionary<int,string>();

1. **SortedDictionary** -

* It uses concept of <Key , value>.
* It stores values on the basis of unique keys.
* It sorts the Dictionary in acending order with respect to key.

1. **SortedList** - It is similar to Sorted Dictionary.
2. **Differnece between Array and array list**

Arrays have fixed size that is set when they are created , while arraylists can shrink and grow as the items are added / removed

1. **What is an array?**

* Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
* To declare an array, define the variable type with square brackets:

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

* Console.WriteLine(cars[0]); // Outputs Volvo
* To find out how many elements an array has, use the Length property

i.e cars.Length();

**Other Ways to Create an Array:**

// Create an array of four elements, and add values later

string[] cars = new string[4];

// Create an array of four elements and add values right away

string[] cars = new string[4] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements without specifying the size

string[] cars = new string[] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements, omitting the new keyword, and without specifying the size

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

1. **C# Exception Handling**

* Exception Handling in C# is a process to handle runtime errors. We perform exception handling so that normal flow of the application can be maintained even after runtime errors.
* In C#, exception is an event or object which is thrown at runtime.
* All exceptions the derived from System.Exception class.
* It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

**Common exception classes.**

|  |  |
| --- | --- |
| **Exception** | **Description** |
| System.DivideByZeroException | handles the error generated by dividing a number with zero. |
| System.NullReferenceException | handles the error generated by referencing the null object. |
| System.InvalidCastException | handles the error generated by invalid typecasting. |
| System.IO.IOException | handles the Input Output errors. |
| System.FieldAccessException | handles the error generated by invalid private or protected field access. |

1. **C# Exception Handling Keywords**

In C#, we use 4 keywords to perform exception handling:

try

catch

finally, and

throw

1. **Try-catch**

The try-catch block is used to enclose a section of code where an exception might occur. If an exception occurs within the try block, it is caught and handled by the catch block, preventing the program from terminating abruptly.

1. **How can you handle multiple exceptions in a single catch block?**

We can use multiple catch blocks to handle different types of exceptions. Catch blocks are evaluated in order, and the first one that matches the type of the thrown exception is executed.

1. **Finally block**

Finally block executes no matter what exception is handled or not

1. Generics
2. Enum
3. Regular expression
4. Exceptions

|  |  |  |
| --- | --- | --- |
| **No.** | **Java** | **C#** |
| 1) | Java is a **high level, robust, secured and object-oriented programming** language developed by Oracle. | C# is an **object-oriented programming** language developed by Microsoft that runs on .Net Framework. |
| 2) | Java programming language is designed to be run on a Java platform, by the help of **Java Runtime Environment (JRE).** | C# programming language is designed to be run on the **Common Language Runtime (CLR).** |
| 3) | Java type safety is safe. | C# type safety is unsafe. |
| 4) | In java, built-in data types that are passed by value are called **primitive types.** | In C#, built-in data types that are passed by value are called **simple types.** |
| 5) | Arrays in Java are direct specialization of **Object.** | Arrays in C# are specialization of **System.** |
| 6) | Java does not support **conditional compilation.** | C# supports conditional compilation using preprocessor directives. |
| 7) | Java doesn't support goto statement. | C# supports goto statement. |
| 8) | Java doesn't support **structures and unions.** | C# supports structures and unions. |
| 9) | Java supports checked exception and unchecked exception. | C# supports unchecked exception. |