UNIT-3

OBJECT ORIENTED PROGRAMMING CONCEPTS

3.1 CLASS AND OBJECT

What is a class in Java?

A class is a group of objects which have common properties.

It is a template or blueprint from which objects are created.

It is a logical entity. It can't be physical.

A class in Java can contain:

- Fields / Variables
- Methods
- Constructors
- Blocks
- Nested class and interface

Syntax to declare a class:

```
class class_name{
  //  field;
   // method;
}
```

What is an object in java?

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object is an instance of a class.

An object has two characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.

Example : Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.

- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

Example of Class and Object :

```
class Student{
 int id;
 String name;
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Output:

0

null

3.2 CONSTRUCTOR AND TYPES OF CONSTRUCTORS

In Java, a constructor is a special method that is used to initialize objects of a class. It is called automatically when an object is created using the **new** keyword. Constructors have the same name as the class and do not have a return type, not even **void.**

3.2.1 TYPES OF CONSTRUCTORS

1.Default Constructor:

A default constructor is provided by Java if you do not explicitly define any constructor in your class. It has no parameters and does not perform any initialization. Its purpose is to initialize the object with default values (e.g., setting numeric properties to 0 and reference properties to **null**).

EXAMPLE:

```
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
```

```
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output :

Bike is created

2. Parameterized Constructor:

A parameterized constructor is a constructor that takes one or more parameters. It allows you to initialize the object with specific values at the time of creation.

EXAMPLE:

```
public class Person {
 private String name;
 private int age;

  // Parameterized constructor
public Person(String n, int a) {
   name = n;
   age = a;
 }}
```

In the example above, the Person class has a parameterized constructor that takes name and age as parameters and initializes the corresponding instance variables.

3.Copy Constructor:

A copy constructor is a constructor that creates a new object by copying the state of another object of the same class. It is useful when you want to create a new object with the same values as an existing object.

EXAMPLE:

```
public class Student {
  private String name;
  private int age;

  // Copy constructor
 public Student(Student other) {
name = other.name;
age = other.age;
```

```
    }
}
```

In this example, the Student class has a copy constructor that takes another Student object as a parameter and copies its name and age values to create a new Student object.

3.3 METHOD AND METHOD OVERLOADING

Method is a collection of statements that are grouped together to perform a specific task. Methods are used to encapsulate code and provide reusability and modularity in your programs. They are defined inside classes and can be called to perform the actions they define.

Syntax:

```
modifier returnType methodName(parameterList) {
// Method body
}
```

Example:

```
public class Calculator{
    //method of addition
     public int add(int num1, int num2) {
      return num1 + num2;
    }
    public static void main(String args[]){
            //Creating an object of Calculator class
            Calculator obj = new Calculator();
            //Calling an method
            System.out.println(obj.add(1,2));
}
}
```

Output :

3

Types of Method :

1. no return no parameter :

Example :

```
void display(){
    System.out.println("Hello world");
}
```

2. no return with parameter :

Example :

```
void display(String msg){
    System.out.println("Your message is : "+msg);
}
```

3. with return no parameter :

Example :

```
int number;
int returnNumber(){
     number = 5;
    return number;
}
```

4. with return with parameter :

Example :

```
int add(int num1, int num2) {
      return num1 + num2;
}
```

5. Recursive function :

A method in java that calls itself is called recursive method.

It makes the code compact but complex to understand.

Syntax :

```
returntype methodname(){
//code to be executed
methodname();//calling same method
}
```

Example :

```
public class RecursionExample3 {
   int factorial(int n){
        if (n == 1)
          return 1;
        else
```

```
        return(n * factorial(n-1));
    }
public static void main(String[] args) {
RecursionExample3 obj = new RecursionExample3();
System.out.println("Factorial of 5 is: "+ obj.factorial(5));
}
}
```

Output :

Factorial of 5 is: 120

3.3.1 METHOD OVERLOADING

Method overloading is a feature in Java that allows you to define multiple methods with the same name but different parameters within the same class. The methods must have different parameter types, different numbers of parameters, or both.

USES OF METHOD OVERLOADING:

Method overloading when a couple of methods are needed with conceptually similar functionality with different parameters.

Memory can be saved by implementing method overloading.

EXAMPLE:

```
public class MathUtils {
  public int add(int num1, int num2) {
   return num1 + num2;
  }

  public double add(double num1, double num2) {
   return num1 + num2;
  }
}
```

3.4 THIS KEYWORD

'this' keyword is a reference to the current instance of a class. It is used within an instance method or constructor to refer to the object on which the method or constructor is being invoked**.**

Here are a few common uses of the **this** keyword:

1.Accessing Instance Variables:

You can use this to refer to the current object's instance variables when there is a naming conflict between instance variables and method parameters or local variables.

```
public class Person {
  private String name;

   public void setName(String name) {
 // Using "this" to refer to the instance variable
  this.name = name;
  }
}
```

In this example, the **this.name** refers to the instance variable name, while name alone refers to the method parameter. It helps differentiate between the two variables and assign the value to the instance variable.

2.Invoking Another Constructor:

In a constructor, you can use this to invoke another constructor of the same class. It is useful when you want to reuse the initialization logic of one constructor in another constructor.

```
public class Car {
 private String brand;
private String color;
 private int year;

 public Car(String brand, String color) {
this.brand = brand;
this.color = color;
  }

    public Car(String brand, String color, int year) {
this(brand, color);  // Invoking the two-argument constructor using "this"
this.year = year;
}
}
```

In this example, the second constructor invokes the first constructor using **this(brand, color).** It allows the common initialization code to be shared between the constructors.

3.Returning The Current Instance:

You can use this to return the current instance of the object from a method. It is commonly used for method chaining or fluent interfaces.

```
public class Person {
```

```
private String name;


 public PersonWithName(String name) {
  this.name = name;
return this;
  } }
```

In this example, the PersonWithName method sets the name and returns the current instance (**this**). It allows method calls to be chained together for a more concise and readable code.

The “this” keyword always refers to the current instance.

The “this” keyword helps in removing the ambiguity of references and allows to refer to the current instance within the running program.

Keyword this is used with variables as "this. variable" and specifies the number of member variable of this instance of that class.

4. this can be used to invoke current class method (implicitly).

5. this can be passed as an argument in the method call.

6. this can be passed as argument in the constructor call.

3.5 STATIC KEYWORD

The **static keyword** is used to declare members (variables and methods) that belong to the class itself rather than to instances (objects) of the class. It allows you to access these members without creating an instance of the class.

Here are some key uses of the **static** keyword:

1.Static Variables:

When a variable is declared as static, it is called a static variable or a class variable. The static variable is shared among all instances of the class. There is only one copy of the static variable that is shared across all objects of the class.

```
public class Counter {
private static int count; // static variable


  public Counter() {
  count++; // Accessing and modifying the static variable
  }


   public static void printCount() { // static method
   System.out.println("Count: " + count); // Accessing the static variable
}
```

```
}
```

In this example, the **count** variable is declared as **static**, meaning it is shared among all instances of the **Counter** class. Each time a new **Counter** object is created, the constructor is called and increments the **count** variable. The **printCount** method is also declared as **static** and can be called without creating an object.

2.Static Methods:

When a method is declared as static, it is called a static method. Static methods belong to the class rather than individual objects. They can be invoked directly on the class itself, without the need to create an instance.

```
public class MathUtils {
public static int add(int a, int b) { // static method
  return a + b; }
        }
```

In this example, the **add** method is declared as **static**. It can be called using the class name, without creating an instance of the **MathUtils** class.

```
int sum = MathUtils.add(5, 3);
System.out.println(sum);
```

3.Static Block:

A static block is a block of code enclosed in curly braces {} that is executed when the class is loaded into memory. It is used to initialize static variables or perform any other static initialization tasks.

```
public class MyClass {
  private static int x;

  static {
   x = 10;
System.out.println("Static block executed");
}
}
```

In this example, the static block sets the value of the static variable **x** to 10. It is executed only once when the class is loaded, before any other static or instance members are accessed.

3.6 STRING CLASS AND ITS METHODS

The **String** class represents a sequence of characters. It is a built-in class and provides various methods to manipulate and work with **strings**.

Here are some commonly used methods of the **String** class:

1.Length: The **length**() method returns the length (number of characters) of a string.

```
String str = "Hello, World!";
int length = str.length();
```

Output: length = 13

2.Concatenation: The **concat()** method concatenates two strings and returns a new string.

```
String str1 = "Hello";
String str2 = "World";
String result = str1.concat(str2);
```

Output: result = "HelloWorld"

Alternatively, you can use the + operator for string concatenation.

```
String result = str1 + str2;
```

Output result = "HelloWorld"

3.Substring: The **substring()** method returns a substring of the original string based on the specified starting and ending indexes.

```
String str = "Hello, World!";
String substr = str.substring(7, 12);
```

Output: substr = "World"

4.Comparison: The **equals()** method checks if two strings have the same content.

```
String str1 = "Hello";
String str2 = "Hello";
boolean isEqual = str1.equals(str2);
```

Output: isEqual = true

The **equalsIgnoreCase()** method compares strings while ignoring case differences.

5.Conversion: The **toUpperCase**() and **toLowerCase**() methods convert the string to uppercase and lowercase, respectively.

```
String str = "Hello, World!";
String uppercase = str.toUpperCase();
String lowercase = str.toLowerCase();
```

Output: uppercase = "HELLO, WORLD!"

lowercase = "hello, world!"

6.Searching: The **indexOf()** method returns the index of the first occurrence of a specified substring within a string.

```
String str = "Hello, World!";
int index = str.indexOf("World");
```

Output: index = 7

7.Splitting: The **split()** method splits a string into an array of substrings based on a specified delimiter.

```
String str = "Hello,World,Java";

String[] parts = str.split(",");
```

Output: parts = ["Hello", "World", "Java"]

8.Replacement: The **replace()** method replaces all occurrences of a specified character or substring with another character or substring.

```
String str = "Hello, World!";

String replaced =str.replace("Hello", "Hi");
```

Output: replaced = "Hi, World!"

Difference between String and String Buffer

No.	String	StringBuffer
1)	The String class is immutable.	The StringBuffer class is mutable.
2)	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate t strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.
4)	String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.
5)	String class uses String constant pool.	StringBuffer uses Heap memory

3.7 I/O CLASSES AND FILE HANDLING

I/O (Input/Output) classes are used for reading input from various sources and writing output to different destinations. These classes provide functionality for file handling, reading from and writing to streams, and interacting with the user via the console.

Here are some commonly used I/O classes and file handling operations in Java:

1.File class: The **File** class is used to represent files and directories. It provides methods to create, delete, rename, and query file properties.

```
import java.io.File;


// Creating a File object
```

```
File file = new File("path/to/file.txt");


// Checking if the file exists
boolean exists = file.exists();


// Creating a directory
File directory = new File("path/to/directory");
directory.mkdir();
```

2.Scanner class: The Scanner class is used for reading user input from the console or parsing formatted data from different sources.

```
import java.util.Scanner;


// Reading user input from the console
Scanner scanner = new Scanner(System.in);
System.out.print("Enter your name: ");
String name = scanner.nextLine();
System.out.println("Hello, " + name);


// Parsing data from a file
File file = new File("data.txt");
Scanner fileScanner = new Scanner(file);
while (fileScanner.hasNext()) {
  String data = fileScanner.nextLine();
 // Process the data
}
fileScanner.close();
```