**UNIT-III**

**JAVA BASICS**

**Classes and objects:**

**Class**

1. Class is a set of object which shares common characteristics/ behavior and common properties/ attributes.
2. Class is not a real world entity. It is just a template or blueprint or prototype from which objects are created.
3. Class does not occupy memory.
4. Class is a group of variables of different data types and group of methods.
5. A class is a blueprint for the object. Before we create an object, we first need to define the class.

A class in java can contain:
• data member
• method
• constructor
• nested class and
• interface

**To create a class, use the keyword class:**

Create a class named "Main" with a variable x:

**public class Main {**

 **int x = 5;**

**}**

**Example:**

For our bicycle object, we can create the class as

class Bicycle {

 // state or field

 private int gear = 5;

 // behavior or method

 public void braking() {

 System.out.println("Working of Braking");

 }

}

**Object:**

**Object** − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

An object has three characteristics:

* **State:** represents the data (value) of an object.
* **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
* **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For 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.

**An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

**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:**

className object = new className();

// for Bicycle class

Bicycle sportsBicycle = new Bicycle();

Bicycle touringBicycle = new Bicycle();

**Strings**

* Strings are used for storing text.
* A String variable contains a collection of characters surrounded by double quotes:
* In [Java](https://www.javatpoint.com/java-tutorial), string is basically an object that represents sequence of char values. An [array](https://www.javatpoint.com/array-in-java) of characters works same as Java string.

For example:

1. char[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=new String(ch);

String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The **java.lang.String** class is used to create a string object.

Once created, a string is immutable -- its value cannot be changed.

A string is sequence of characters. A class is a user-defined template for creating an object. A string class is a user-defined template for creating and manipulating string objects, which are sequences of characters.

Strings are **used for storing text/characters**

 For example, "Hello World" is a string of characters.

**How to create a string object?**

There are two ways to create String object:

1. By string literal
2. By new keyword

**1) String Literal**

Java String literal is created by using double quotes. For Example:

1. String s="welcome";

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

String s1="Welcome";

String s2="Welcome";//It doesn't create a new instance

**2) By new keyword**

1. String s=new String("Welcome");//creates two objects and one reference variable

**Using new keyword**

 **String s = new String(“Welcome”);**

In such a case, JVM will create a new string object in normal (non-pool) heap memory and the literal “Welcome” will be placed in the string constant pool. The variable s will refer to the object in the heap (non-pool).

**Syntax:**

**<String\_Type> <string\_variable> = "<sequence\_of\_string>";**

**Java String Example**

StringExample.java

 public class StringExample{

 public static void main(String args[]){

 String s1="java";//creating string by Java string literal

 char ch[]={'s','t','r','i','n','g','s'};

 String s2=new String(ch);//converting char array to string

 String s3=new String("example");//creating Java string by new keyword

 System.out.println(s1);

 System.out.println(s2);

 System.out.println(s3);

 }}

**Inheritance**

* Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).
* The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

**Why use inheritance in java**

* For Method Overriding (so runtime polymorphism can be achieved).
* For Code Reusability.

**Terms used in Inheritance**

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

**The syntax of Java Inheritance**

 **class Subclass-name extends Superclass-name**

 **{**

 **//methods and fields**

 **}**

* The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
* A class which is inherited is called a parent or superclass, and the new class is called child or subclass.

## Types of inheritance

There are five types of inheritance.

### 1. Single Inheritance

In single inheritance, a single subclass extends from a single superclass. For example,



### 2. Multilevel Inheritance

In multilevel inheritance, a subclass extends from a superclass and then the same subclass acts as a superclass for another class. For example,



### 3. Hierarchical Inheritance

In hierarchical inheritance, multiple subclasses extend from a single superclass. For example,



### 4. Multiple Inheritance

In multiple inheritance, a single subclass extends from multiple superclasses. For example,



### 5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. For example,



**Advantages Of Inheritance in Java:**

1. Code Reusability: Inheritance allows for code reuse and reduces the amount of code that needs to be written. The subclass can reuse the properties and methods of the superclass, reducing duplication of code.
2. Abstraction: Inheritance allows for the creation of abstract classes that define a common interface for a group of related classes. This promotes abstraction and encapsulation, making the code easier to maintain and extend.
3. Class Hierarchy: Inheritance allows for the creation of a class hierarchy, which can be used to model real-world objects and their relationships.
4. Polymorphism: Inheritance allows for polymorphism, which is the ability of an object to take on multiple forms. Subclasses can override methods of the superclass, which allows them to change their behavior in different ways.

**Disadvantages of Inheritance in Java:**

1. Complexity: Inheritance can make the code more complex and harder to understand. This is especially true if the inheritance hierarchy is deep or if multiple inheritances is used.
2. Tight Coupling: Inheritance creates a tight coupling between the superclass and subclass, making it difficult to make changes to the superclass without affecting the subclass.

**Polymorphism and Interfaces**

**Polymorphism**

* Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
* Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
* **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.)

### Example

class Animal {

 public void animalSound() {

 System.out.println("The animal makes a sound");

 }

}

class Pig extends Animal {

 public void animalSound() {

 System.out.println("The pig says: wee wee");

 }

}

class Dog extends Animal {

 public void animalSound() {

 System.out.println("The dog says: bow wow");

 }

}

**Interfaces**

* An interface in Java is a blueprint of a class. It has static constants and abstract methods.
* The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
* In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

## Why use Java interface?

There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

**How to declare an interface?**

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

**Syntax:**

 **interface <interface\_name>{**

 **// declare constant fields**

 **// declare methods that abstract**

 **// by default. }**

**Java Interface Example: Drawable**

 **//Interface declaration: by first user**

 **interface Drawable{**

 **void draw();**

 **}**

 **//Implementation: by second user**

 **class Rectangle implements Drawable{**

 **public void draw(){System.out.println("drawing rectangle");}**

 **}**

 **class Circle implements Drawable{**

 **public void draw(){System.out.println("drawing circle");}**

 **}**

 **//Using interface: by third user**

 **class TestInterface1{**

 **public static void main(String args[]){**

 **Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()**

 **d.draw();**

 **}}**

**Regular Expressions**

* A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.
* A regular expression can be a single character, or a more complicated pattern.
* Regular expressions can be used to perform all types of text search and text replace operations.
* Java does not have a built-in Regular Expression class, but we can import the java.util.regex package to work with regular expressions. The package includes the following classes:
* Pattern Class - Defines a pattern (to be used in a search)
* Matcher Class - Used to search for the pattern
* PatternSyntaxException Class - Indicates syntax error in a regular expression pattern

**Example of Java Regular Expressions**

import java.util.regex.\*;

 public class RegexExample1{

 public static void main(String args[]){

 //1st way

 Pattern p = Pattern.compile(".s");//. represents single character

 Matcher m = p.matcher("as");

 boolean b = m.matches();

 //2nd way

 boolean b2=Pattern.compile(".s").matcher("as").matches();

 //3rd way

 boolean b3 = Pattern.matches(".s", "as");

 System.out.println(b+" "+b2+" "+b3);

 }}

**Exception Handling**

**What is Exception in Java?**

An exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

**What is Exception Handling?**

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

**Types of Exceptions**



A list of different approaches to handle exceptions in Java.

* try...catch block
* finally block
* throw and throws keyword

## Java try...catch block

The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.

**try {**

 **// code**

**}**

**catch(Exception e) {**

 **// code}**

Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by a catch block. When an exception occurs, it is caught by the catch block. The catch block cannot be used without the try block.

**Example: Exception handling using try...catch**

class Main {

 public static void main(String[] args) {

 try {

 // code that generate exception

 int divideByZero = 5 / 0;

 System.out.println("Rest of code in try block");

 }

 catch (ArithmeticException e) {

 System.out.println("ArithmeticException => " + e.getMessage());

 }

 }

}

## 2. Java finally block

* The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
* In Java, the finally block is always executed no matter whether there is an exception or not.
* The finally block is optional. And, for each try block, there can be only one finally block.

The basic syntax of finally block is:

**try {**

 **//code**

**}**

**catch (ExceptionType1 e1) {**

 **// catch block**

**}**

**finally {**

 **// finally block always executes**

**}**

If an exception occurs, the finally block is executed after the try...catch block. Otherwise, it is executed after the try block. For each try block, there can be only one finally block.

**Example: Java Exception Handling using finally block**

class Main {

 public static void main(String[] args) {

 try {

 // code that generates exception

 int divideByZero = 5 / 0;

 }

 catch (ArithmeticException e) {

 System.out.println("ArithmeticException => " + e.getMessage());

 }

 finally {

 System.out.println("This is the finally block");

 }

 }

}

## 3. Java throw and throws keyword

* The Java throw keyword is used to explicitly throw a single exception.
* When we throw an exception, the flow of the program moves from the try block to the catch block.

**Example: Exception handling using Java throw**

class Main {

 public static void divideByZero() {

 // throw an exception

 throw new ArithmeticException("Trying to divide by 0");

 }

 public static void main(String[] args) {

 divideByZero();

 }

}

**Throws**

The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

**Example: Java throws keyword**

import java.io.\*;

class Main {

 // declareing the type of exception

 public static void findFile() throws IOException {

 // code that may generate IOException

 File newFile = new File("test.txt");

 FileInputStream stream = new FileInputStream(newFile);

 }

 public static void main(String[] args) {

 try {

 findFile();

 }

 catch (IOException e) {

 System.out.println(e);

 }

 }

}