1. **Inheritance**

   Here, properties of a class (super class/ parent class) to be inherited by another class (sub class / child class). Various types of inheritance are single, multilevel, hierarchical, hybrid. Multiple inheritance is not supported in Java. Inheritance supports code reuse.

   Task TO DO:

   - Modify the given example to form an example of hybrid inheritance.
   - what is **instanceof** Operator?
   - How is protected access modifier useful in inheritance?

2. **Polymorphism:**

   Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance. Overloading exihibits compile-time polymorphism , while Overriding shows runtime Overloading.

3. **Overloading:**

   The same function name can be used for different method(with different argument sets). In the program below, overload the add() function for adding the combination of integer and double numbers as shown in the main function.

4. **Simple Overriding:**

   In a class hierarchy when a method in a sub-class has the same name and type signature as in a method in its super class, then the method in the sub-class is said to override the method in the super class.

5. **Overriding with runtime polymorphism (Dynamic Method Dispatch):**

   - Upcasting: A call is made to an overridden method of child class via its parent type reference.
   - During Upcasting, the type of the object determines which method to be invoked. Making of this decision happens during runtime (by JVM) is called as Run time polymorphism.
   - Each time during the call of show() method, which method gets executed (Maps, WebMaps or PaperMaps) depends on the type of the object being referred to at the time of the call (i.e. during runtime).

```java
class Maps{

    void show()
    {
        System.out.println("Maps");
    }
}

class WebMaps extends Maps {
    void show()
    {
        System.out.println("Web Maps: Bing Map, Google Map");
    }
}


class  PaperMaps extends Maps {
    void show()
    {
        System.out.println("Paper Maps: All Maps printed on paper");
    }
}

public class RunTimePolyDemo{

    public static void main (String[] args) {

        // We can have references of Maps type.
        Maps m = new Maps();
        WebMaps w = new WebMaps();
        PaperMaps p = new PaperMaps();

        m = w ; // upcasting
        m.show();

        m = p; // upcasting
        m.show();
        // the show() method is called via the reference variable of the
        //Parent class. Since it refers to the subclass object and subclass
        // method overrides the Parent class method, the subclass method
        // is invoked at runtime.
        //Each time during the call of show() method (e.g line 35 or line 38 ),
        // which method gets executed (show() mehtod in Maps, WebMaps or PaperMaps)
        // depends on the type of the object being referred to at the time of the
        // call (i.e. during runtime):
    }
}
```

6. Abstract classes and abstract methods:

Abstract class is an incomplete class. It cannot be instantiated. An abstract class must consist of at least one abstract method. An abstract method is a method that is just declared and have no implementation.

```
abstract class Maps{

    abstract void show();
}
// The class which inherits this base class "Maps" must provide
//the body of show() method, otherwise that sub class will also be abstract.

class WebMaps extends Maps {

    void show()
    {
        System.out.println("Web Maps: Bing Map, Google Map");
    }
}


public class AbstractClassDemo{

    public static void main (String[] args) {

        // Maps m = new Maps();
        // Uncommenting the above statelemt will induce
        // compiler error as it to create an
        // instance of abstract class ( which is illegal).

        WebMaps w = new WebMaps();

        w.show();
        }
}
```

7. Interface

- The interface declares a set of public method (no implementation) and fields
- A class that implements the interface provides an implementation for each of the methods the interface defines.
- **Compare interfaces with inheritance?**

```java
// Write two interfaces "AnimalInterface" and "HumanInterface".
// Create a class "PersonClass" to implement the interface.

/* File name : HumanInterface.java */
interface HumanInterface {
    String type ="Human"; // constant
    public void study(String qualification);
}

/* File name : AnimalInterface.java */
interface AnimalInterface {

    public void eat();
    public void travel();
}

/* File name : PersonClass.java */
class PersonClass implements AnimalInterface, HumanInterface{

    public void eat(){
        System.out.println("Person eats");
    }

    public void travel(){
        System.out.println("Person travels");
    }

     public void study(String qualification){
        System.out.println(type +" studying " + qualification);
    }

    public int noOfLegs(){
        return 0;
    }
}

/* File name : InterfaceDemo.java */
public class InterfaceDemo{

    public static void main(String args[]){
        PersonClass m = new PersonClass();
        m.eat();
        m.travel();
        m.study("BESE");
    }
}
```

8. **Package**:
   - used to encapsulate a group of classes, interfaces, and sub-packages
   - It is a container of a group of related classes where some classes are made public and others are not exposed.
   - Refer Example:
     1. Package name: mypackage
     2. Class inside package: Food.java, Animal.java
     3. Classes which use the package: Vegetables.java , PackageDemo.java , Fruits.java

```java
/* File name : myPackage/Animal.java */
package myPackage;

public class Animal {
    
    public String eat(String food){
        return("Animal is eating "+ food);
    }
    
    public void makeSound(String sound){
        System.out.println("Animal is saying "+ sound);
    }
}
```

```java
import myPackage.*;

public class PackageDemo
{
    public static void main(String[] args) {
        
        System.out.println(" \nUsing <Animal> class in myPackage");
        Animal cat= new Animal();
        System.out.println(cat.eat("milk"));
        cat.makeSound("mew");
        
        
        System.out.println(" \n\nUsing <Food> class in myPackage\n");
        Foods obj = new Foods();
        obj.show();
    }
}
```

```java
/* File name : myPackage/Foods.java */
package myPackage;

import java.util.Scanner;

public class Foods {

    protected String foodName;
    protected int foodID;

    public void show()
    {
        System.out.println("Show method in Food Class");
    }

    public int getFoodID()
    {
        int id;
        Scanner b = new Scanner (System.in);
        System.out.print("Enter id: ");
        id = Integer.parseInt( b.nextLine() );
        return(id);
    }

    public String getFoodName()
    {
        String  name;
        Scanner b = new Scanner (System.in);
        System.out.print("Enter Food Name: ");
        name =  b.nextLine() ;
        return(name);

    }

    public void setFoodID(int fID, String fName){

        this.foodName = fName;
        this.foodID = fID;
        System.out.println( this.foodID  +": "+  this.foodName  );
    }
}
```

9. Provide an example program demonstrating the usage of the “final” keyword in Java.
10. **Reflection:**
    - Reflection is the ability of a program to analyze itself.
    - The **java.lang.reflect** package provides the ability to obtain the information about the fields, constructors, methods, and modifiers of a class.
    - **Demerits**:
        - Performance Overhead
        - Security Restrictions
        - Exposure of Internals

```java
import java.lang.reflect.*;
public class ReflectionDemo {
    public static void main(String args[])
    {
        try {
            Class c = Class.forName("java.util.Stack");
            Method methods[] = c.getDeclaredMethods();

            // dump all methods in the Stack class
            for (int i = 0; i < methods.length; i++)
                System.out.println(methods[i].toString());
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }
}
```