Day7 and 8: Inheritance, method Overriding, Object class, super keyword, Object downcasting, toString method.

Day 7

Inheritance in Java:

Inheritance is one of the most important features of OOP(Object-Oriented Programming).

Inheritance is a relationship[Parent-Child] between classes, it will bring variables and methods from one class[Super class / Base Class / Parent Class] to another class[Sub
class / Derived Class / Child Class] in order to reuse.

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 to your current class also.

Note: In Java, Inheritance is represented in the form of the "extends" keyword.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

example:

- **Car** is a **Vehicle**
- **Orange** is a **Fruit**
- **Surgeon** is a **Doctor**
- **Dog** is an **Animal**

The main advantage of Inheritance in Java is:

1. Code Reusability
2. To get Runtime polymorphism (using method overriding)

The new class that is created is known as **subclass** (child or derived class) and the existing class from where the child class is derived is known as **superclass** (parent or base class).

Interface is defined when you don’t have implemetation details.

A concrete class is defined when you have complete details about the object.

Abstract class is like a transition(transform) between interface and concrete class.

```
interface Animal{
abstract void move();
}
abstract class Fish implements Animal{
@Override
void move(){
System.out.print(”swimming fish”);
}
}
class Dolphin{
@Override
void move(){
System.out.print(”swimming and jump”);
}
}
```

Example1:

```
//parent class
class Animal {
  // methods and fields
}

//child class
class Dog extends Animal {

  // methods and fields of Animal is inherited
  // This dog class can have its own methods and fields as well
}
```

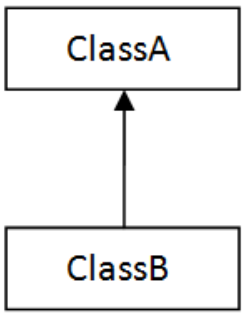
Example 2:

```
//parent class
class Animal {

  // field and method of the parent class
  String name;
  public void eat() {
    System.out.println("I can eat");
  }
}

// child class inherit from Animal
//Dog is an Animal
class Dog extends Animal {

  // new method in subclass
  public void display() {
    System.out.println("My name is " + name);
  }
}

class Main {
  public static void main(String[] args) {
```

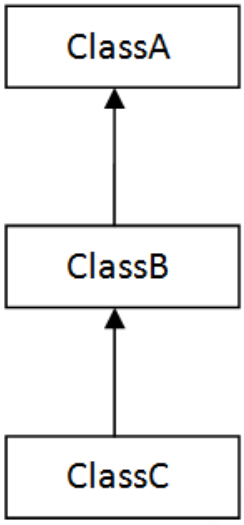
```
    // create an object of the subclass
    Dog labrador = new Dog();

    // access field of superclass
    labrador.name = "Rohu";
    labrador.display();

    // call method of superclass
    // using object of subclass
    labrador.eat();

  }
}
```

Types of Inheritance:

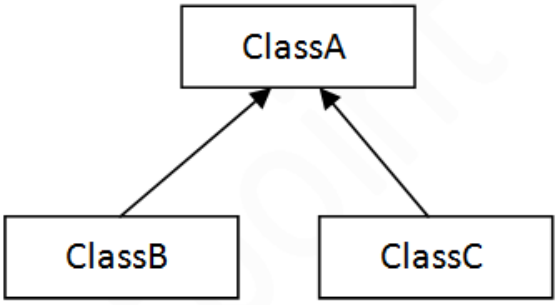
1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance



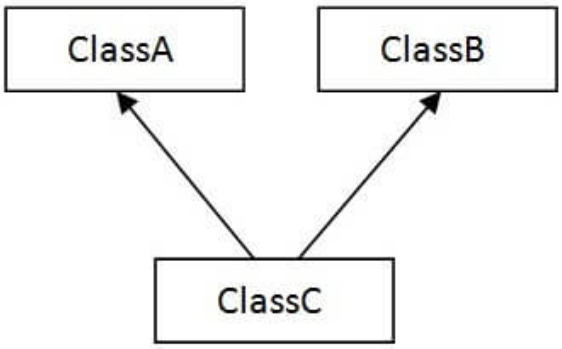
1) Single



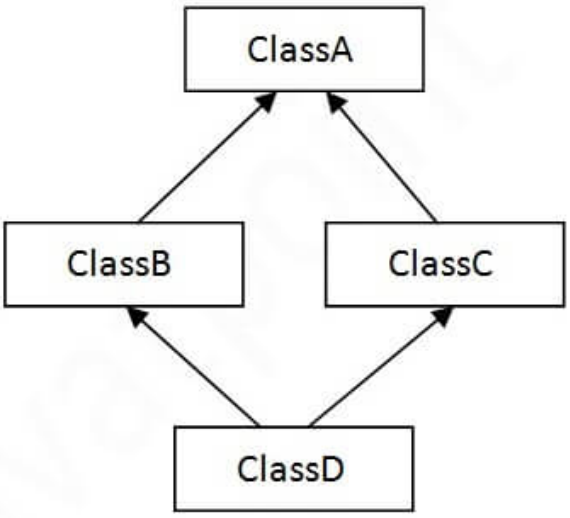
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

Note: The multiple inheritance and the hybrid inheritance have diamond problem meaning one class is deriving from multiple classes.

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

Note: Multiple inheritance is not supported in Java through class.

Single Inheritance:

When a class inherits another class, it is known as *single inheritance*. In the example given below, the Dog class inherits the Animal class, so there is a single inheritance.

```
class Animal{
    void eat(){
      System.out.println("eating...");
    }
}

class Dog extends Animal{
  void bark(){
    System.out.println("barking...");
  }
}

class Main{
  public static void main(String args[]){
    Dog d=new Dog();
    d.bark();
    d.eat();
  }
}
```

Multilevel Inheritance:

When there is a chain of inheritance, it is known as *multilevel inheritance.*
As you can see in the example given below, the BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
class Animal{
  void eat(){
    System.out.println("eating...");
  }
}

class Dog extends Animal{
  void bark(){
      System.out.println("barking...");
  }
}

class BabyDog extends Dog{
  void weep(){
      System.out.println("weeping...");
    }
}

class Main{
  public static void main(String args[]){
    BabyDog d=new BabyDog();
    d.weep();
    d.bark();
    d.eat();
  }
}
```

Hierarchical Inheritance:

When two or more classes inherit a single class, it is known as *hierarchical inheritance.*
In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

```
class Animal{
  void eat(){
    System.out.println("eating...");
  }
}

class Dog extends Animal{
  void bark(){
    System.out.println("barking...");
  }
}

class Cat extends Animal{
  void meow(){
      System.out.println("meowing...");
  }
}

class Main{
public static void main(String args[]){
  Cat c=new Cat();
  c.meow();
  c.eat();
  //c.bark();//C.T.Error
}}
```

Why multiple inheritance is not supported in java at class level ?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

Example:

```
class A{
  void msg(){
    System.out.println("Hello");
  }
}

class B{
  void msg(){
    System.out.println("Welcome");
  }
}
class C extends A,B{//suppose if it were, compilation error

 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
  }
}
```

Method Overriding in Java:

E.g. : TV Interface → B/W TV → Color TV

As we know that object of a child's class can access the method of its parent class also.

but, if the child class object does not satisfy with the implementation of the inherited method, the child class can re-implement the inherited method with his implementation, this concept is known as Method Overriding in java.

Usage of Java Method Overriding:

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class

2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

I Problem:

Creating a class Vehicle as a parent class that has a run method, lets create another class call Bike as child of this Vehicle class and override the run method from the parent class.

```
class Vehicle{
  //defining a method
  void run(){
    System.out.println("Vehicle is running");
  }
}
//Creating a child class
class Bike extends Vehicle{
  //defining the same method as in the parent class
  @Override
  void run(){
    System.out.println("Bike is running safely");
  }

  public static void main(String args[]){
    Bike obj = new Bike();//creating object
    obj.run();//calling method
  }
}
```

We Problem:

create a class Bank, with a method getRateOfInterest and create multiple child classes of This Bank class as SBI, ICICI and override the getRateOfInterest method in all the

child classes

```
class Bank{
    int getRateOfInterest(){
      return 0;
    }
}

//Creating child classes.
class SBI extends Bank{
    int getRateOfInterest(){
      return 8;
    }
}

class ICICI extends Bank{
    int getRateOfInterest(){
      return 7;
    }
}

class AXIS extends Bank{
  int getRateOfInterest(){
    return 9;
  }
}

//Main class to create objects and call the methods
class Main{
  public static void main(String args[]){

    SBI s=new SBI();
    ICICI i=new ICICI();
    AXIS a=new AXIS();

    System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
    System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
    System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
  }
}
```

Note: We can not override a static method :

We should use @Override annotation to validate a valid method overriding at compile time.

super keyword in Java:

The **super** keyword in Java is a reference variable that is used to refer to immediate parent class object.

Whenever you create the instance of a subclass, an instance of parent class is created implicitly which is referred by the super reference variable.

Usage of Java super Keyword

1. super can be used to refer to the immediate parent class instance variable.
2. super can be used to invoke the immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

Example1: referring to the immediate parent class instance variable:

```
class Animal{
  String color="white";
}
class Dog extends Animal{

  String color="black";

  void printColor(){
    System.out.println(color);//prints color of Dog class
    System.out.println(super.color);//prints color of Animal class
  }
}
class Main{
public static void main(String args[]){
  Dog d=new Dog();
  d.printColor();
```

```
  }
}
```

Example2: referring to the immediate parent class instance method:

```
class Animal{
  void eat(){
      System.out.println("eating...");
  }
}
class Dog extends Animal{
  void eat(){
    System.out.println("eating bread...");
  }
  void bark(){
    System.out.println("barking...");
  }
  void work(){
    super.eat();
    bark();
  }
}
class Main{
  public static void main(String args[]){
    Dog d=new Dog();
    d.work();
  }
}
```

Example3: invoking the parent class constructor

The first line in each constructor is call to the super constructor `super();`

But it is there by defalut, meaning you don't have to write it.

```
class Animal{
  Animal(){
    System.out.println("animal is created");
}
}

class Dog extends Animal{
  Dog(){
    super();
```

```
    System.out.println("dog is created");
  }
}

class Main{
  public static void main(String args[]){
  Dog d=new Dog();
  }
}
```

Note: As we know that default constructor is provided by the compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Example:

```
class Animal{
  Animal(){
    System.out.println("animal is created");}
  }

class Dog extends Animal{
  Dog(){
    System.out.println("dog is created");
  }
}
class Main{
  public static void main(String args[]){
  Dog d=new Dog();
  }
}

Output:
animal is created
dog is created
```

We Problem: [skipping]

```
class Person{

  int id;
  String name;

  Person(int id,String name){
    this.id=id;
    this.name=name;
  }
}

class Employee extends Person{

  float salary;
  Employee(int id,String name,float salary){
    super(id,name);//reusing parent constructor
    this.salary=salary;
  }

  void display(){
    System.out.println(id+" "+name+" "+salary);}
  }

class Main{
  public static void main(String[] args){

  Employee e1=new Employee(1,"ankit",45000f);
  e1.display();
  }
}

output:
1 ankit 45000
```

Day 8

Dynamic or Runtime polymorphism:

It is a process in which a call to an overridden method is resolved at runtime rather than

compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Upcasting:

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:

```
class A{ //Parent class
  --
}
class B extends A{  //Child class
  --
}


A a=new B();//upcasting, this is only possible if B class is a child class of A
```

Example:

```
class Bike{
  void run(){
    System.out.println("running");
  }
}
class Splendor extends Bike{

  void run(){
    System.out.println("running safely with 60km");
  }

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run(); //
  }
}
```

```
output:
running safely with 60km.
```

Explanation:

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

You Problem:

create a class Bank with following instance method:

```
double getRateOfInterest(){
    return 0;
}
```

create 3 different child classes SBI, ICICI, AXIS and override the getRateOfInterest() method with different rates of interest.

create a Main class with the main method.

create 3 Bank class (super class) reference and assign the above 3 child classes object to these 3 different reference variable .

call the getRateOfInterest method on these 3 Bank class reference variable.

Note: **Runtime polymorphism can't be achieved by data members.**

Example:

```
class Bike{
 int speedlimit=90;
}
class Honda3 extends Bike{
 int speedlimit=150;

 public static void main(String args[]){
  Bike obj=new Honda3();
  System.out.println(obj.speedlimit);//90
}
```

Java RunTime Polymorphism with Multilevel Inheritance:

```
class Animal{
  void eat(){
    System.out.println("eating");
  }
}

class Dog extends Animal{
  void eat(){
    System.out.println("eating fruits");
  }
}

class BabyDog extends Dog{
  void eat(){
    System.out.println("drinking milk");
  }

public static void main(String args[]){

Animal a1,a2,a3;

    a1=new Animal();
    a2=new Dog();
    a3=new BabyDog();

    a1.eat();
    a2.eat();
    a3.eat();
  }
```

```
}

output:
eating
eating fruits
drinking Milk
```

Object Downcasting and instanceof operator:

instanceof operator:

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or an interface).

The instanceof operator in java is also known as a type *comparison operator* because it compares the instance with a reference type. It returns either true or false. If we apply the instanceof operator with any reference variable that has a null value, it returns false.

Example1:

```
class Animal{
 public static void main(String args[]){
 Animal a=new Animal();
 System.out.println(a instanceof Animal);//true
 }
}
```

Note: An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

Example2:

```
class Animal{
  --
}
class Dog extends Animal{//Dog inherits Animal

 public static void main(String args[]){
 Dog d=new Dog();
 System.out.println(d instanceof Dog);//true
 System.out.println(d instanceof Animal);//true
 }
}
```

Object Downcasting:

As we know that to a parent class variable we can assign the child class object also, and from that parent class variable if we try to call any overriden method then due to Runtime polymorphism the overriden method will be called. but if a parent class reference points to a child class object, with that parent class refernce we can not call the child class specific methods, which are not available inside the parent class.

to call the child class specific method from the parent class reference variable we need to downcast the parent class variable to the appropriate child class object.

Example:

```
class Animal{

  void eat(){
      System.out.println("eating...");
  }
}
class Dog extends Animal{

  @Override
  void eat(){
    System.out.println("eating bread...");
  }
  //specific method of child class
  void bark(){
    System.out.println("barking...");
```

```
  }

}
class Main{
  public static void main(String args[]){

    Animal parent = new Dog();
    parent.eat(); //eating bread...

    //calling child class specific method with parent class variable
    parent.bark(); //C T Error

    //downcasting parent class variable to the child class object.
//downcasting works fine at compile time
    Dog d = (Dog) parent;

    d.bark();

  }
}
```

Note: We can downcast the parent class variable to the child class object only if the Parent class variable points to the Child class object , otherwise it will throw a runtime exception called *ClassCastException*.

Example:

```
class Animal{

  void eat(){
      System.out.println("eating...");
  }
}
class Dog extends Animal{

  @Override
  void eat(){
    System.out.println("eating bread...");
  }
  //specific method of child class
  void bark(){
    System.out.println("barking...");
```

```
  }

}
class Main{

    void doSomething(Animal a){

        a.eat();

        if(a instanceof Dog){

            Dog d = (Dog)a;
            d.bark();
        }

    }

  public static void main(String args[]){

        Main main = new Main();
        main.doSomething(new Animal());
        main.doSomething(new Dog());


  }
}
```

final keyword in Java:

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

If you make any variable final, you cannot change the value of a final variable(It will be constant).

In Java, the final variable must be initialized before we use it, either at the time of declaration or inside the constructor of the class.

If you make any method final, you cannot override it inside the child class.

If you make any class as final, you cannot extend it. final class does not have the child class.

Object class In Java:

The **Object** class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a class does not extend any other class then it is a direct child class of the **Object** and if extends another class then it is indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

The Object class is beneficial if you want to refer to any object whose type you don't know. Notice that the parent class reference variable can refer to the child class object, known as upcasting.

Let's take an example, there is the getObject() method that returns an object but it can be of any type like Employee, Student etc, we can use Object class reference to refer to that object. For example:

```
Object obj=getObject();//we don't know what object will be returned from this method
```

The Object class provides some common behaviors to all the objects such as the object can be compared, the object can be cloned, the object can be notified etc.

Methods of Object class:

The Object class provides many methods. these methods are inherited in all the objects in java.

1. **public final Class getClass()** : returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
2. **public int hashCode():** returns the hashcode number for this object.
3. **public boolean equals(Object obj)** : compares the given object to this object.
4. **protected Object clone() throws CloneNotSupportedException** : creates and returns the exact copy (clone) of this object.
5. **public String toString()** : returns the string representation of this object.

To be discussed after multithreading.

1. **public final void notify()** : wakes up single thread, waiting on this object's monitor.
2. **public final void notifyAll()** : wakes up all the threads, waiting on this object's monitor.
3. **public final void wait(long timeout)throws InterruptedException** : causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
4. **public final void wait(long timeout,int nanos)throws InterruptedException:** causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
5. **public final void wait()throws InterruptedException**: causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
6. **protected void finalize()throws Throwable:** is invoked by the garbage collector before object is being garbage collected.

We will discuss about these methods in upcoming classes.

public String toString() method:

The toString() provides a String representation of an object and is used to convert an object to String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, it is defined as:

```
// Default behavior of toString() is to print class name, then
// @, then unsigned hexadecimal representation of the hash code
// of the object

public String toString()
{
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Note: *Whenever we try to print any Object reference, then internally toString() method is called.*

It is always recommended to override the **toString()** method to get our own String representation of Object. For more on override of toString() method.

```
class Student{
 int rollno;
 String name;
 String city;

 Student(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }

 public static void main(String args[]){
   Student s1=new Student(101,"Raj","lucknow");
```

```
   Student s2=new Student(102,"Vijay","ghaziabad");

   System.out.println(s1);//println method call s1.toString()
   System.out.println(s2);//println method call s2.toString()
 }
}

output:
Student@1fee6fc
Student@1eed786
```

Lets override the toString() method from the Object class to our Student class.

```
class Student{
 int rollno;
 String name;
 String city;

 Student(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }

@Override
 public String toString(){//overriding the toString() method
  return rollno+" "+name+" "+city;
 }
 public static void main(String args[]){
   Student s1=new Student(101,"Raj","lucknow");
   Student s2=new Student(102,"Vijay","ghaziabad");

   System.out.println(s1);
   System.out.println(s2);
 }
}

output:
101 Raj lucknow
102 Vijay ghaziabad
```

Reference :

https://www.javatpoint.com/inheritance-in-java

https://www.edureka.co/blog/upcasting-and-downcasting-in-java/