DATE:18/09/2024 2022503045

ASSIGNMENT – 6

1.Single Inheritance

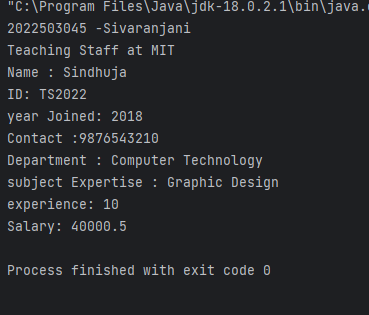
Defintion: Single inheritance is the simplest type of inheritance in java. In this, a class inherits the properties from a single class. The class which inherits is called the derived class or child class or subclass, while the class from which the derived class inherits is called the base class or superclass or parent class.

For example, we can take Staffs as Super class here and teachingStaff as a subclass.The common attributes like name,Id ,contact details are stored in one class.And specific attributes of teachingStaff are included in the subclass.And then we inherit the properties and attributes from the Staffs super class

CODE:

class Staffs{  
 String name;  
 String ID;  
 String contact\_info;  
 int year\_joined;  
 public void print(){  
 System.*out*.println("Staff at MIT");  
 }  
 public void Details(String name,String ID,String contact\_info,int year\_joined){  
 this.name=name;  
 this.ID=ID;  
 this.contact\_info=contact\_info;  
 this.year\_joined=year\_joined;  
 }  
 public void displayDetails(){  
 System.*out*.println("Name : "+name+"\nID: "+ID+"\nyear Joined: "+year\_joined+"\nContact :"+contact\_info);  
 }  
}  
class TeachingStaff extends Staffs{  
 @Override  
 public void print(){  
 System.*out*.println("Teaching Staff at MIT");  
 }  
 String Department;  
 String subjectExpertise;  
 int experience;  
 double MonthlySalary;  
 public void information(String Department,String subjectExpertise,int experience,double MonthlySalary){  
 this.Department=Department;  
 this.subjectExpertise=subjectExpertise;  
 this.experience=experience;  
 this.MonthlySalary=MonthlySalary;  
 }  
 public void displayinfo(){  
 System.*out*.println("Department : "+Department+"\nsubject Expertise : "+subjectExpertise+"\nexperience: "+experience+"\nSalary: "+MonthlySalary);  
 }  
}  
public class singleInheritance\_3045 {  
 public static void main(String[] args){  
 System.*out*.println("2022503045 -Sivaranjani");  
 TeachingStaff teachingStaff=new TeachingStaff();  
 teachingStaff.print();  
 teachingStaff.Details("Sindhuja","TS2022","9876543210",2018);  
 teachingStaff.information("Computer Technology","Graphic Design",10,40000.50);  
 teachingStaff.displayDetails();  
 teachingStaff.displayinfo();  
 }  
}

OUTPUT :



2.Multilevel Inheritance

Definition : In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class .In Java, a class cannot directly access the grandparent’s members.

For example,we can add another derived class for teachingStaff as lecturer which can get the properties and other attributes from teaching Staff as well as staffs. This is multilevel inheritance.

CODE:

class lecturer extends TeachingStaff{

/\*contractType (e.g., Full-time, Part-time)

teachingLoad (number of courses per semester)

areasOfExpertise (array of subjects)

studentFeedback (average rating)\*/

String contractType;

int noOfCourses\_assigned;

double MonthlySalary;

int workingHoursPerWeek;

@Override

public void print(){

System.out.println("Teaching Staff -Lecturer");

}

public void addInfo(String contractType,int noOfCourses\_assigned,double MonthlySalary,int workingHours){

this.contractType=contractType;

this.noOfCourses\_assigned=noOfCourses\_assigned;

this.MonthlySalary=MonthlySalary;

workingHoursPerWeek=workingHours;

}

public void moreInfo(){

System.out.println("contract Type: "+contractType+"\nno Of Courses Assigned : "+noOfCourses\_assigned+"\nMonthly Salary : "+MonthlySalary+"Working hours per week : "+workingHoursPerWeek);

}

}

public class MultilevelInheritance\_3045 {

public static void main(String[] args){

System.out.println("Sivaranjani - 2022503045");

lecturer l=new lecturer();

l.print();

l.Details("Neela","TS2004L","9875432124",2020);

l.addInfo("Full-time", 3, 35000.3, 20);

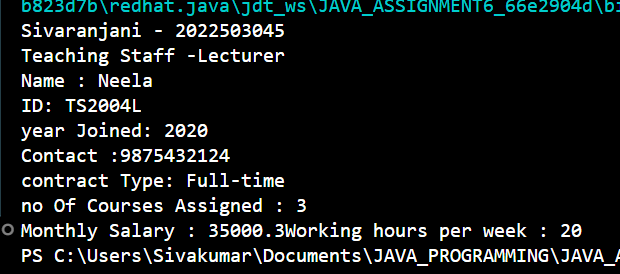
l.displayDetails();

l.moreInfo();

}

}

OUTPUT:



3. Hierarchical Inheritance

DEFINITION : In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.

Like previous we had only only teachingStaff now we will have many subclasses that derive the Staff class like administrativeStaff,Technical Staff that inherit the Staffs superclass.This is called hierarchical Inheritance.

CODE:

class AdministrativeStaffs extends Staffs{

String role;

String[] skills;

String adminID;

@Override

public void print(){

System.out.println("Administrative Staff at MIT");

}

public void adminDetails(String role,String[] skills,String adminID){

this.role=role;

this.skills=skills;

this.adminID=adminID;

}

public void displayAdminDetails(){

System.out.print("Role :"+role+"\n skills : ");

for(String skill:skills){

System.out.print(skill+",");

}

System.out.print("\n AdminID :"+adminID+"\n");

}

}

class TechnicalStaffs extends Staffs{

int[] lab\_nos\_incharged;

String shiftTiming;

String department;

@Override

public void print(){

System.out.println("Technical Staff at MIT");

}

public void techDetails(String shiftTiming,int[] lab\_nos\_incharged,String department){

this.shiftTiming=shiftTiming;

this.lab\_nos\_incharged=lab\_nos\_incharged;

this.department=department;

}

public void techStaffDetails(){

System.out.print("ShiftTiming :"+shiftTiming+"\n Labs Incharge : ");

for(int lab:lab\_nos\_incharged){

System.out.print("lab"+lab+",");

}

System.out.print("\n department:"+department+"\n");

}

}

public class HierarchicalInherit\_3045 {

public static void main(String[] args){

System.out.println("Sivaranjani -2022503045");

AdministrativeStaffs admin=new AdministrativeStaffs();

admin.print();

admin.Details("gokul","TS2762A","9857286611", 2022);

admin.adminDetails("Office Manager",new String[]{"Communication","MS EXCEL","MSWORD"},"174@ADMIN");

admin.displayDetails();

admin.displayAdminDetails();

System.out.println();

System.out.println();

TechnicalStaffs techStaff=new TechnicalStaffs();

techStaff.print();

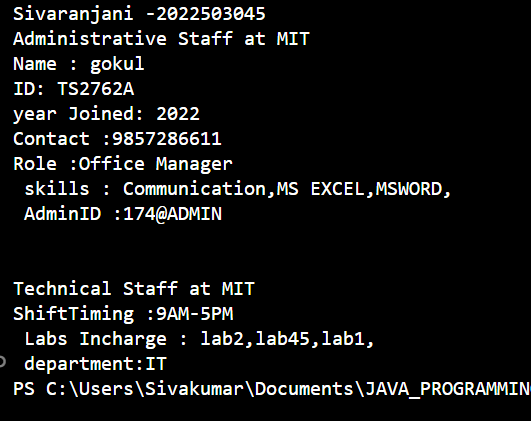
techStaff.techDetails("9AM-5PM", new int[]{2,45,1}, "IT");

techStaff.techStaffDetails();

}

}

OUTPUT:



4. Multiple Inheritance (class? Vs Interface)

Definition: Multiple Inheritance is a feature of an object-oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with the same signature in both the superclasses and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

For example, In our case we can consider a subclass SupportingStaffs who extends both technical Staff and also administrative staff. Now

CODE:

class SupportingStaffs extends TechnicalStaffs,AdministrativeStaffs{

public void display(){

System.out.println("Supporting Staff in administrative Office for technical Issue ");

}

}

public class MultipleInherit\_3045 {

public static void main(String[] args){

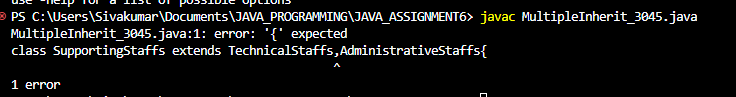
SupportingStaffs Sf=new SupportingStaffs();

sf.display();

}

}

OUTPUT :

 From this we can infer that java doesn’t support multiple inheritance in case of classes we can’t implement multiple inheritance.So for this we have to use Interfase.

Java 8 supports default methods where interfaces can provide a default implementation of methods. And a class can implement two or more interfaces. In case both the implemented interfaces contain default methods with the same method signature, the implementing class should explicitly specify which default method is to be used in some method excluding the main() of implementing class using super keyword, or it should override the default method in the implementing class, or it should specify which default method is to be used in the default overridden method of the implementing class.

CODE with interface:

interface techStaff{

default void display(){

System.out.println("Technical Staff at Mit");

}

}

interface adminStaff{

default void print(){

System.out.println("Administrative Staff at MIT");

}

}

class SupportingStaffs implements techStaff,adminStaff{

public void joinDisplay(){

techStaff.super.display();

adminStaff.super.print();

}

}

public class MultipleInherit\_3045 {

public static void main(String[] args){

SupportingStaffs Sf=new SupportingStaffs();

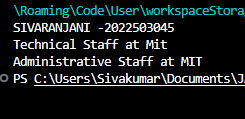
System.out.println("SIVARANJANI -2022503045");

Sf.joinDisplay();

}

}

OUTPUT:



5. Polymorphism

Definition: Polymorphism is that in which we can perform a task in multiple forms or ways. It is applied to the functions or methods. Polymorphism allows the object to decide which form of the function to implement at compile-time as well as run-time.

Types of Polymorphism are:

1.Compile-time polymorphism (Method overloading)

2.Run-time polymorphism (Method Overriding)

For example we can have salary for the staffs as two methods one with bonus and other with bonus and allowance for specific staffs alone we can have two functionalities.

CODE:

class SalaryforStaff{

double total\_salary;

public double Salary(double base\_salary,double bonus\_amount){

total\_salary=base\_salary+bonus\_amount;

return total\_salary;

}

public double Salary(double base\_salary,double bonus\_amount,double allowance ){

return base\_salary+bonus\_amount-allowance;

}

public void print(){

System.out.println("calculating Salary for the Staff");

}

}

public class Polymorphism\_3045 extends SalaryforStaff{

public void print(){

System.out.println("salary calculated using polymorphism");

}

public static void main(String[] args){

System.out.println("Sivaranjani -2022503045");

SalaryforStaff s=new SalaryforStaff();

//overloading at compile-time

System.out.println("Salary without allowance :"+s.Salary(10000.38,5000.5));

System.out.println("Salary with allowance :"+s.Salary(10000.38, 50000.5,1000));

//overriding

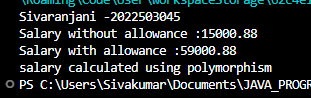
Polymorphism\_3045 obj=new Polymorphism\_3045();

obj.print();

}

}

OUTPUT:



6.Super usage in Inheritance

Super keyword is used for calling the parent method while overriding the method in the subclass and also if two interface has same method name we can use super for calling the specific method.

Example in our previous code where we had display function for each subclass we can call the display function of the super class now without invoking in the main class.

CODE:

class lecturer extends TeachingStaff{

String contractType;

int noOfCourses\_assigned;

double MonthlySalary;

int workingHoursPerWeek;

@Override

public void print(){

System.out.println("Teaching Staff -Lecturer");

}

public void addInfo(String contractType,int noOfCourses\_assigned,double MonthlySalary,int workingHours){

this.contractType=contractType;

this.noOfCourses\_assigned=noOfCourses\_assigned;

this.MonthlySalary=MonthlySalary;

workingHoursPerWeek=workingHours;

}

public void displayDetails(){

super.displayDetails();

System.out.println("contract Type: "+contractType+"\nno Of Courses Assigned : "+noOfCourses\_assigned+"\nMonthly Salary : "+MonthlySalary+"Working hours per week : "+workingHoursPerWeek);

}

}

public class MultilevelInheritance\_3045 {

public static void main(String[] args){

System.out.println("Sivaranjani - 2022503045");

lecturer l=new lecturer();

l.print();

l.Details("Neela","TS2004L","9875432124",2020);

l.addInfo("Full-time", 3, 35000.3, 20);

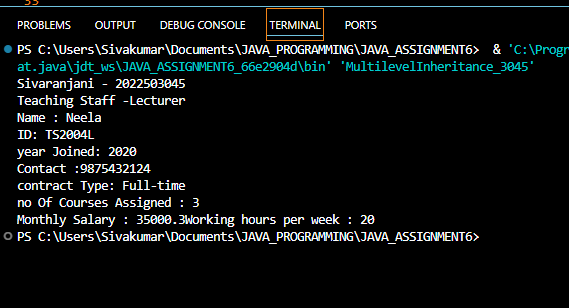
l.displayDetails();

}

}

Now the output would be same :

OUTPUT:



7.Construtor in inheritance

Constructors in Java are used to initialize the values of the attributes of the object serving the goal to bring Java closer to the real world. We already have a default constructor that is called automatically if no constructor is found in the code. But if we make any constructor say parameterized constructor in order to initialize some attributes then it must write down the default constructor because it now will be no more automatically called.

Example for our case we can consider creating a librarystaffs class with constructor that have parameter as well as empty constructor.

CODE:

class LibraryStaffs extends Staffs{

int no\_of\_staffs;

String last\_updated\_date;

LibraryStaffs(){

System.out.println("Library Staffs class is created empty");

}

LibraryStaffs(int no\_of\_staffs,String last\_updated\_date){

this.no\_of\_staffs=no\_of\_staffs;

this.last\_updated\_date=last\_updated\_date;

System.out.println("No of Staffs :"+no\_of\_staffs+"\t --- last Updated :"+last\_updated\_date +"\n Library created with parameters");

}

}

public class Construtor\_3045 {

public static void main(String[] args){

System.out.println("Sivaranjani - 2022503045");

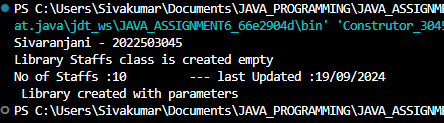
LibraryStaffs lb1=new LibraryStaffs();

LibraryStaffs lb2=new LibraryStaffs(10,"19/09/2024");

}

}

Output:



8.shallow copy vs Deep copy

|  |  |
| --- | --- |
| Shallow Copy stores the references of objects to the original memory address. | Deep copy stores copies of the object’s value. |
| Shallow Copy reflects changes made to the new/copied object in the original object. | Deep copy doesn’t reflect changes made to the new/copied object in the original object. |
| Shallow Copy stores the copy of the original object and points the references to the objects. | Deep copy stores the copy of the original object and recursively copies the objects as well. |
| A shallow copy is faster. | Deep copy is comparatively slower. |

In our example we can take lecturer while shallow copy is modified we can notice there will be modification in the original as well whereas the deep copy modification doesn’t result in any changes in original.

CODE:

public class ShallowDeepCopy\_3045 {

public static void main(String[] args){

System.out.println("Sivaranjani -2022503045");

lecturer l\_original=new lecturer();

l\_original.addInfo("full-time", 10,90000,11);

//modification in deepcopy doesn't effect original

lecturer l\_deepcopy=new lecturer();

l\_deepcopy.addInfo(l\_original.contractType, l\_original.noOfCourses\_assigned, l\_original.MonthlySalary,l\_original.workingHoursPerWeek);

l\_deepcopy.contractType="Part-time";

System.out.println("DEEP COPY after modification ");

l\_deepcopy.displayDetails();;

System.out.println("Original : ");

l\_original.displayDetails();;

System.out.println();

lecturer l\_shallowcopy=l\_original;

l\_shallowcopy.MonthlySalary=70000.65;

System.out.println("SHALLOW COPY :");

l\_shallowcopy.displayDetails();;

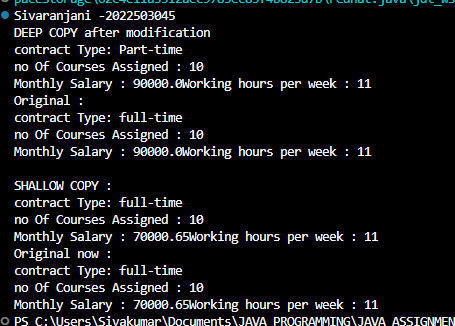
System.out.println("Original now : ");

l\_original.displayDetails();;

}

}

Output :



9.Shallow Cloning vs Deep Cloning

Deep cloning

Creates a new object with the same structure and values as the original object, but without any reference to the original. Deep cloning is useful when working with complex data structures, and it ensures that changes to one copy don't affect the others. However, deep cloning can be slower and consume more memory, especially for large data structures.

Shallow cloning

Creates a clone of a repository with a truncated history of the commits. Shallow cloning is a feature of the version control system Git, and it allows you to get only the most recent commits instead of the entire repository history. Shallow cloning saves bandwidth and disk space, and it makes it easier to search the recent Git history and troubleshoot problems.

CODE :

import java.util.ArrayList;

import java.util.Arrays;

class skills implements Cloneable{

ArrayList<String> techskills;

ArrayList<String> non\_techSkills;

public skills(ArrayList<String> techskills,ArrayList<String> non\_techSkills){

this.techskills=techskills;

this.non\_techSkills=non\_techSkills;

}

@Override

public Object clone() throws CloneNotSupportedException{

return super.clone();

}

}

class SkillSetOfTechStaff implements Cloneable{

String staffID;

skills skills;

public void displaySkills(){

System.out.println("Staff ID : "+staffID+"\nSkills :"+skills.techskills+"\t"+skills.non\_techSkills);

}

@Override

protected Object clone() throws CloneNotSupportedException{

return super.clone();

}

}

public class ShallowDeepClone\_3045 {

public static void main(String[] args) throws CloneNotSupportedException{

SkillSetOfTechStaff OriginalStaff =new SkillSetOfTechStaff();

OriginalStaff.staffID="TS00374";

OriginalStaff.skills=new skills(new ArrayList<>(Arrays.asList("Operating System","DBMS","DSA","JAVA","C++")),new ArrayList<>(Arrays.asList("Communication","Critical Thinking")));

SkillSetOfTechStaff ShallowCopyStaff=(SkillSetOfTechStaff)OriginalStaff.clone();

System.out.println("Original TechStaff : ");

OriginalStaff.displaySkills();

System.out.println();

System.out.println("Shallow Copied : ");

ShallowCopyStaff.displaySkills();

System.out.println("Equality of original staff and shallow copied staff : "+(OriginalStaff==ShallowCopyStaff));

System.out.println("Equality of skillset object in shallow and original : "+(OriginalStaff.skills==ShallowCopyStaff.skills));

System.out.println("-----------------------------------------------------------------------------------------------");

SkillSetOfTechStaff DeepCopyStaff=(SkillSetOfTechStaff)OriginalStaff.clone();

DeepCopyStaff.skills=(skills)OriginalStaff.skills.clone();

System.out.println("Original TechStaff : ");

OriginalStaff.displaySkills();

System.out.println();

System.out.println("Deep Copied : ");

ShallowCopyStaff.displaySkills();

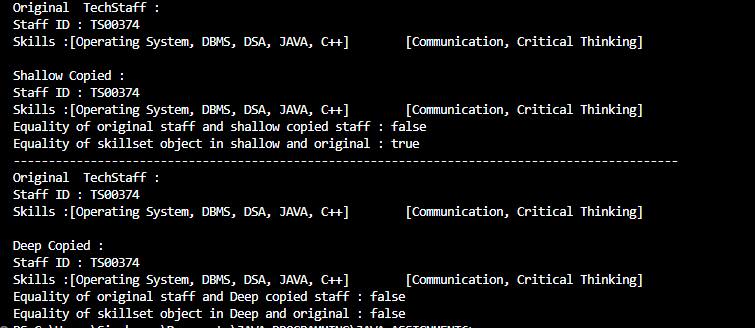
System.out.println("Equality of original staff and Deep copied staff : "+(OriginalStaff==DeepCopyStaff));

System.out.println("Equality of skillset object in Deep and original : "+(OriginalStaff.skills==DeepCopyStaff.skills));

}

}

OUTPUT:



10.Finalize method

The Java finalize() method of Object class is a method that the Garbage Collector always calls just before the deletion/destroying the object which is eligible for Garbage Collection to perform clean-up activity. Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection, or we can say resource de-allocation. Remember, it is not a reserved keyword. Once the finalize() method completes immediately, Garbage Collector destroys that object.

CODE:

public class Finalize\_3045 {

protected void finalize() throws Throwable{

try{

System.out.println("Finalizing in main class");

}

catch(Throwable e){

throw e;

}

finally{

super.finalize();

}

}

public static void main(String[] args){

System.out.println("Sivaranjani - 2022503045");

Finalize\_3045 obj=new Finalize\_3045();

try{

obj.finalize();

}

catch(Throwable e){

System.out.println(e);

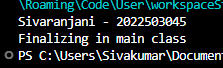
}

System.gc();

}

}

Output:



11.Packages

• Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

• Without package: The name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions.

• Need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. • The package is both a naming and a visibility control mechanism.

• You can define classes inside a package that are not accessible by code outside that package

CODE:

MYPACKAGE CONTAINS THESE CLASSES:

package mypackage;

public class professor extends mypackage.staff {

public void print(){

System.out.println("This is professor from mypackage");

super.display();

}

}

package mypackage;

public class staff {

public staff(){

System.out.println("This is from staff class of mypackage");

}

public void display(){

System.out.println("display method of staff");

}

}

IMPORTING THE PACKAGE

import mypackage.\*;

public class packagetesting\_3045 {

public static void main(String[] args){

professor prof=new professor();

System.out.println("Sivaranjani - 2022503045");

prof.print();

staff st=new staff();

}

}

OUTPUT:

