Code 1:

The code contains a compilation error because the constructor call xyz x1 = new xyz(10,20); is ambiguous.

Java cannot determine which constructor to use because both constructors

can match the argument types when implicit type conversion is considered.

class xyz{

int a;

float b;

public xyz(int a, float b){

this.a=a;

this.b=b;

}

}

public xyz(float b, int a){

this.b=b;

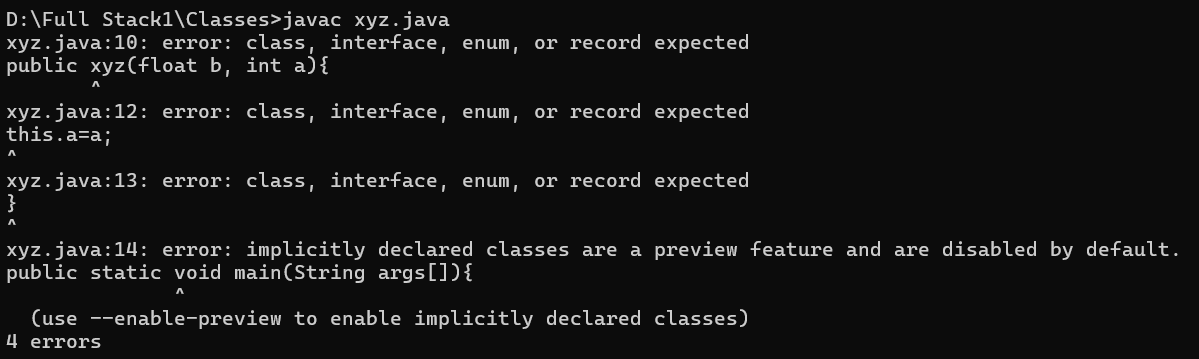
this.a=a;

}

public static void main(String args[]){

xyz x1=new xyz(10,20);

}



Code 2: Parent and Child Constructor

This code demonstrates inheritance with parent and child constructors,

where the child class explicitly calls the parent's constructor using super() to initialize inherited properties.

class parent {

int id;

String name;

public parent(int id, String name) {

this.id = id;

this.name = name; }

}

class child extends parent {

// there is cons in parent class so you have create a cons here also as it is inherited by parent class

int grade;

public child(int id, String name, int grade) {

super(id, name);

this.grade = grade; }

public void display() {

System.out.println("Child created: " + name + ", Grade: " + grade); }

}

public class ParentChildCons {

public static void main(String[] args) {

child c1 = new child(10, "Anjali", 7);

child c2 = new child(10, "Issar", 8);

c1.display();

c2.display();

}

}

**A black screen with white text

Description automatically generated**

Code 3: Another way of printing the output of same code

class parent {

int id;

String name;

public parent(int id, String name) {

this.id = id;

this.name = name;

}

}

class child extends parent {

// there is constructor in parent class so you have create a constructor here also as it is inherited by parent class

int grade;

public child(int id, String name, int grade) {

super(id, name);

this.grade = grade;

}

}

class ParentChildCons1{

public static void main(String[] args) {

child c1 = new child(10, "Anjali", 7);

child c2 = new child(10, "Issar", 8);

System.out.println("Child created: " + c1.name + c1.id + c1.grade);

System.out.println("Child created: " + c2.name + c2.id + c2.grade);

}

}

A black screen with white text

Description automatically generated

Code 4: Example of Single Inheritance

This code illustrates single inheritance, where the circle class inherits

from the shape class and calculates the area of a circle using inherited and additional properties.

class shape{

int radius;

float pi=3.14f;

int side;

public void area(int radius, float pi){

System.out.println("Calculate the area of circle: ");

}

}

class circle extends shape{

float area;

public void calculateArea(int radius, float pi){

area=pi\*radius\*radius;

System.out.println("Area of circle is "+area);

}

}

class singleInheritance{

public static void main(String args[]){

circle c=new circle();

c.calculateArea(10,3.14f);

}

}

A black background with white text

Description automatically generated

Code 5:Another Example of Single Inheritance

This code demonstrates single inheritance, where Programmer and Manager classes inherit

from the employee class to calculate their respective salaries using additional benefits.

class employee{

int salary=10000;

}

class Programmer extends employee{

int inc1=1000;

int inc2=2000;

int pf=500;

public void calculateSalary(){

int prosalary=salary+inc1+inc2-pf;

System.out.println("Programmer salary is: "+prosalary);

}

}

class Manager extends employee{

int inc3=3000;

int inc4=5000;

public void calculateSalaryOfman(){

int mansalary=salary+inc3+inc4;

System.out.println("Manager salary is: "+mansalary);

}

}

class SingleInheritance2{

public static void main(String args[]){

Manager m=new Manager();

Programmer p=new Programmer();

p.calculateSalary();

m.calculateSalaryOfman();

}

}

A black screen with white text

Description automatically generated

Code 6: Hierarchical Inheritance

This code showcases hierarchical inheritance, where both Programmer and Manager classes inherit

from the employee class and compute their respective salaries.

class employee{

int salary=10000;

public void emsalary(){

System.out.println("Employee salary is: "+salary);

}

}

class Programmer extends employee{

int inc1=1000;

int inc2=2000;

int pf=500;

public void calculateSalary(){

super.emsalary();

int prosalary=salary+inc1+inc2-pf;

System.out.println("Programmer salary is: "+prosalary);

}

}

class Manager extends employee {

int inc3=3000;

int inc4=5000;

public void calculateSalaryOfman(){

super.emsalary();

int mansalary=salary+inc3+inc4;

System.out.println("Manager salary is: "+mansalary);

}

}

class HierarchicalInheritance{

public static void main(String args[]){

Manager m=new Manager();

Programmer p=new Programmer();

p.calculateSalary();

m.calculateSalaryOfman();

}

}

A black screen with white text

Description automatically generated

Code 7: MultiLevelInheritance

This code demonstrates multilevel inheritance, where Manager inherits from Programmer,

which in turn inherits from Employee, enabling layered salary calculations.

class Employee {

int salary = 10000;

public void emsalary() {

System.out.println("Employee salary is: " + salary);

}

}

class Programmer extends Employee {

int inc1 = 1000;

int inc2 = 2000;

int pf = 500;

public void calculateSalary() {

super.emsalary();

int prosalary = salary + inc1 + inc2 - pf;

System.out.println("Programmer salary is: " + prosalary);

}

}

class Manager extends Programmer {

int inc3 = 3000;

int inc4 = 5000;

public void calculateSalaryOfman() {

super.emsalary();

int mansalary = salary + inc1 + inc2 + inc3 + inc4 - pf;

System.out.println("Manager salary (including Programmer's benefits) is: " + mansalary);

}

}

public class MultiInheritance1 {

public static void main(String[] args) {

Manager m = new Manager();

Programmer p = new Programmer();

p.calculateSalary();

m.calculateSalaryOfman();

}

}

A black screen with white text

Description automatically generated

Code 8:

class bike{

int speed;

int gear;

public bike(int gear, int speed){

this.gear=gear;

this.speed=speed;

}

public void getspeed(){

System.out.println("The speed of bike is"+speed+"km/hr");

}

public void getgear(){

System.out.println("The gear of bike is"+gear);

}

}

class splender extends bike{

public splender(int gear, int speed){

super(gear,speed);

}

public void getspeed(){

System.out.println("The speed of splender is"+speed+"km/hr");

}

public void getgear(){

System.out.println("The gear of splender is"+gear);

}

}

class boxer extends bike{

boxer(int gear, int speed){

super(gear,speed);

}

public void getspeed(){

System.out.println("The speed of boxer is"+speed+"km/hr");

}

public void getgear(){

System.out.println("The gear of boxer is"+gear);

}

}

class HierarchicalInheritance2{

public static void main(String args[]){

bike b=new bike(4,80);

b.getspeed();

b.getgear();

splender s=new splender(3,100);

s.getspeed();

s.getgear();

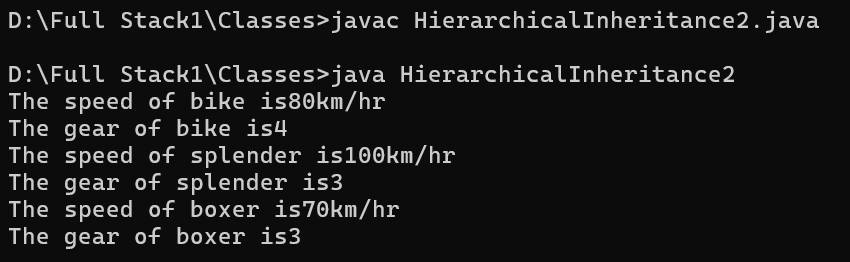
boxer b1=new boxer(3,70);

b1.getspeed();

b1.getgear();

}

}



Code 9:

We achieve compile-time polymorphism by overloading the display method with different parameter lists, allowing method resolution during compilation.

public class MethodOverloading {

public void display(){

System.out.println("I am first method without any parameter");

}

public void display(int a){

System.out.println("I am second method with parameter a which is equals to:"+a);

}

public static void main(String args[]){

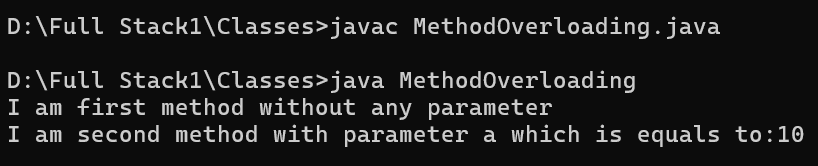
MethodOverloading m1=new MethodOverloading();

m1.display();

m1.display(10);

}

}



Code 10:

We achieve runtime polymorphism by overriding the display method in the derived class D2, allowing the method behavior to change based on the object type.

class D1{

public void display(){

System.out.println("Method of class D1");

}

}

class D2 extends D1{

public void display(){

System.out.println("Same Method of class D2");

}

}

class Override{

public static void main(String args[]){

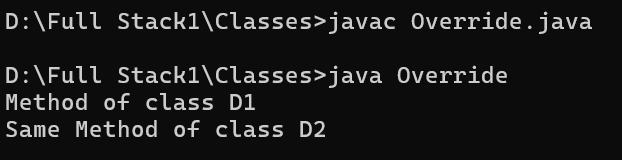
D1 d=new D1();

d.display();

D2 b=new D2();

b.display();

}}



Code 11:

class D1{

public void display(){

System.out.println("Method of class D1");

}

}

class D2 extends D1{

protected void display(){

System.out.println("Same Method of class D2");

}

}

class Override{

public static void main(String args[]){

D1 d=new D1();

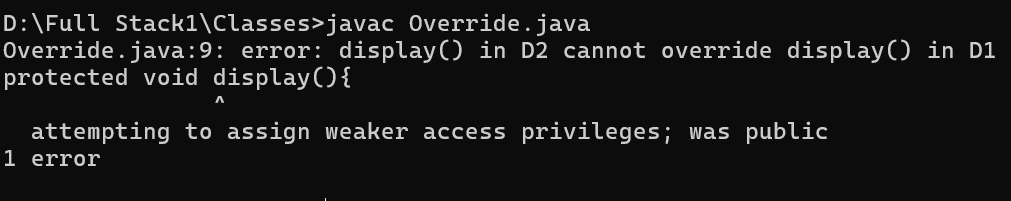
d.display();

D2 b=new D2();

b.display();

}}

The display method in D2 has narrower access (protected) than the display method in D1 (public), violating method overriding rules which require equal or broader visibility.



Code 12:

class D1{

protected void display(){

System.out.println("Method of class D1");

}

}

class D2 extends D1{

public void display(){

System.out.println("Same Method of class D2");

}

}

class Override{

public static void main(String args[]){

D1 d=new D1();

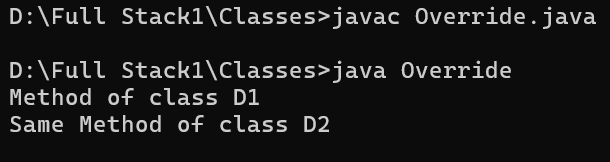
d.display();

D2 b=new D2();

b.display();

}}

The display method in D2 has broader access (public) than the display method in D1 (protected), maintaining proper overriding rules.



Code 13:

abstract class BasicAbstract{

public abstract void display();

public void show(){

System.out.println("I am non Abstract Method");

}}

class ImplentationClass extends BasicAbstract{

public void display(){

System.out.println("I am Abstract Method");}}

public class TestAbstraction{

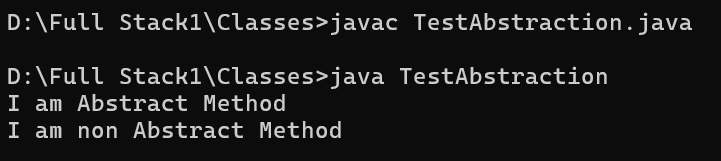
public static void main(String[] args){

ImplentationClass ic=new ImplentationClass();

ic.display();

ic.show();

}}



Code 14:

abstract class Shape{

public abstract void area();

public abstract void perimeter();

}

class Circle extends Shape{

float radius=10.10f;

float pi=3.14f;

float area, perimeter;

public void area(){

area=pi\*radius\*radius;

System.out.println("Area of Circle is:"+area);

}

public void perimeter(){

area=pi\*radius\*2;

System.out.println("Area of Circle is:"+perimeter);

}

}

class TestArea{

public static void main(String args[]){

Circle c=new Circle();

c.area();

c.perimeter();}

}

