BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

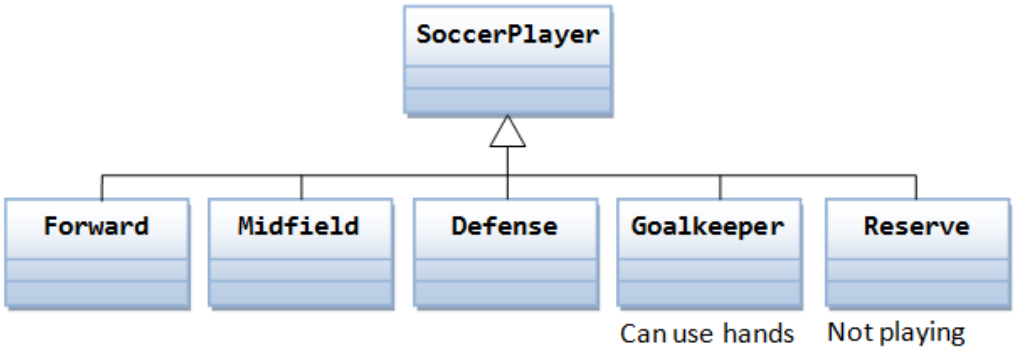
CS F213 – Object Oriented Programming

Lab 6: Inheritance, Polymorphism, Abstract Classes, Reflections

Time:	2 hours
Objective:	To understand and implement OOP concepts like inheritance, polymorphism, abstract classes, and reflection in Java.
Concepts Covered:	Class inheritance, method overloading and overriding, constructor overloading, abstract classes and methods, toString() method overriding, and Java reflection API for runtime class inspection and modification.
Prerequisites:	Basic knowledge of Java programming, understanding of classes and objects, familiarity with OOP concepts, and a Java development environment.

1. Inheritance

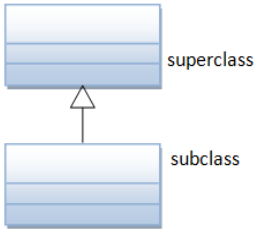
In OOP, we often organize classes in hierarchy to avoid duplication and reduce redundancy. The classes in the lower hierarchy inherit all the state variables and methods from the higher hierarchies. A class in the lower hierarchy is called a **subclass** (or derived, child, extended class). A class in the upper hierarchy is called a **superclass** (or base, parent class). For example,



In Java, you define a subclass using the keyword "extends", e.g.

```
class Cylinder extends Circle {......}
class Goalkeeper extends SoccerPlayer {......}
```

UML Notation: The UML notation for inheritance is a solid line with a hollow arrowhead leading from the subclass to its superclass. By convention, superclass is drawn on top of its subclasses as shown.



1.1. Examples on Inheritance

In this example, we derive a subclass called **`Cylinder`** from the superclass **`Circle`**. The class **`Cylinder`** inherits all the member variables (**`radius`** and **`color`**) and methods (**`getRadius(), getArea(),`** among others) from its superclass **`Circle`**. It further defines a variable called **`height`**, two public methods - **`getHeight()`** and **`getVolume()`** and its own constructors.

Let's consider a Circle superclass and Cylinder subclass.

`Circle.java`

```
// Define the Circle class

public class Circle { // Save as "Circle.java"
// Private variables
      private double radius;
      private String color;

// Constructors (overloaded)
      public Circle() { // 1st Constructor
            radius = 1.0;
            color = "red";
      }

      public Circle(double r) { // 2nd Constructor
            radius = r;
            color = "red";
      }

      public Circle(double r, String c) { // 3rd Constructor
            radius = r;
            color = c;
      }

// Public methods
      public double getRadius() {
            return radius;
      }

      public String getColor() {
            return color;
      }

      public double getArea() {
            return radius * radius * Math.PI;
      }
}
```

`Cyllinder.java`

```
// Define Cylinder class, which is a subclass of Circle
public class Cylinder extends Circle {
      private double height; // Private member variable
```

```
	public Cylinder() { // constructor 1
		super(); // invoke superclass' constructor Circle()
		height = 1.0;
	}

	public Cylinder(double radius, double height) { // Constructor 2
		super(radius); // invoke superclass' constructor Circle(radius)
		this.height = height;
	}

	public double getHeight() {
		return height;
	}

	public void setHeight(double height) {
		this.height = height;
	}

	public double getVolume() {
		return getArea() * height; // Use Circle's getArea()
	}
}
```

TestCylinder.java

```
public class TestCylinder {
	public static void main(String[] args) {
	Cylinder cy1 = new Cylinder();         // Use constructor 1
	System.out.println("Radius is " + cy1.getRadius()
	   + " Height is " + cy1.getHeight()
	   + " Color is " + cy1.getColor()
	   + " Base area is " + cy1.getArea()
	   + " Volume is " + cy1.getVolume());
	Cylinder cy2 = new Cylinder(5.0, 2.0); // Use constructor 2
	System.out.println("Radius is " + cy2.getRadius()
	   + " Height is " + cy2.getHeight()
	   + " Color is " + cy2.getColor()
	   + " Base area is " + cy2.getArea()
	   + " Volume is " + cy2.getVolume());
   }
}
```

Keep the "**Cylinder.java**" and "**TestCylinder.java**" in the same directory as "**Circle.class**" (because we are reusing the class Circle). Compile and run the program. The expected output is as follows:

```
Radius is 1.0 Height is 1.0 Color is red Base area is 3.141592653589793 Volume is 3.141592653589793

Radius is 5.0 Height is 2.0 Color is red Base area is 78.53981633974483 Volume is 157.07963267948966
```

The Circle class is a superclass, but not because it is superior to its subclass or contains more functionality. In fact, the opposite is true: Subclasses have more functionality than their superclasses. For example, as you will see when we go over the rest of the Cylinder class code, the Cylinder class encapsulates more data and has more functionality than its superclass Circle.

Let's go over some important keywords here:

super: the super keyword has two uses.

(i) The first use allows to make a call to the constructor of parent class. An implicit call to parent class's constructor is made before the child's constructor is executed. If no-arguments are given with the super(), then the call is made to default no-arg constructor. Be careful, if the parent class's no-arg constructor has been hidden and a constructor with arguments exists for parent class, then arguments must be passed in the super() keyword or a compile-time error will result.

(ii) The second use allows to refer to the parent's members which can be either variables or methods. The syntax is `super.member`.

Exercise 1: Compile and execute the following code by completing it as per commented specification given. Write the whole code in file.

```
// Inheritance example on variables
class A { public int a = 100; } // End of class A
class B extends A { public int a = 80; } // End of class B
class C extends B { public int a = 60; } // End of class C
class D extends C { public int a = 40; } // End of class D

// NOTE : The variable named 'a' used in above classes is the instance field of each class
class E extends D{
      public int a =10;
      public void show(){
            int a =0;
            // Write Java statements to display the values of  all a's used in this file on
              System.out
      } // End of show() Method
}// End of class E

class Ex3Test{
      public static void main(String args[]){
            new E().show(); // This is an example of anonymous object
            A a1 = new E();
            D d1 = (D) a1; // what's wrong with this statement?
      }
}
```

2. Polymorphism

The concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler's job to select the specific action (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

A dog's sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose! This same general concept can be implemented in Java as it applies to methods within a Java program.

Two ways by which Java implements Polymorphism :

- *Compile time*: Overloading. (The discussion on polymorphism in the class pertains here)
- *Run time*: Overriding.

2.1. Method Overloading

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.

Method overloading is one of the ways that Java supports polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call. [1]

Overloaded methods:

- appear in the same class or a subclass
- have the same name but,
- have different parameter lists, and,
- can have different return types

Here's an example:

```
class OverloadDemo {
      void test() {
            System.out.println("No parameters");
      }

// Overload test for one integer parameter.
      void test(int a) {
            System.out.println("a: " + a);
      }
```

[1] Schildt, H. (2014). Java: The Complete Reference, Ninth Edition (INKLING CH). McGraw Hill Professional.

```
// Overload test for two integer parameters.
      void test(int a, int b) {
            System.out.println("a and b: " + a + " " + b);
      }

// Overload test for a double parameter
      double test(double a) {
            System.out.println("double a: " + a);
            return a * a;
      }
}

class Overload {
      public static void main(String args[]) {
            OverloadDemo ob = new OverloadDemo();
            double result;
// call all versions of test()
            ob.test();
            ob.test(10);
            ob.test(10, 20);
            result = ob.test(123.25);
            System.out.println("Result of ob.test(123.25): " + result);
      }
}
```

The output should look like:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

As you can see, `test()` is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter. The fact that the fourth version of `test()` also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution.

2.1.1. Constructor Overloading:

Like method overloading, constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by considering the number of parameters in the list and their type.

```
class Room {
      double length, breadth, height;
```

```
	Room() {// default constructor for class Room
		length = -1;
		breadth = -1;
		height = -1;
	}

	// overloading the constructor
	// Parameterized constructor for the class Room
	Room(double l, double b, double h) {
		length = l;
		breadth = b;
		height = h;
	}

	Room(double len) {// Single parameterized constructor
		length = breadth = height = len;
	}

	double volume() {
		return length * breadth * height;
	}
}

// Demonstrating the use of Overloaded constructors
class OverloadConstructors {
	public static void main(String args[]) {
		Room a = new Room(20, 30, 40);
		Room b = new Room();
		Room c = new Room(10);
		double vol;
		vol = a.volume();
		System.out.println("Volume of room a is " + vol);

		vol = b.volume();
		System.out.println("Volume of room b is " + vol);

		vol = c.volume();
		System.out.println("Volume of room c is " + vol);
	}
}
```

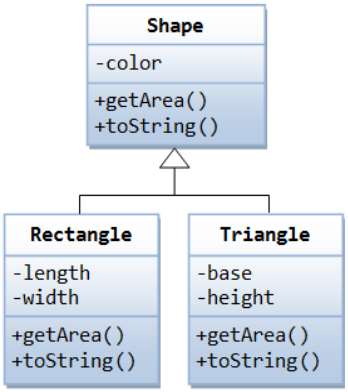
2.2. Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

- applies ONLY to inherited methods is related to polymorphism
- object type (NOT reference variable type) determines which overridden method will be used at runtime

- overriding method **MUST** have the same argument list (if not, it might be a case of overloading)
- overriding method **MUST** have the same return type; the exception is covariant return (used as of Java 5) which returns a type that is a subclass of what is returned by the overridden method
- overriding method **MUST NOT** have more restrictive access modifier, but MAY have less restrictive one
- overriding method **MUST NOT** throw new or broader checked exceptions, but MAY throw fewer or narrower checked exceptions or any unchecked exceptions
- abstract methods **MUST** be overridden
- final methods **CANNOT** be overridden
- static methods **CANNOT** be overridden
- constructors **CANNOT** be overridden

2.2.1. Example for Method Overriding



Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on. We should design a superclass called `Shape`, which defines the public interface (or behaviors) of all the shapes. For example, we would like all the shapes to have a method called `getArea()`, which returns the area of that shape. The `Shape` class can be written as follows:

`Shape.java`

```
public class Shape {
// Private member variable
      private String color;

// Constructor
      public Shape(String color) {
            this.color = color;
      }

      @Override
      public String toString() {
            return "Shape of color=\"" + color + "\"";
      }

// All shapes must has a method called getArea()
```

```
	public double getArea() {
		System.err.println("Shape unknown! Cannot compute area!");
		return 0; // Need a return to compile the program
	}
}
```

We can then derive subclasses, such as **Triangle** and **Rectangle**, from the superclass **Shape**.

Rectangle.java

```
public class Rectangle extends Shape {
// Private member variables
	private int length;
	private int width;

// Constructor
	public Rectangle(String color, int length, int width) {
		super(color);
		this.length = length;
		this.width = width;
	}

	@Override
	public String toString() {
		return "Rectangle of length=" + length + " and width=" + width + ", subclass of
			"+ super.toString();
	}

	@Override
	public double getArea() {
		return length * width;
	}
}
```

Triangle.java

```
public class Triangle extends Shape {
// Private member variables
	private int base;
	private int height;

// Constructor
	public Triangle(String color, int base, int height) {
		super(color);
		this.base = base;
		this.height = height;
	}

	@Override
	public String toString() {
		return "Triangle of base=" + base + " and height=" + height + ", subclass of " +
super.toString();
	}

	@Override
```

```
	public double getArea() {
		return 0.5 * base * height;
	}
}
```

The subclasses override the `getArea()` method inherited from the superclass and provide the proper implementations for `getArea()`.

`TestShape.java`

```
public class TestShape {
	public static void main(String[] args) {
		Shape s1 = new Rectangle("red", 4, 5);
		System.out.println(s1);
		System.out.println("Area is " + s1.getArea());
		Shape s2 = new Triangle("blue", 4, 5);
		System.out.println(s2);
		System.out.println("Area is " + s2.getArea());
	}
}
```

Compile the above code and observe the output. As an exercise, create another body like a Cuboid extending from Rectangle and write the respective driver code to test it.

2.3. Overriding the toString():

Overriding the `toString()` method in Java is essential because it provides a meaningful string representation of an object, making it easier to understand its data immediately. By default, the `toString()` method of the Object class returns the object's memory reference, which is often not useful. When overridden, it allows developers to customize the output, such as displaying an object's attributes in a human-readable format, making debugging and logging more effective.

When overriding `toString()`, ensure the representation is concise and includes key attributes that define the object's state. Avoid exposing sensitive information like passwords, and ensure the format is intuitive for users and developers. Keeping the representation consistent with the class's purpose enhances code readability and maintainability. An optional `@Override` annotation should be provided, to maintain code clarity. As a rule of thumb, use of annotations should be a developer's habit.

```
/**
 * Java program to demonstrate How to override toString() method in Java.
 */

public class Country {
	private String name;
	private String capital;
	private long population;

	public Country(String name) {
		this.name = name;
	}

	public String getName() {
```

```
            return name;
      }

      public void setName(String name) {
            this.name = name;
      }

      public String getCapital() {
            return capital;
      }

      public void setCapital(String capital) {
            this.capital = capital;
      }

      public long getPopulation() {
            return population;
      }

      public void setPopulation(long population) {
            this.population = population;
      }

      @Override
      public String toString() {
            return "Country [name=" + name + "capital=" + capital + " population=" +
population + "]";
      }

      public static void main(String args[]) {
            Country India = new Country("India");
            India.setCapital("New Delhi");
            India.setPopulation(1200000000);
            System.out.println(India);
      }
}
```

2.4. Exercises

Make a class Employee with attributes

- `name`: String
- `salary`: double.

This class supplies

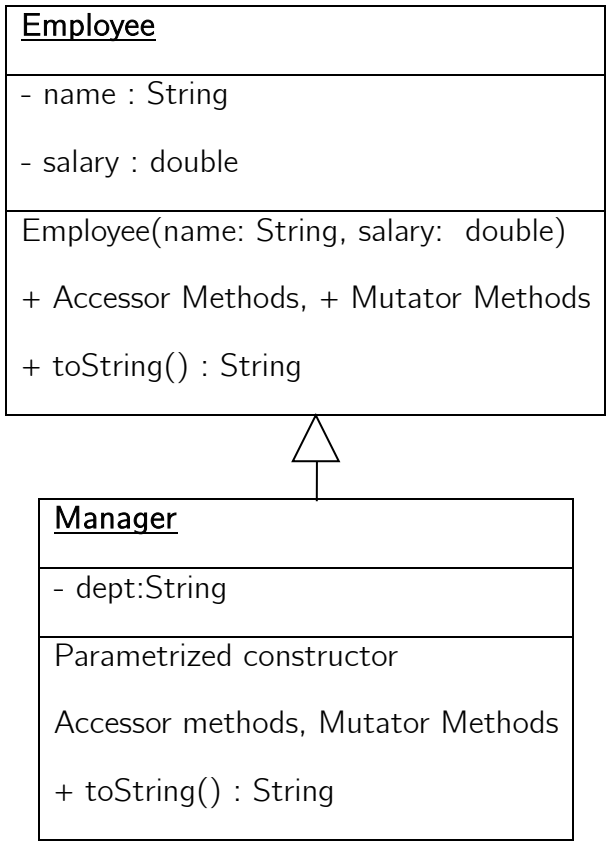
(i) A parameterized constructor

(ii) Accessor and Mutator method(s) for every instance field and

(iii) `toString()` method which returns the values of instance fields by adding proper heading labels and spaces.

Make a class Manager that inherits from Employee and add an instance field named – `department`: String. This class also supplies parameterized constructor, accessor and mutator methods and a `toString`() method that returns the manager's name, department, and salary by adding proper labels and spaces.

2.4.1.1 The complete UML class diagram representation for these classes is shown below:



You have to write the code as per following description:

1. Write java implementations for classes Employee and Manager.
2. Write a Driver code which creates two Employee and Manager instances each and display their attribute values on standard output using polymorphism.

---

3. Abstract Classes

An abstract method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body).

You use the keyword `abstract` to declare an abstract method.

A class containing one or more abstract methods is called an abstract class. An abstract class must be declared with a class-modifier abstract. An abstract class can't be instantiated. Any class extending from the abstract class must either implement the abstract method(s) or be marked as an abstract class itself.

Why Abstract classes?

Abstract classes are a great way to enforce method implementation. They can be used to define a common template for subclasses.

`Shape.java`

```
abstract public class Shape {
	// Private member variable
	private String color;

	// Constructor
	public Shape(String color) {
		this.color = color;
	}

	@Override
	public String toString() {
		return "Shape of color=\"" + color + "\"";
	}

	// All Shape subclasses must implement a method called getArea()
	abstract public double getArea();
}
```

Now, create instances of the subclasses such as **`Triangle`** and **`Rectangle`** (classes which we used previously), and upcast them to **`Shape`** (you cannot create instance of Shape).

```
public class TestShape {
	public static void main(String[] args) {
		Shape s1 = new Rectangle("red", 4, 5);
		System.out.println(s1);
		System.out.println("Area is " + s1.getArea());

		Shape s2 = new Triangle("blue", 4, 5);
		System.out.println(s2);
		System.out.println("Area is " + s2.getArea());

		// Cannot create instance of an abstract class
		Shape s3 = new Shape("green");// Compilation Error!!
	}
}
```

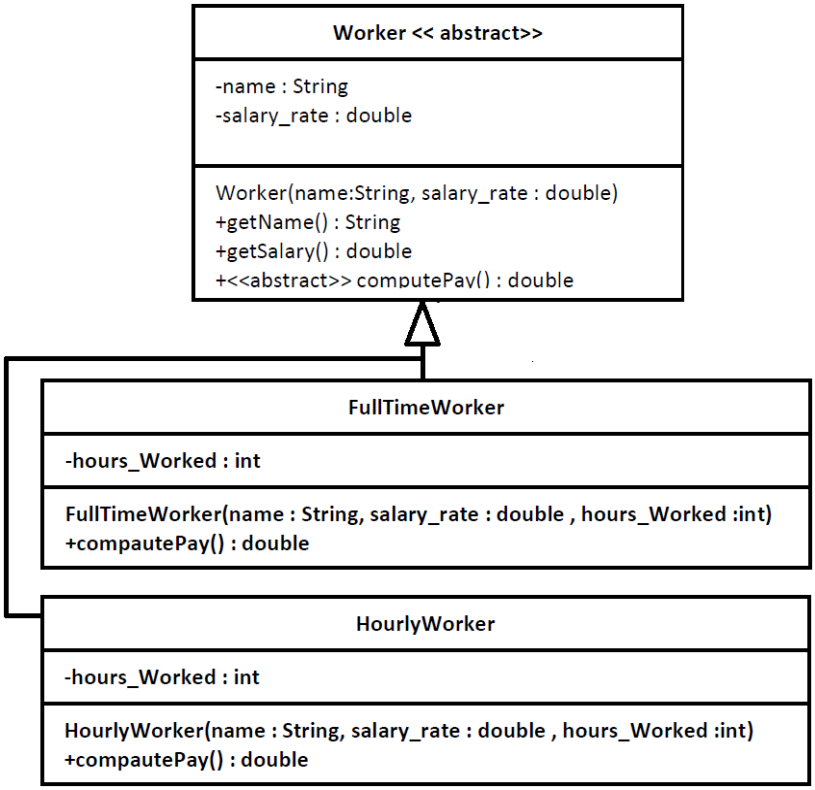
3.1. Exercises

Define an abstract class Worker that has a abstract method public double **`computePay()`**. Every worker has a name and a **`salary_rate`**. Define two concrete classes **`FullTimeWorker`**, and **`HourlyWorker`**. A full time worker gets paid the hourly wage for a maximum of 240 hours in a month at the rate of Rs. 100/hour. An hourly worker gets paid the hourly wage for the actual number of hours he has worked at the rate of Rs. 50/hour, he is not allowed to work for more than 60 hours in a month. The complete UML class diagram is shown in Figure below.

You have to write the code as per following specification:

1. Write the java implementations for classes `Worker`, `HourlyWorker` and `FullTimeWorker`

2. Write a Driver code in the same file `TestWorker.java` which demonstrates late binding.



---

4. Reflections

Reflection is the ability of software to analyze itself. This is provided by the `java.lang.reflect` package and elements in `Class`.

Using Reflection, you can:

- Discover the structure of a class (its fields, methods, constructors, etc.).
- Access private fields and methods.
- Dynamically create objects and invoke methods at runtime.

4.1. Common Use Cases:

- Writing frameworks (like Spring, Hibernate).
- Testing, debugging, and extending libraries without accessing source code.
- Serialization and deserialization (e.g., in JSON or XML frameworks).

4.2. Some key reflection classes in Java

- `Class<?>:` The entry point for using Reflection to inspect a class.
- `Field`` Used to inspect and modify fields.
- `Method:` Used to inspect and invoke methods.

- `Constructor<?>`: Used to inspect and create instances of classes.

How to begin with using Reflections:

- Obtain the Class Object: Every Java class has a Class object associated with it, which provides access to the class structure. You can obtain it in the following ways:
  - Using `ClassName.class`.
  - Using `Object.getClass()`.
  - Using `Class.forName("className")` (useful for loading classes dynamically).
- Access Class Information: Once you have the Class object, you can inspect the class for fields, methods, constructors, etc.
- Modify and Invoke: Using Reflection, you can modify fields (even private ones) and invoke methods.

Let's see reflections in action.

4.3. Examples

4.3.1. Inspecting fields and methods of a class

Task: Write a program that uses Reflection to inspect the fields and methods of a class.

```
// Save this as ReflectionExample1.java
import java.lang.reflect.*;
public class ReflectionExample1 {
	public static void main(String[] args) {
		try {
			// Obtain the Class object for the Person class
			Class<?> personClass = Person.class;
			// Print the class name
			System.out.println("Class Name: " + personClass.getName());
			// Get and print all declared fields (private, public, etc.)
			Field[] fields = personClass.getDeclaredFields();
			System.out.println("\nDeclared Fields:");
			for (Field field : fields) {
				System.out.println(field.getName() + " of type " + field.getType());
			}
			// Get and print all declared methods
			Method[] methods = personClass.getDeclaredMethods();
			System.out.println("\nDeclared Methods:");
			for (Method method : methods) {
				System.out.println(method.getName());
			}
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
class Person {
	private String name;
	private int age;
	public Person(String name, int age) {
```

```
            this.name = name;
            this.age = age;
      }
      public void greet() {
            System.out.println("Hello, my name is " + name + ". I am "+age+" years old.");
      }
}
```

4.3.2. Modifying Private Fields

```
// Save this as ReflectionExample2.java
import java.lang.reflect.*;
class Employee {
      private String name;
      private double salary;

      public Employee(String name, double salary) {
            this.name = name;
            this.salary = salary;
      }
      public void displayInfo() {
            System.out.println("Employee Name: " + name + ", Salary: " + salary);
      }
}
public class ReflectionExample2 {
      public static void main(String[] args) {
            try {
                  // Create an Employee object
                  Employee emp = new Employee("John", 50000);
                  emp.displayInfo();
                  // Obtain the Class object for the Employee class
                  Class<?> empClass = emp.getClass();
                  // Access the private field 'salary'
                  Field salaryField = empClass.getDeclaredField("salary");
                  // Set accessible to true to modify the private field
                  salaryField.setAccessible(true);
                  // Modify the 'salary' field
                  salaryField.set(emp, 75000);
                  // Display modified info
                  emp.displayInfo();
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }
}
```

5. Exercises

5.1. Exercise 1: Autonomous Vehicle System

Task Overview

You will build a hierarchical autonomous vehicle system that simulates self-driving decision-making based on terrain and fuel efficiency. This will require deep use of polymorphism, method overriding, constructor chaining, and dynamic runtime behavior.

1. Create an abstract class `AutonomousVehicle`:
    - Attributes:
        - `fuelEfficiency` (double, km/l)
        - `maxSpeed` (double, km/h)
        - `terrainType` (String: "Land", "Water", "Air")
    - Constructor initializes all attributes.
        - Abstract methods:
        - double `computeTravelTime(double distance)` – calculates time needed to travel.
        - void `adaptToTerrain()` – adjusts vehicle settings based on terrain.
        - `toString()` should return vehicle details.
2. Create a subclass `LandAutonomousVehicle` extending `AutonomousVehicle`:
    - Additional attributes:
        - `numWheels` (int)
    - Overrides:
        - `computeTravelTime()`: distance / maxSpeed.
        - `adaptToTerrain()`: Adjusts fuelEfficiency based on rough/smooth terrain.
3. Create a subclass `WaterAutonomousVehicle` extending `AutonomousVehicle`:
    - Additional attributes:
        - `displacement` (double, in tons)
    - Overrides:
        - `computeTravelTime()`: distance / (maxSpeed * 0.8), since water slows speed.
        - `adaptToTerrain()`: Adjusts fuel efficiency based on water turbulence.
4. Create a subclass `FlyingAutonomousVehicle` extending `AutonomousVehicle`:
    - Additional attributes:
        - `altitudeCapacity` (double, meters)
    - Overrides:
        - `computeTravelTime()`: distance / (maxSpeed * 1.2), since flying is faster.
        - `adaptToTerrain()`: Adjusts fuel efficiency based on altitude.
5. Create a `TestAutonomousSystem` driver program:
    - Instantiate at least 2 objects of each vehicle type.
    - Call `adaptToTerrain()` for each vehicle and print changes.

- Compute and compare travel times for a 1000 km journey.

```
class AutonomousVehicle{
      void adaptToTerrain() {
      }

      public String computeTravelTime(double distance) {
            return null;
      }
}
public class TestAutonomousSystem {
    public static void main(String[] args) {
        AutonomousVehicle[] vehicles = new AutonomousVehicle[6];

        // TODO: Instantiate two LandAutonomousVehicles (one for highway, one for off-road)
        // TODO: Instantiate two WaterAutonomousVehicles (one for calm water, one for stormy
seas)
        // TODO: Instantiate two FlyingAutonomousVehicles (one for low altitude, one for
high altitude)

        double distance = 1000; // Distance in km
        System.out.println("Comparing travel times for a 1000 km journey:");

        for (AutonomousVehicle v : vehicles) {
            v.adaptToTerrain();
            System.out.println(v);
            System.out.println("Travel Time: " + v.computeTravelTime(distance) + "
hours\n");
        }

        // TODO: Determine which vehicle reaches the destination the fastest.
    }
}
```

5.2. Exercise 2: Advanced Banking System

Task Overview

You will build a banking system with inheritance, dynamic role assignment using Java Reflection API and dynamic method invocation. This system will simulate transactions and detect fraudulent activities at runtime.

Problem Statement

1. Create an abstract class `BankAccount`:
   - Private attributes:
     - `accountHolder` (String)
     - `balance` (double)
     - Constructor initializes attributes.
   - Abstract method:
     - `void processTransaction`(double amount, String transactionType)

- `toString()` should return account details.

2. Create a subclass `SavingsAccount` extending `BankAccount`:
   - Additional attribute:
     - `interestRate` (double)
   - Overrides:
     - `processTransaction()`:
   - Deposits are added to balance.
   - Withdrawals cannot exceed 80% of balance.
3. Create a subclass `CheckingAccount` extending `BankAccount`:
   - Additional attribute:
     - `overdraftLimit` (double)
   - Overrides:
     - `processTransaction()`:
   - Deposits are added to balance.
   - Withdrawals can go into overdraft up to the limit.
4. Create a subclass `CorporateAccount` extending `BankAccount`:
   - Additional attribute:
     - `businessName` (String)
   - Overrides:
     - `processTransaction()`:
   - Deposits must exceed ₹50,000 or transaction is rejected.
   - Withdrawals cannot exceed 1 crore per transaction.
5. Write an `AIFraudDetection` class:
   - Dynamically inspects all transactions:
   - Uses reflection to detect suspicious withdrawals (withdrawal > ₹10 lakhs).
   - If fraudulent, method blocks transaction dynamically.
6. Write a `TestBankingSystem` driver program:
   - Use Reflection to invoke `processTransaction()` dynamically.
   - Simulate transactions and AI-based fraud detection.
   - Randomly swap balance amounts between accounts at runtime.

```
import java.lang.reflect.*;

public class TestBankingSystem {
    public static void main(String[] args) {
        BankAccount[] accounts = new BankAccount[3];

        // TODO: Instantiate accounts (SavingsAccount, CheckingAccount, CorporateAccount)

        try {
            Class<?> fraudClass = AIFraudDetection.class;
            Method fraudMethod = fraudClass.getMethod("detectFraud", BankAccount.class,
double.class);
```

```
            for (BankAccount acc : accounts) {
                System.out.println("\nProcessing Transactions for: " + acc);

                // TODO: Dynamically invoke processTransaction() using reflection
                Method transactionMethod = acc.getClass().getMethod("processTransaction",
double.class, String.class);

                // Simulate transactions
                double amount = Math.random() * 1000000; // Random amount
                transactionMethod.invoke(acc, amount, "withdraw");

                // Invoke fraud detection
                fraudMethod.invoke(null, acc, amount);
            }

            // TODO: Swap balances between accounts using reflection.
            Field balanceField = BankAccount.class.getDeclaredField("balance");
            balanceField.setAccessible(true);

            int first = (int) (Math.random() * accounts.length);
            int second = (int) (Math.random() * accounts.length);

            double tempBalance = (double) balanceField.get(accounts[first]);
            balanceField.set(accounts[first], balanceField.get(accounts[second]));
            balanceField.set(accounts[second], tempBalance);

            System.out.println("\nAfter balance swapping:");
            for (BankAccount acc : accounts) {
                System.out.println(acc);
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```