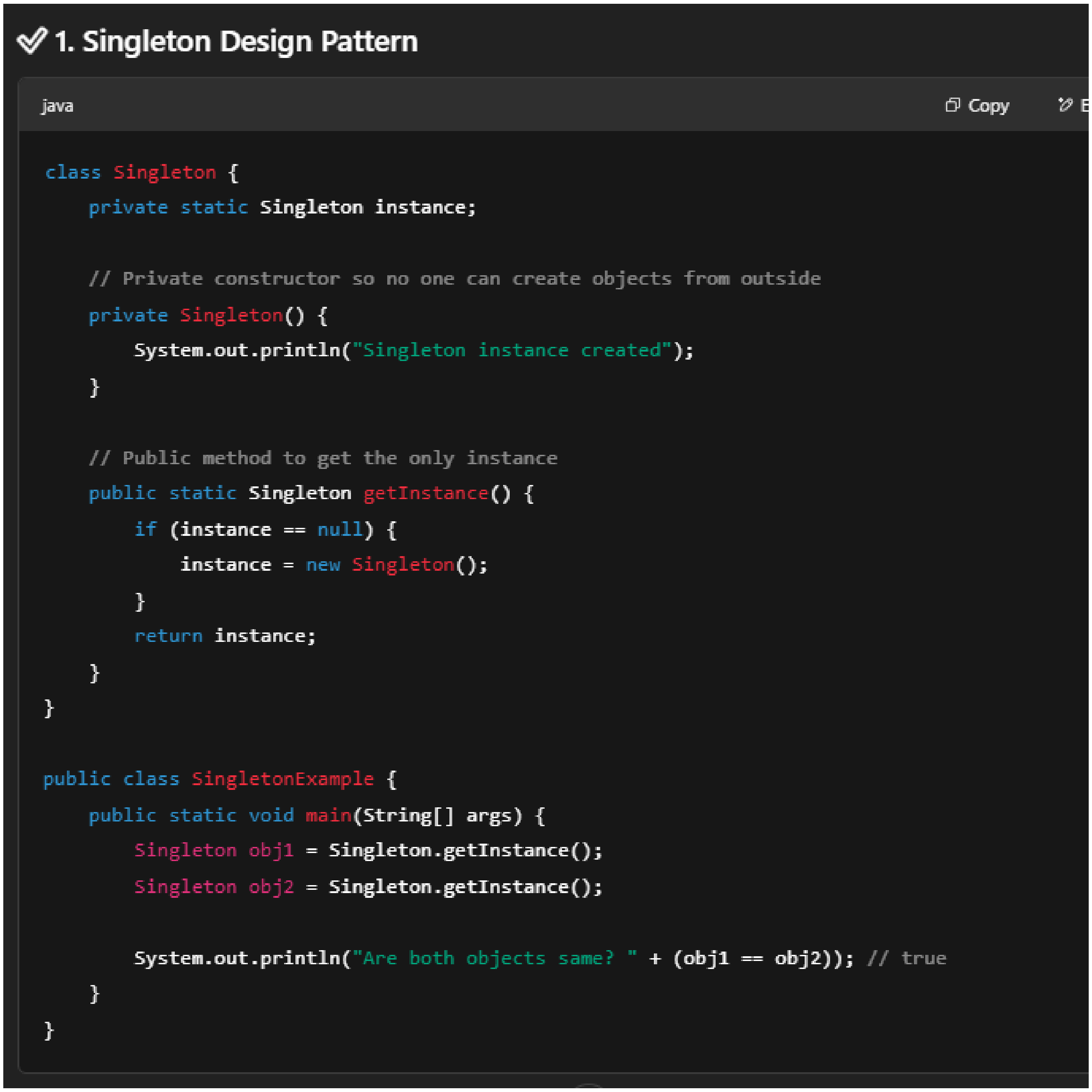
1. Implement Singleton Design Pattern

Write a Java class that ensures only one instance is created. Show how to access this instance from multiple points.

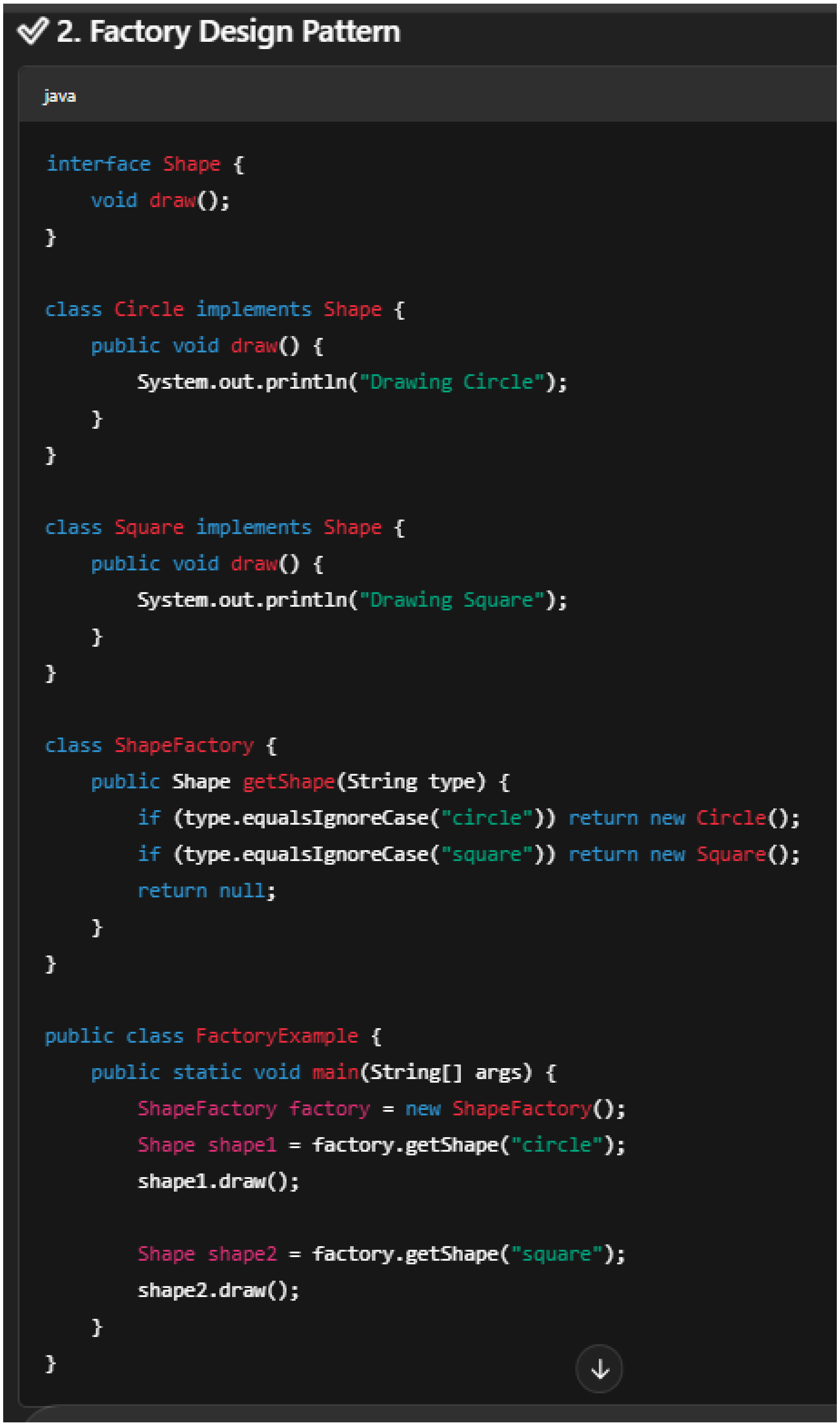


✓ 1. Singleton Design Pattern

```java
class Singleton {
    private static Singleton instance;

    // Private constructor so no one can create objects from outside
    private Singleton() {
        System.out.println("Singleton instance created");
    }

    // Public method to get the only instance
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}

public class SingletonExample {
    public static void main(String[] args) {
        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = Singleton.getInstance();

        System.out.println("Are both objects same? " + (obj1 == obj2)); // true
    }
}
```

2. Implement Factory Design Pattern

Create a factory method that returns different types of shapes (e.g., Circle, Square) based on input.

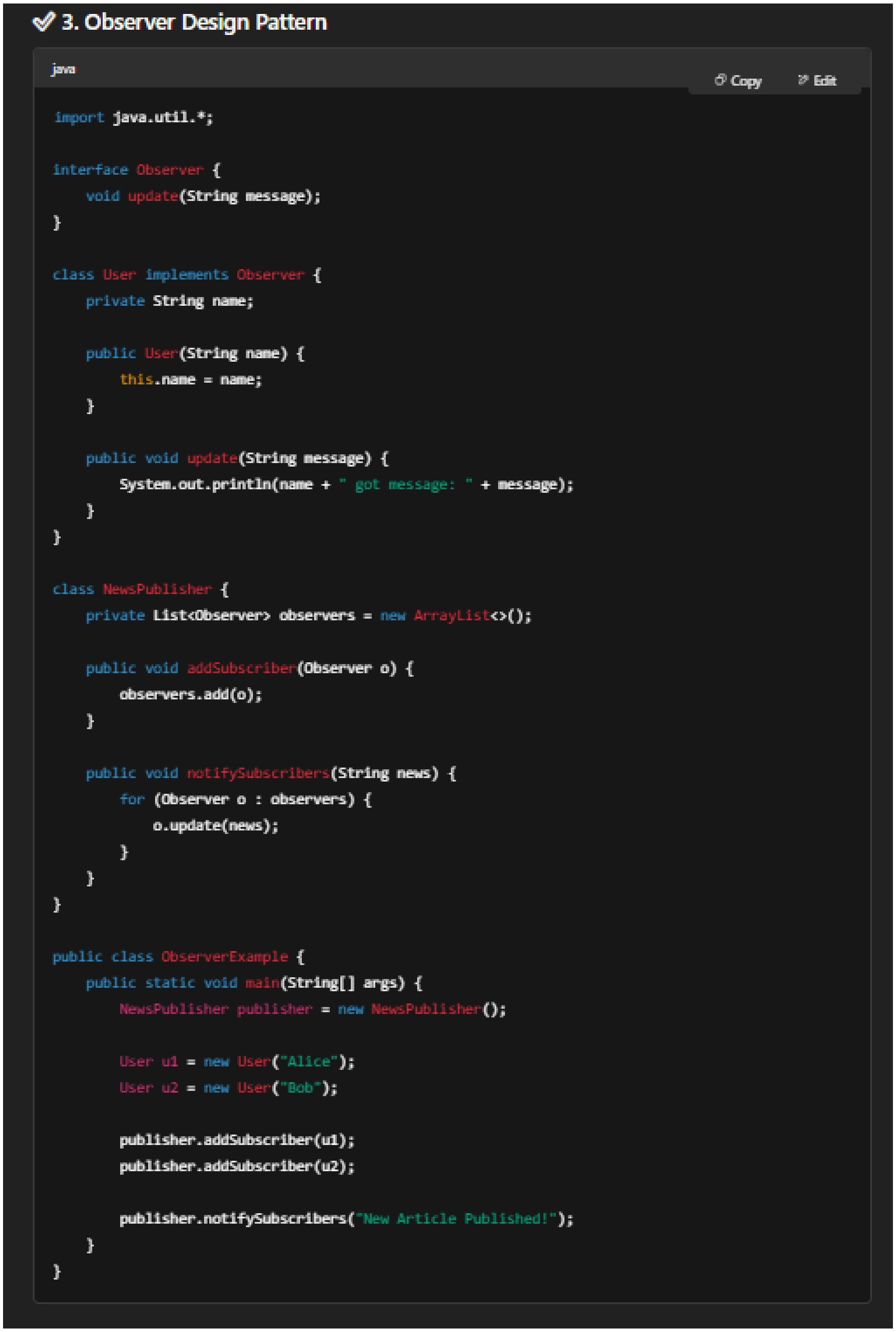


✔ 2. Factory Design Pattern

```java
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

class Square implements Shape {
    public void draw() {
        System.out.println("Drawing Square");
    }
}

class ShapeFactory {
    public Shape getShape(String type) {
        if (type.equalsIgnoreCase("circle")) return new Circle();
        if (type.equalsIgnoreCase("square")) return new Square();
        return null;
    }
}

public class FactoryExample {
    public static void main(String[] args) {
        ShapeFactory factory = new ShapeFactory();
        Shape shape1 = factory.getShape("circle");
        shape1.draw();

        Shape shape2 = factory.getShape("square");
        shape2.draw();
    }
}
```

3. Implement Observer Design Pattern

Create a subject-observer structure where multiple observers get notified when the subject's state changes.

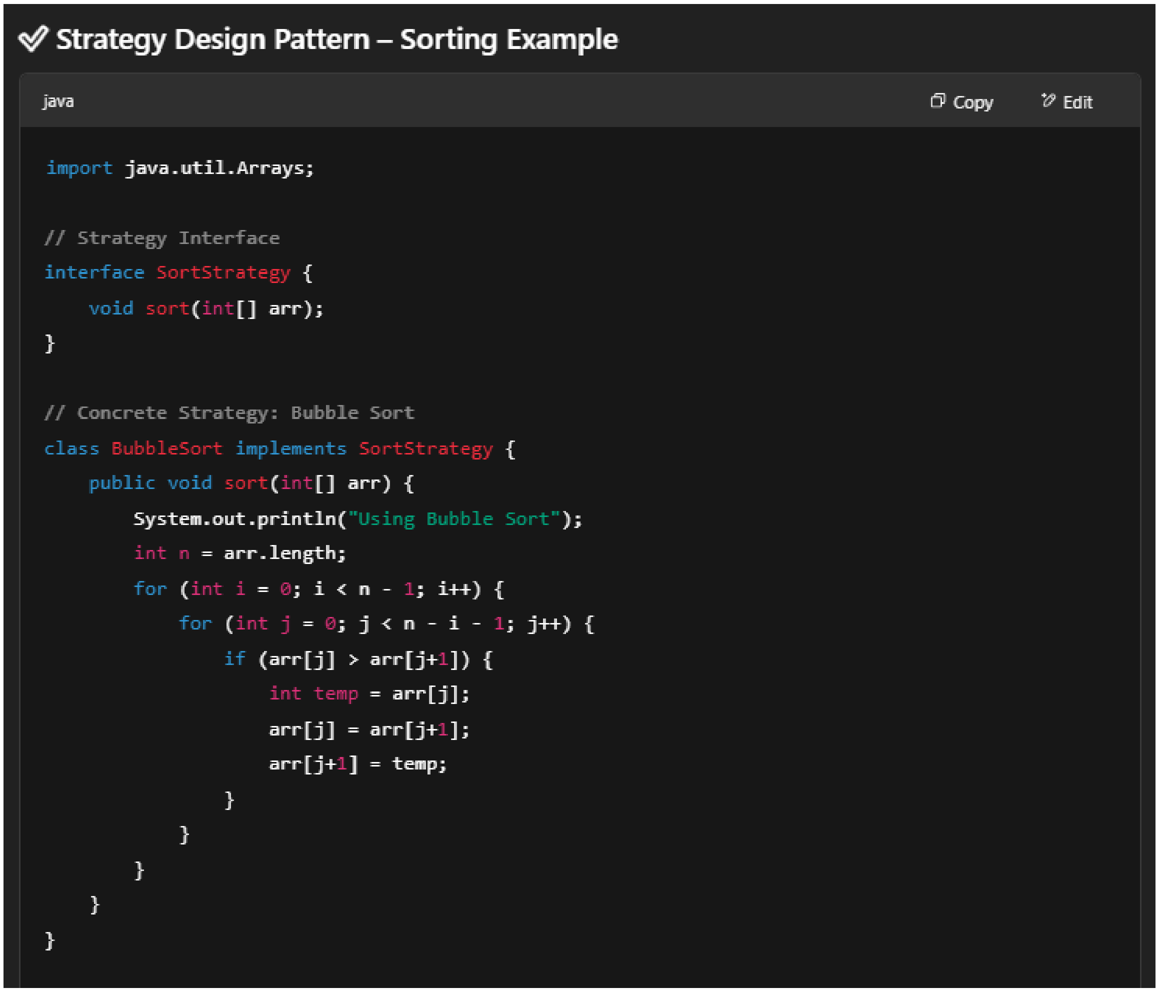


✅ 3. Observer Design Pattern

```java
import java.util.*;

interface Observer {
    void update(String message);
}

class User implements Observer {
    private String name;

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

    public void update(String message) {
        System.out.println(name + " got message: " + message);
    }
}

class NewsPublisher {
    private List<Observer> observers = new ArrayList<>();

    public void addSubscriber(Observer o) {
        observers.add(o);
    }

    public void notifySubscribers(String news) {
        for (Observer o : observers) {
            o.update(news);
        }
    }
}

public class ObserverExample {
    public static void main(String[] args) {
        NewsPublisher publisher = new NewsPublisher();

        User u1 = new User("Alice");
        User u2 = new User("Bob");

        publisher.addSubscriber(u1);
        publisher.addSubscriber(u2);

        publisher.notifySubscribers("New Article Published!");
    }
}
```

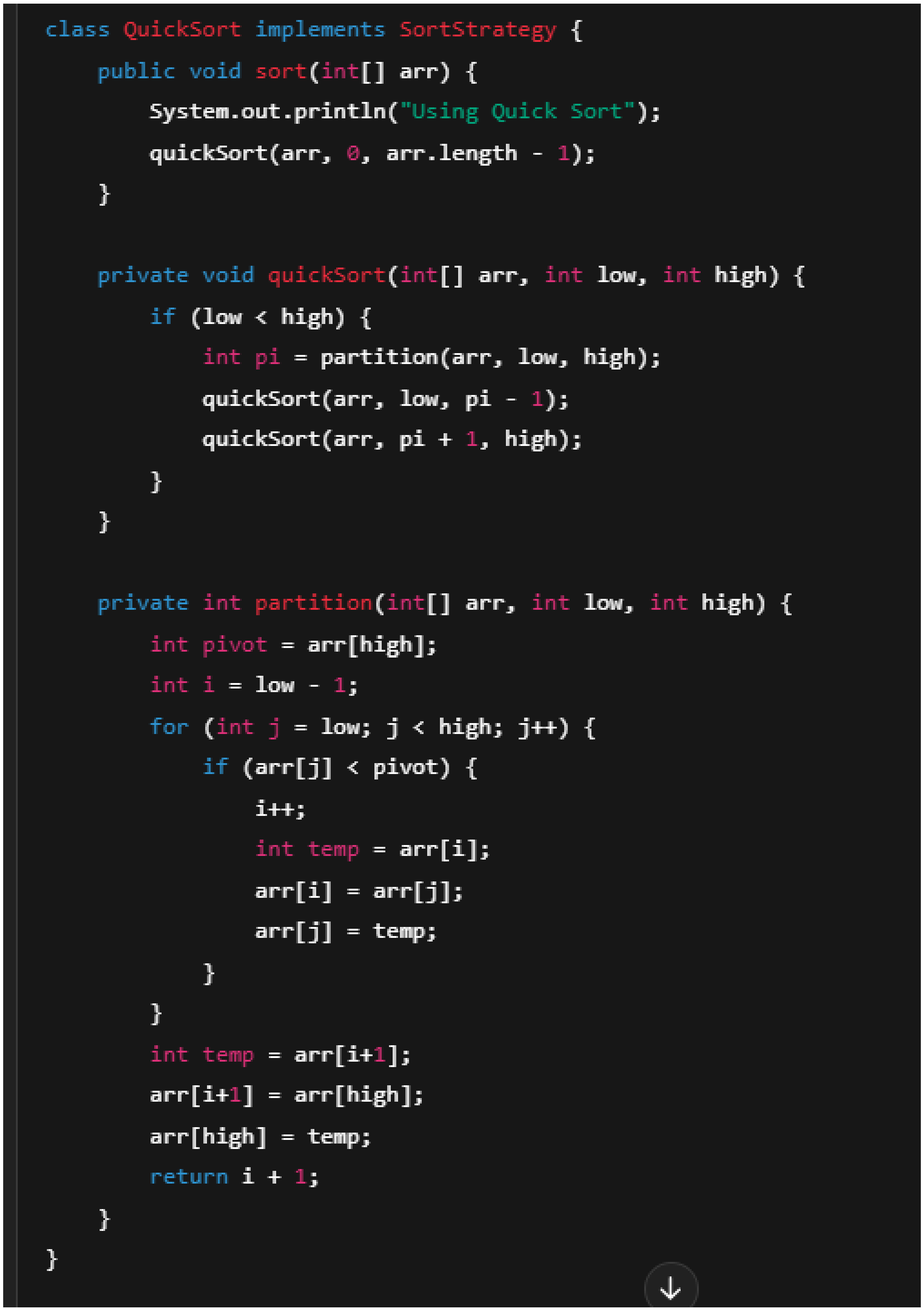
4. Implement Strategy Design Pattern

Create a context class that uses different sorting strategies (bubble sort, quick sort) at runtime.

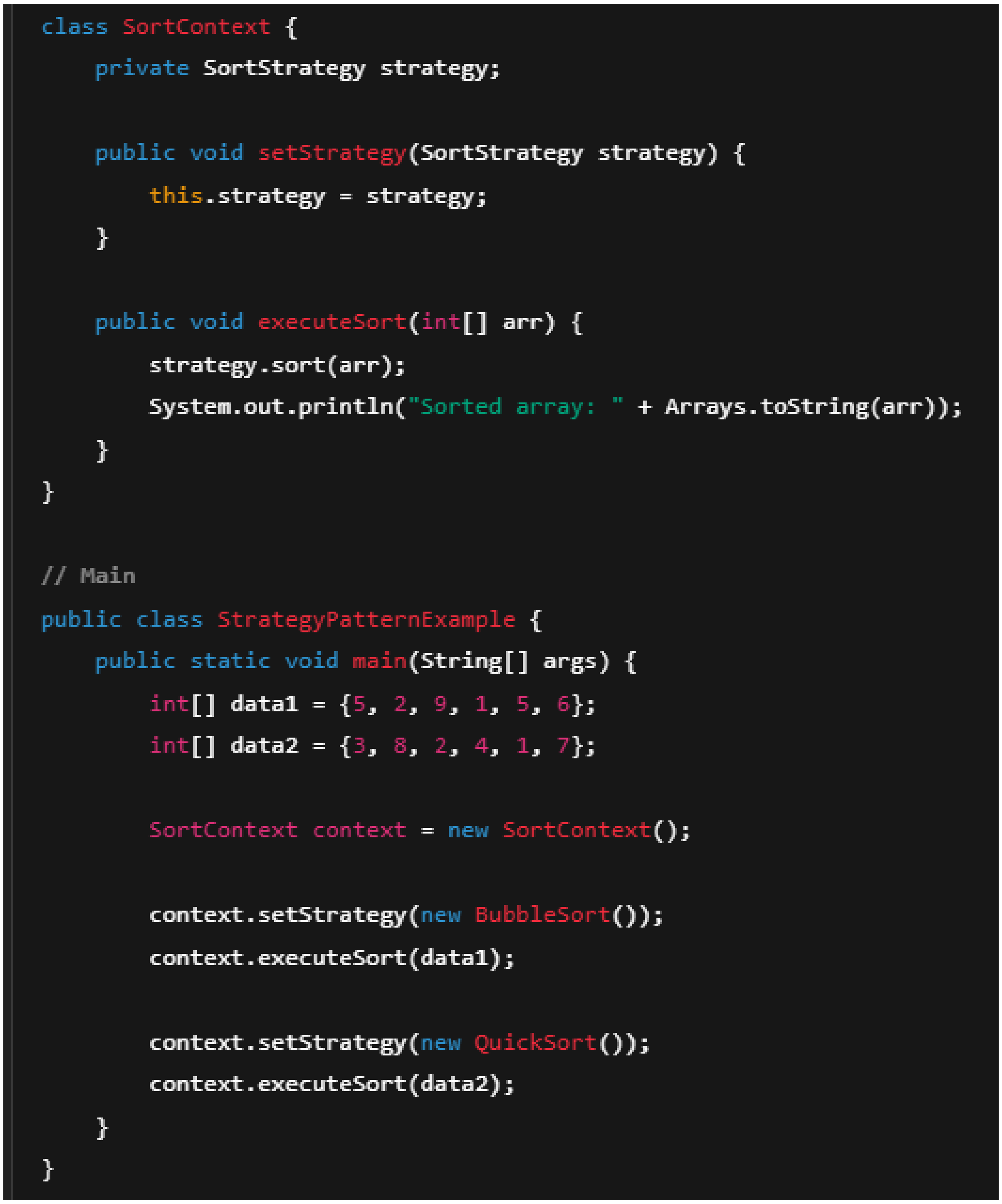


✔ Strategy Design Pattern – Sorting Example

```java
import java.util.Arrays;

// Strategy Interface
interface SortStrategy {
    void sort(int[] arr);
}

// Concrete Strategy: Bubble Sort
class BubbleSort implements SortStrategy {
    public void sort(int[] arr) {
        System.out.println("Using Bubble Sort");
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j+1]) {
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }
}
```



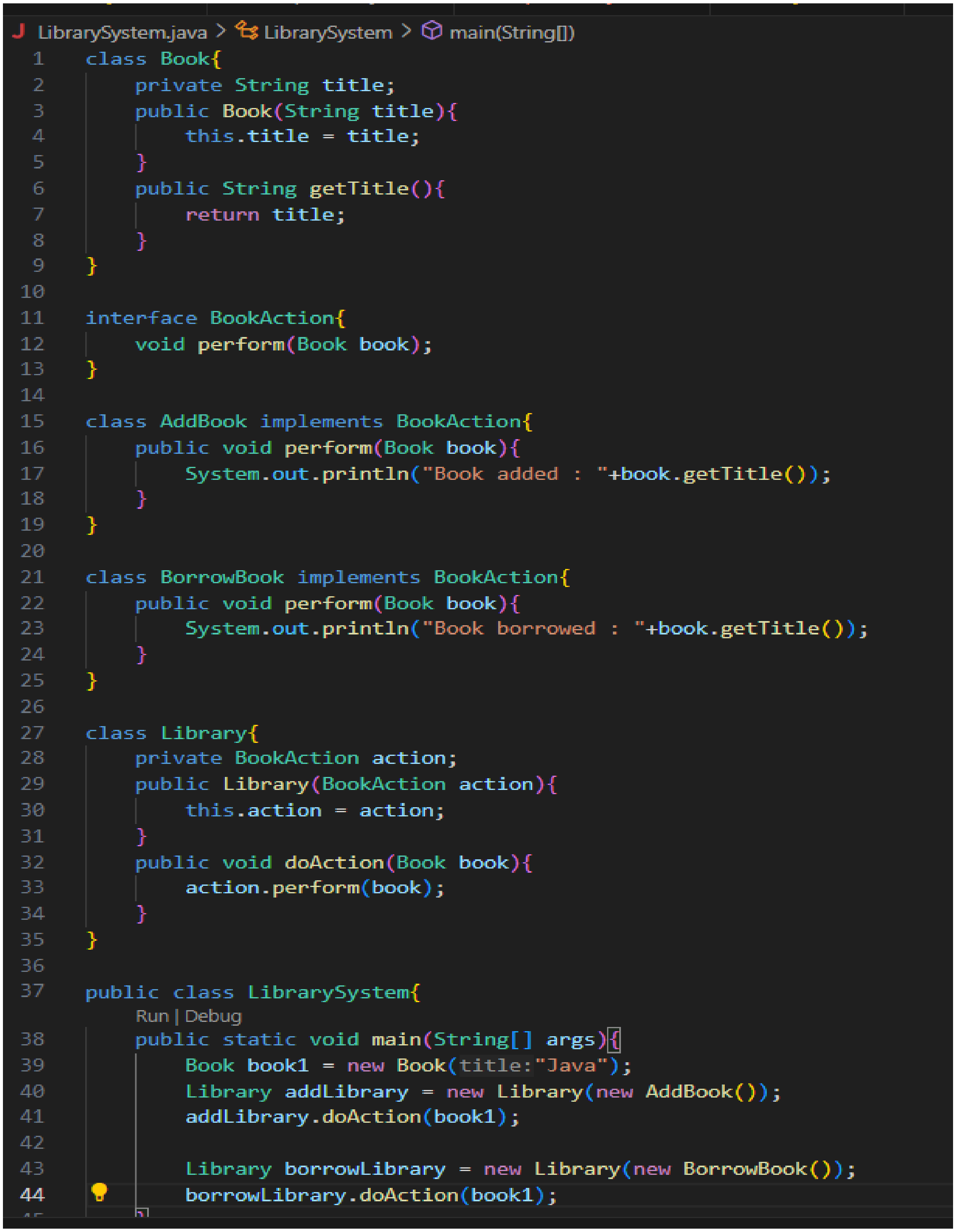
```java
class QuickSort implements SortStrategy {
    public void sort(int[] arr) {
        System.out.println("Using Quick Sort");
        quickSort(arr, 0, arr.length - 1);
    }

    private void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    private int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }
}
```



```java
class SortContext {
    private SortStrategy strategy;

    public void setStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }

    public void executeSort(int[] arr) {
        strategy.sort(arr);
        System.out.println("Sorted array: " + Arrays.toString(arr));
    }
}

// Main
public class StrategyPatternExample {
    public static void main(String[] args) {
        int[] data1 = {5, 2, 9, 1, 5, 6};
        int[] data2 = {3, 8, 2, 4, 1, 7};

        SortContext context = new SortContext();

        context.setStrategy(new BubbleSort());
        context.executeSort(data1);

        context.setStrategy(new QuickSort());
        context.executeSort(data2);
    }
}
```

5. Apply SOLID Principles - Case Study

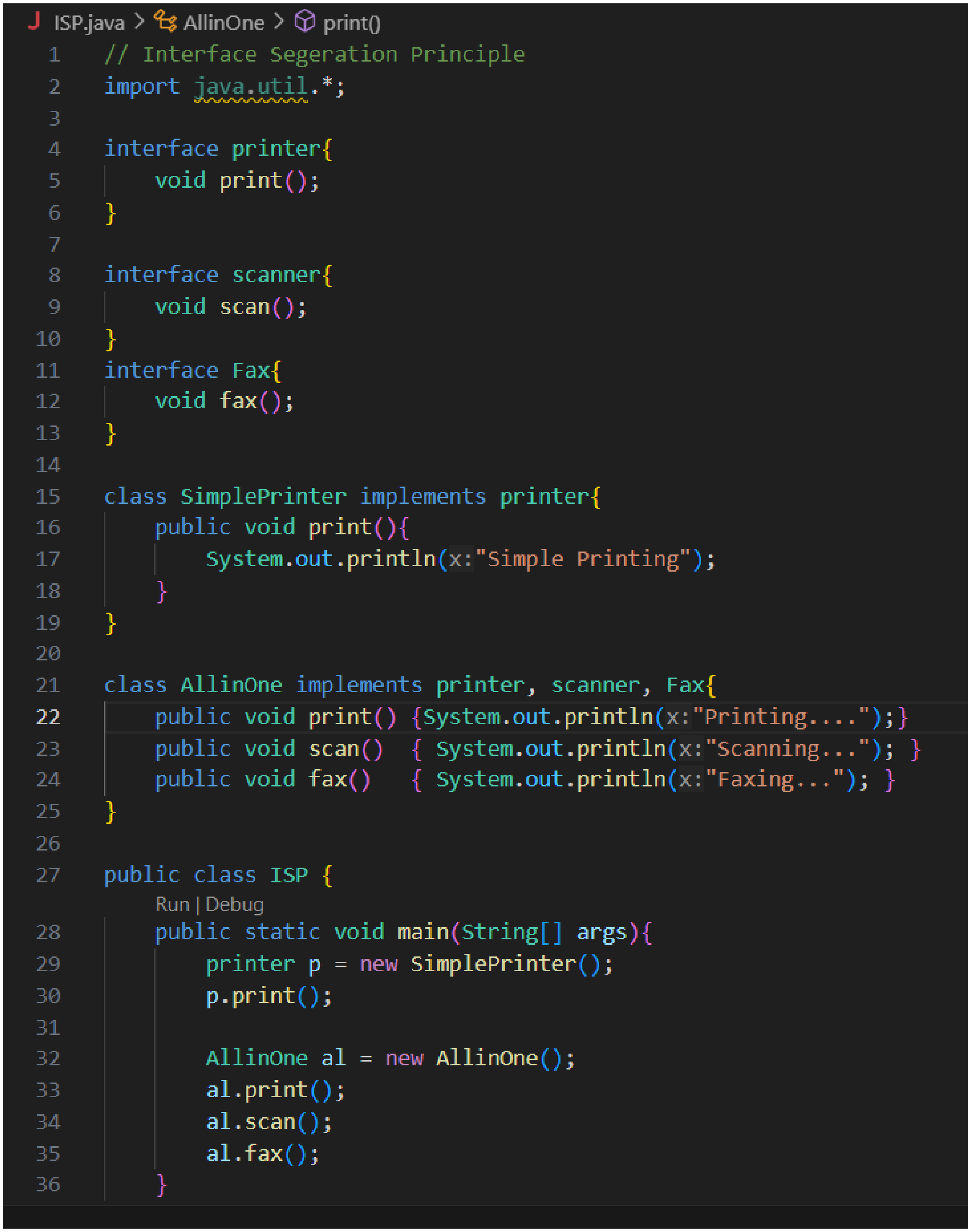
Design a simple library system following all SOLID principles with at least 2-3 classes/interfaces.



```java
class Book{
    private String title;
    public Book(String title){
        this.title = title;
    }
    public String getTitle(){
        return title;
    }
}

interface BookAction{
    void perform(Book book);
}

class AddBook implements BookAction{
    public void perform(Book book){
        System.out.println("Book added : "+book.getTitle());
    }
}

class BorrowBook implements BookAction{
    public void perform(Book book){
        System.out.println("Book borrowed : "+book.getTitle());
    }
}

class Library{
    private BookAction action;
    public Library(BookAction action){
        this.action = action;
    }
    public void doAction(Book book){
        action.perform(book);
    }
}

public class LibrarySystem{
    public static void main(String[] args){
        Book book1 = new Book(title:"Java");
        Library addLibrary = new Library(new AddBook());
        addLibrary.doAction(book1);

        Library borrowLibrary = new Library(new BorrowBook());
        borrowLibrary.doAction(book1);
```

6. Apply Interface Segregation Principle

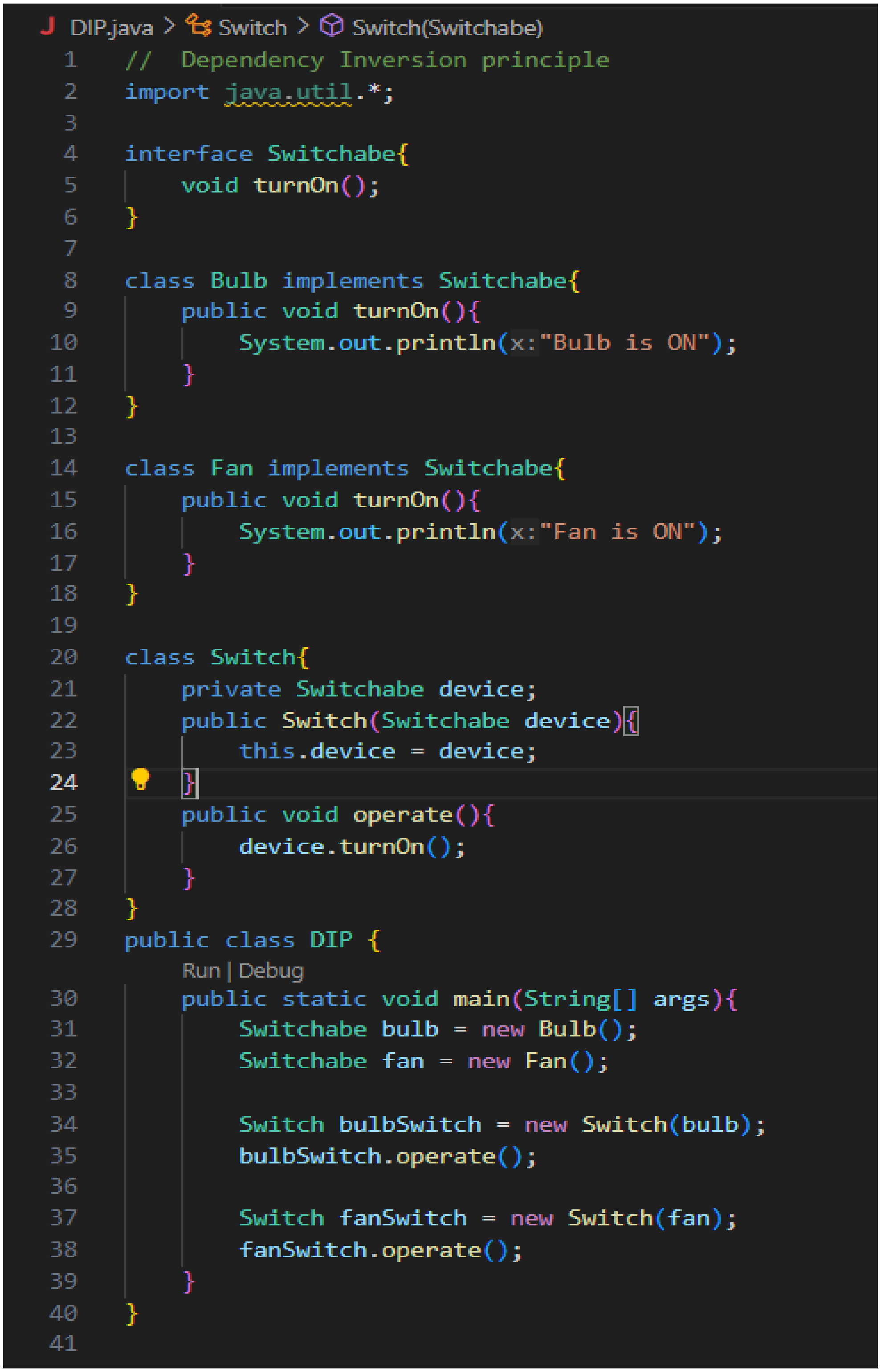
Create separate interfaces for print, scan, and fax operations and implement only required ones.



```java
// Interface Segeration Principle
import java.util.*;

interface printer{
    void print();
}

interface scanner{
    void scan();
}
interface Fax{
    void fax();
}

class SimplePrinter implements printer{
    public void print(){
        System.out.println(x:"Simple Printing");
    }
}

class AllinOne implements printer, scanner, Fax{
    public void print() {System.out.println(x:"Printing....");}
    public void scan()  { System.out.println(x:"Scanning..."); }
    public void fax()   { System.out.println(x:"Faxing..."); }
}

public class ISP {
    public static void main(String[] args){
        printer p = new SimplePrinter();
        p.print();

        AllinOne al = new AllinOne();
        al.print();
        al.scan();
        al.fax();
    }
```

7. Apply Dependency Inversion Principle

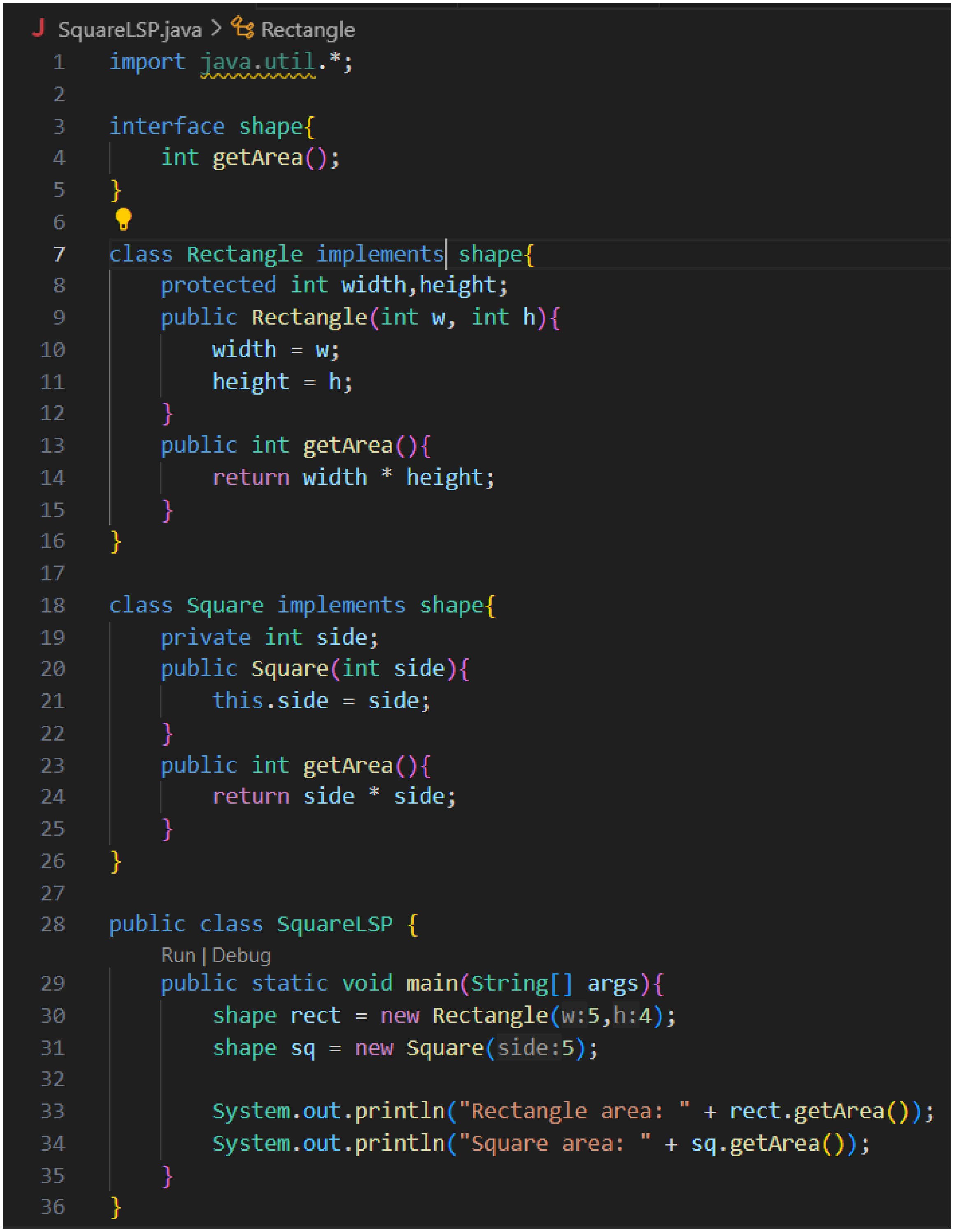
Demonstrate loose coupling by injecting service objects through constructors or interfaces.



```java
//  Dependency Inversion principle
import java.util.*;

interface Switchabe{
    void turnOn();
}

class Bulb implements Switchabe{
    public void turnOn(){
        System.out.println(x:"Bulb is ON");
    }
}

class Fan implements Switchabe{
    public void turnOn(){
        System.out.println(x:"Fan is ON");
    }
}

class Switch{
    private Switchabe device;
    public Switch(Switchabe device){
        this.device = device;
    }
    public void operate(){
        device.turnOn();
    }
}
public class DIP {
    public static void main(String[] args){
        Switchabe bulb = new Bulb();
        Switchabe fan = new Fan();

        Switch bulbSwitch = new Switch(bulb);
        bulbSwitch.operate();

        Switch fanSwitch = new Switch(fan);
        fanSwitch.operate();
    }
}

```

8. Apply Liskov Substitution Principle

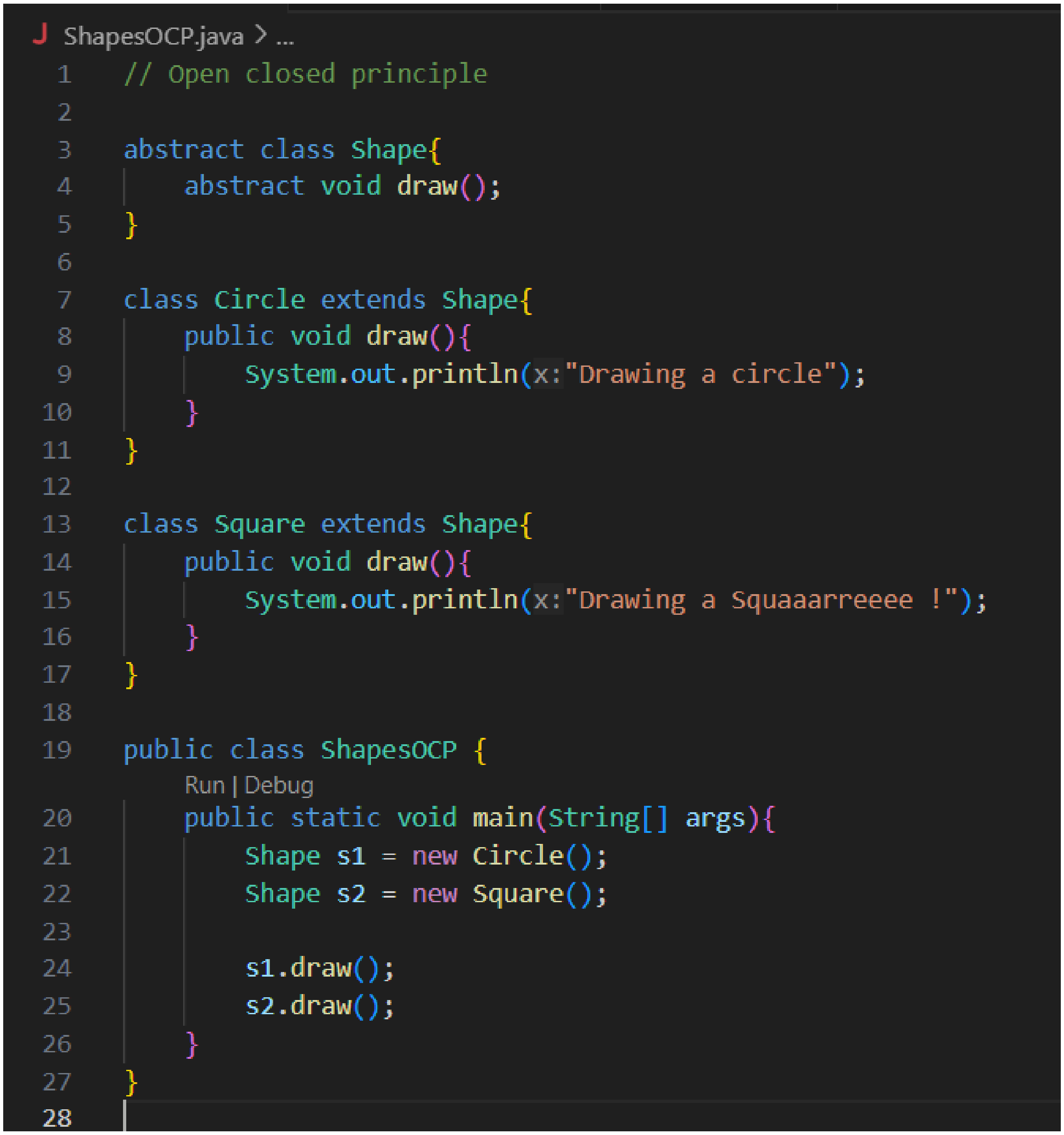
Show how a subclass (e.g., Square) can be substituted for a superclass (e.g., Rectangle) without altering behavior.



```java
import java.util.*;

interface shape{
    int getArea();
}

class Rectangle implements shape{
    protected int width,height;
    public Rectangle(int w, int h){
        width = w;
        height = h;
    }
    public int getArea(){
        return width * height;
    }
}

class Square implements shape{
    private int side;
    public Square(int side){
        this.side = side;
    }
    public int getArea(){
        return side * side;
    }
}

public class SquareLSP {
    public static void main(String[] args){
        shape rect = new Rectangle(w:5,h:4);
        shape sq = new Square(side:5);

        System.out.println("Rectangle area: " + rect.getArea());
        System.out.println("Square area: " + sq.getArea());
    }
}
```

9. Apply Open/Closed Principle

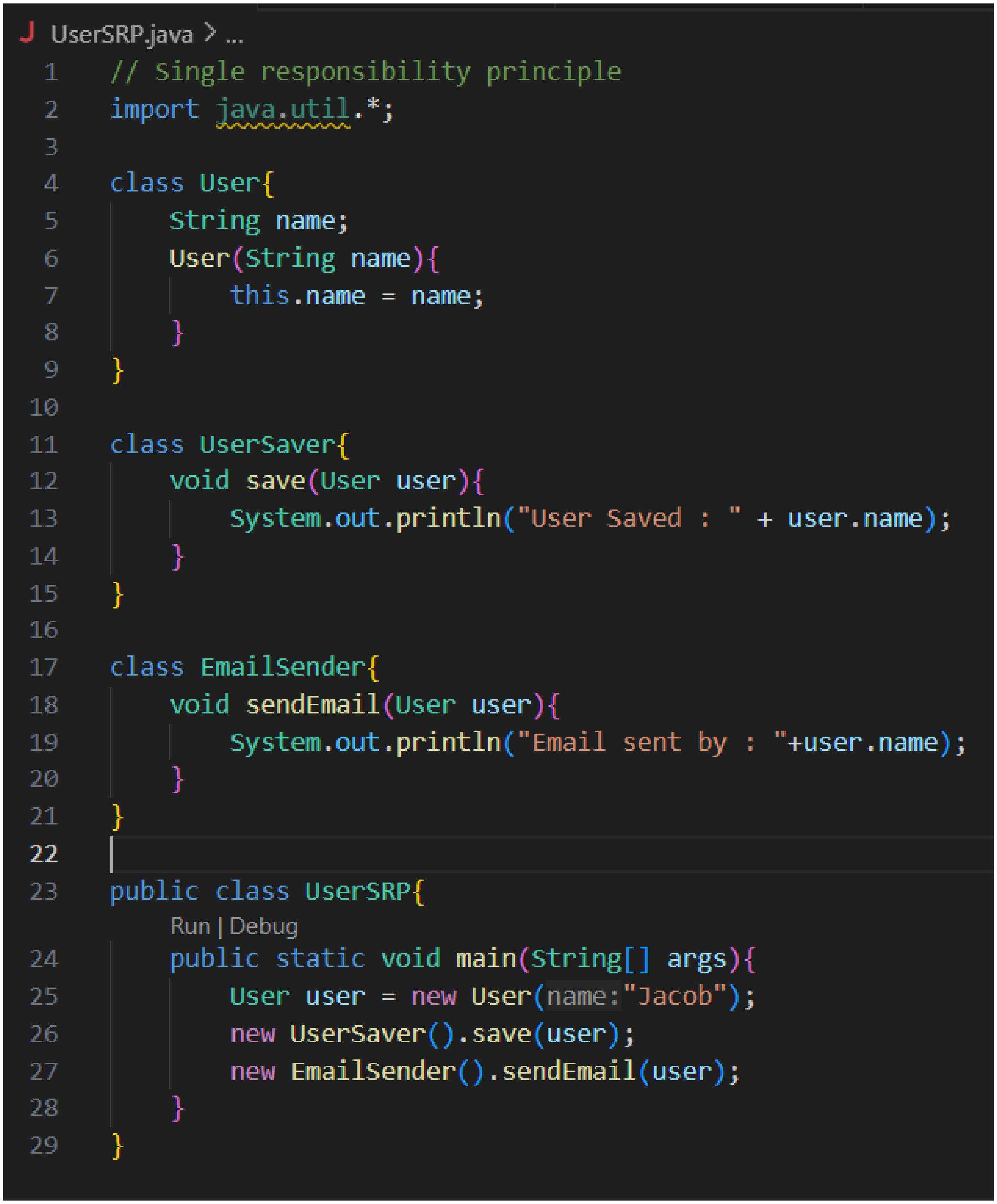
Create a class that can be extended for new functionality without modifying the existing code.



```java
// Open closed principle

abstract class Shape{
    abstract void draw();
}

class Circle extends Shape{
    public void draw(){
        System.out.println(x:"Drawing a circle");
    }
}

class Square extends Shape{
    public void draw(){
        System.out.println(x:"Drawing a Squaaarreeee !");
    }
}

public class ShapesOCP {
    Run | Debug
    public static void main(String[] args){
        Shape s1 = new Circle();
        Shape s2 = new Square();

        s1.draw();
        s2.draw();
    }
}
```

10. Apply Single Responsibility Principle

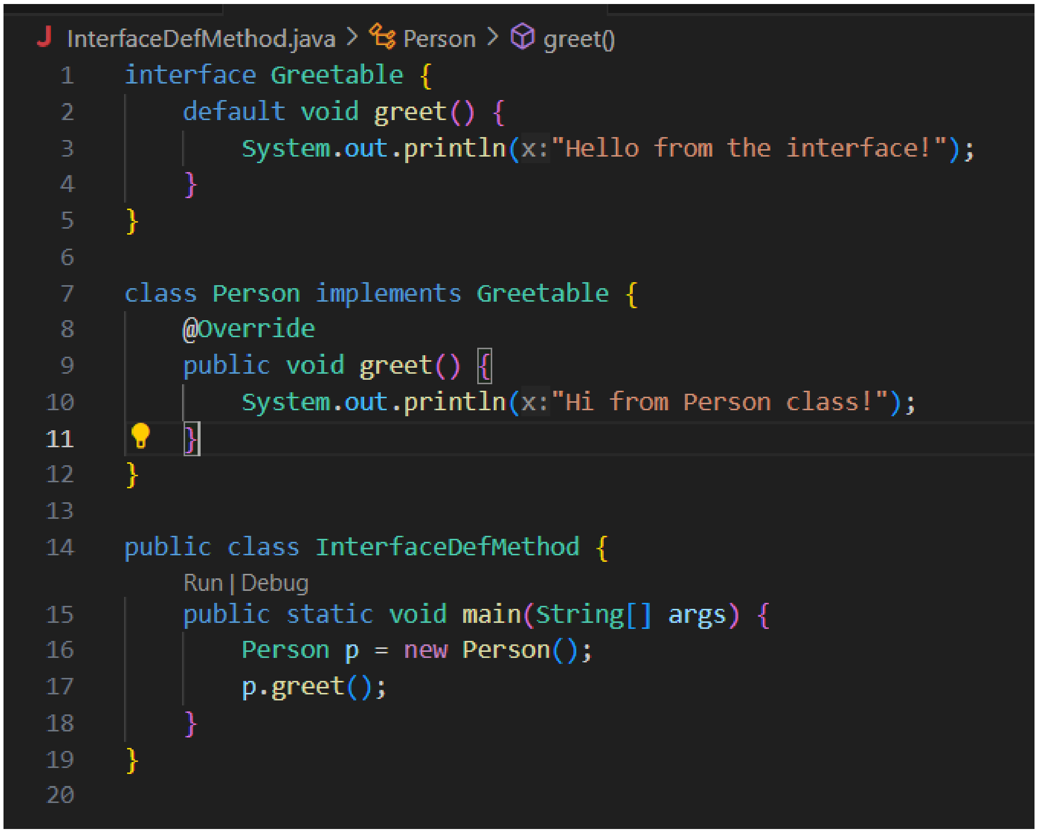
Design a class that performs one specific task like handling user input or processing data.



```java
// Single responsibility principle
import java.util.*;

class User{
    String name;
    User(String name){
        this.name = name;
    }
}

class UserSaver{
    void save(User user){
        System.out.println("User Saved : " + user.name);
    }
}

class EmailSender{
    void sendEmail(User user){
        System.out.println("Email sent by : "+user.name);
    }
}

public class UserSRP{
    public static void main(String[] args){
        User user = new User(name:"Jacob");
        new UserSaver().save(user);
        new EmailSender().sendEmail(user);
    }
}
```

11. Use Interface with Default Method

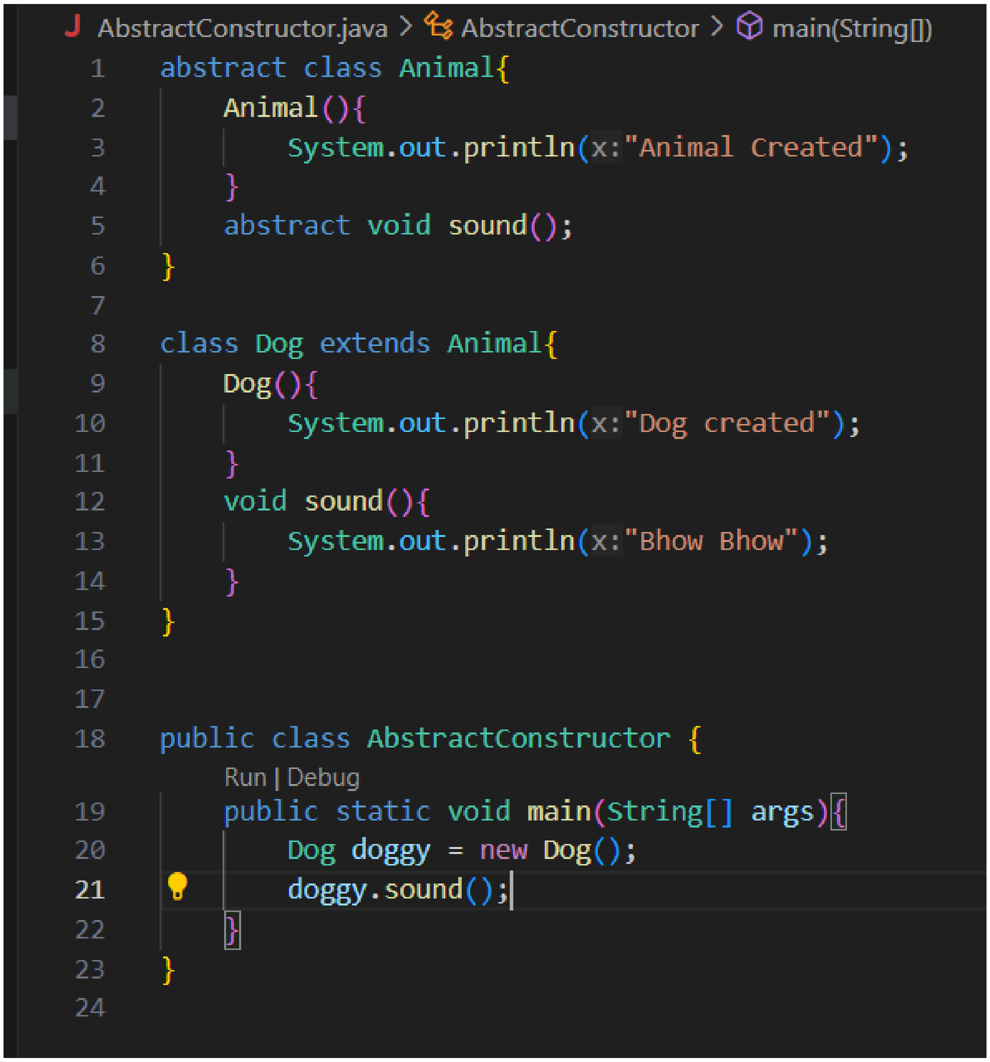
Create an interface with a default greeting method and override it in implementing class.



```java
interface Greetable {
    default void greet() {
        System.out.println(x:"Hello from the interface!");
    }
}

class Person implements Greetable {
    @Override
    public void greet() {
        System.out.println(x:"Hi from Person class!");
    }
}

public class InterfaceDefMethod {
    Run | Debug
    public static void main(String[] args) {
        Person p = new Person();
        p.greet();
    }
}
```

12. Abstract Class with Constructor

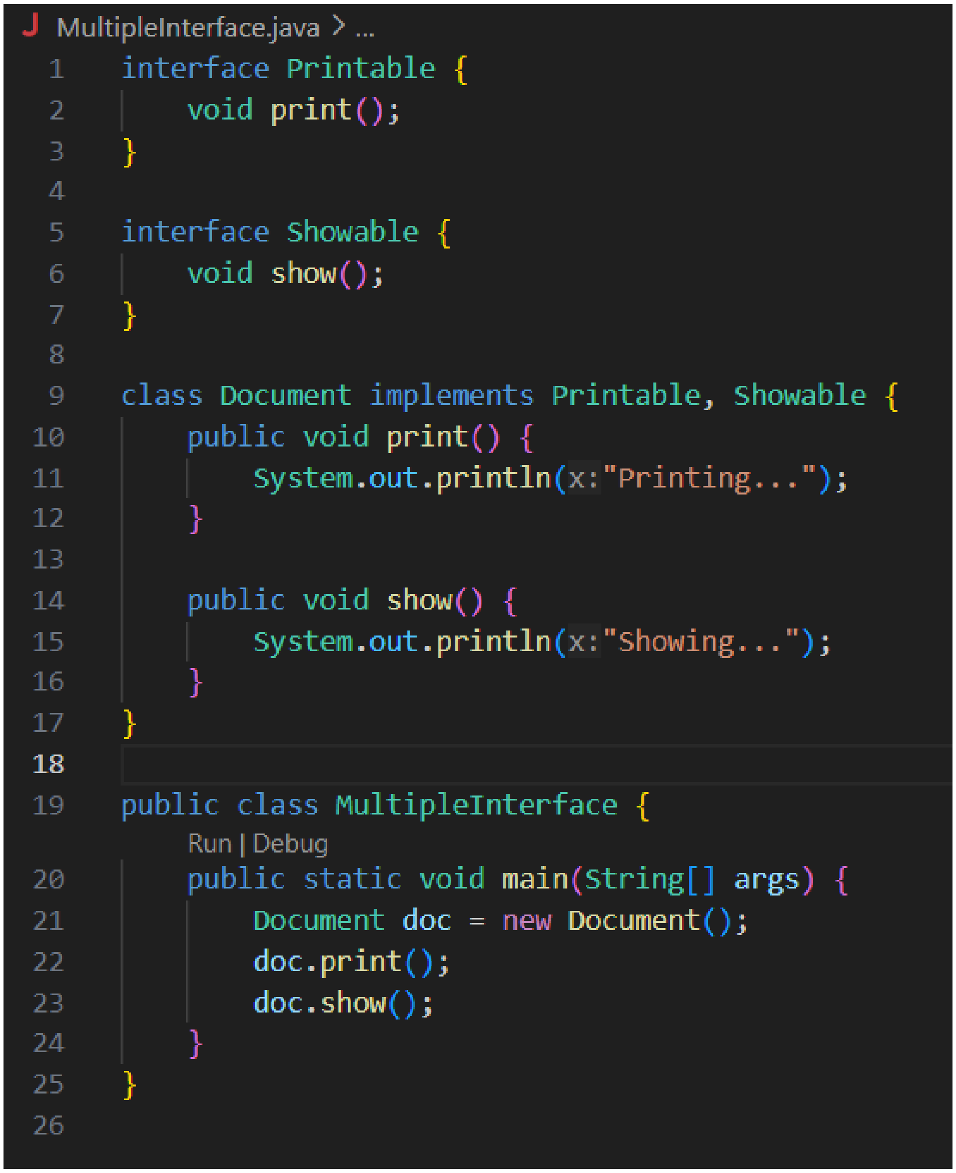
Create an abstract class with a constructor and extend it in a subclass with additional logic.


AbstractConstructor.java > AbstractConstructor > main(String[])
Run | Debug

```
abstract class Animal{
    Animal(){
        System.out.println(x:"Animal Created");
    }
    abstract void sound();
}

class Dog extends Animal{
    Dog(){
        System.out.println(x:"Dog created");
    }
    void sound(){
        System.out.println(x:"Bhow Bhow");
    }
}


public class AbstractConstructor {
    public static void main(String[] args){
        Dog doggy = new Dog();
        doggy.sound();
    }
}

```

13. Multiple Interfaces in One Class

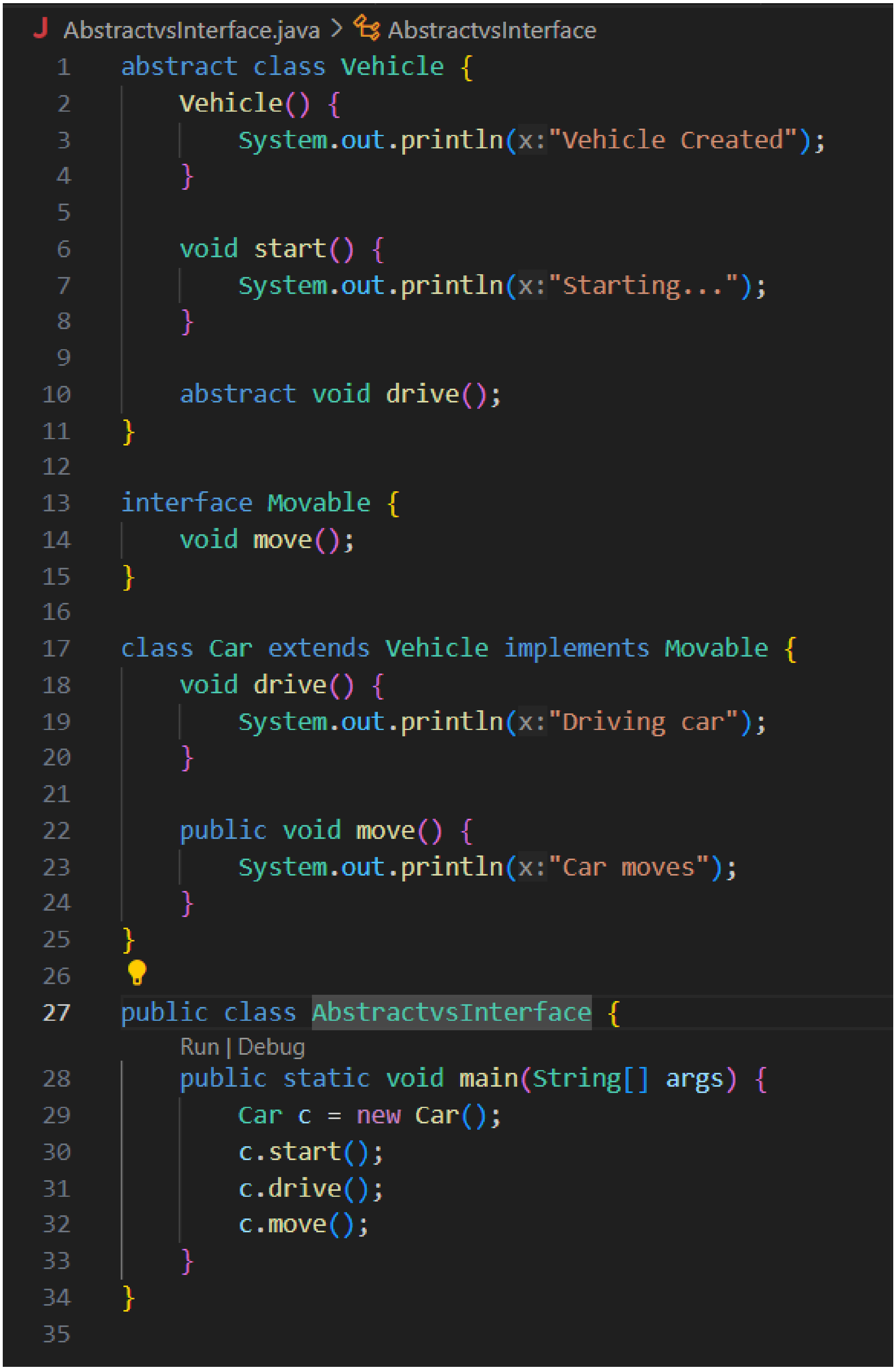
Implement two interfaces in a single class and invoke their methods.



```java
interface Printable {
    void print();
}

interface Showable {
    void show();
}

class Document implements Printable, Showable {
    public void print() {
        System.out.println(x:"Printing...");
    }

    public void show() {
        System.out.println(x:"Showing...");
    }
}

public class MultipleInterface {
    public static void main(String[] args) {
        Document doc = new Document();
        doc.print();
        doc.show();
    }
}
```

14. Compare Abstract Class and Interface

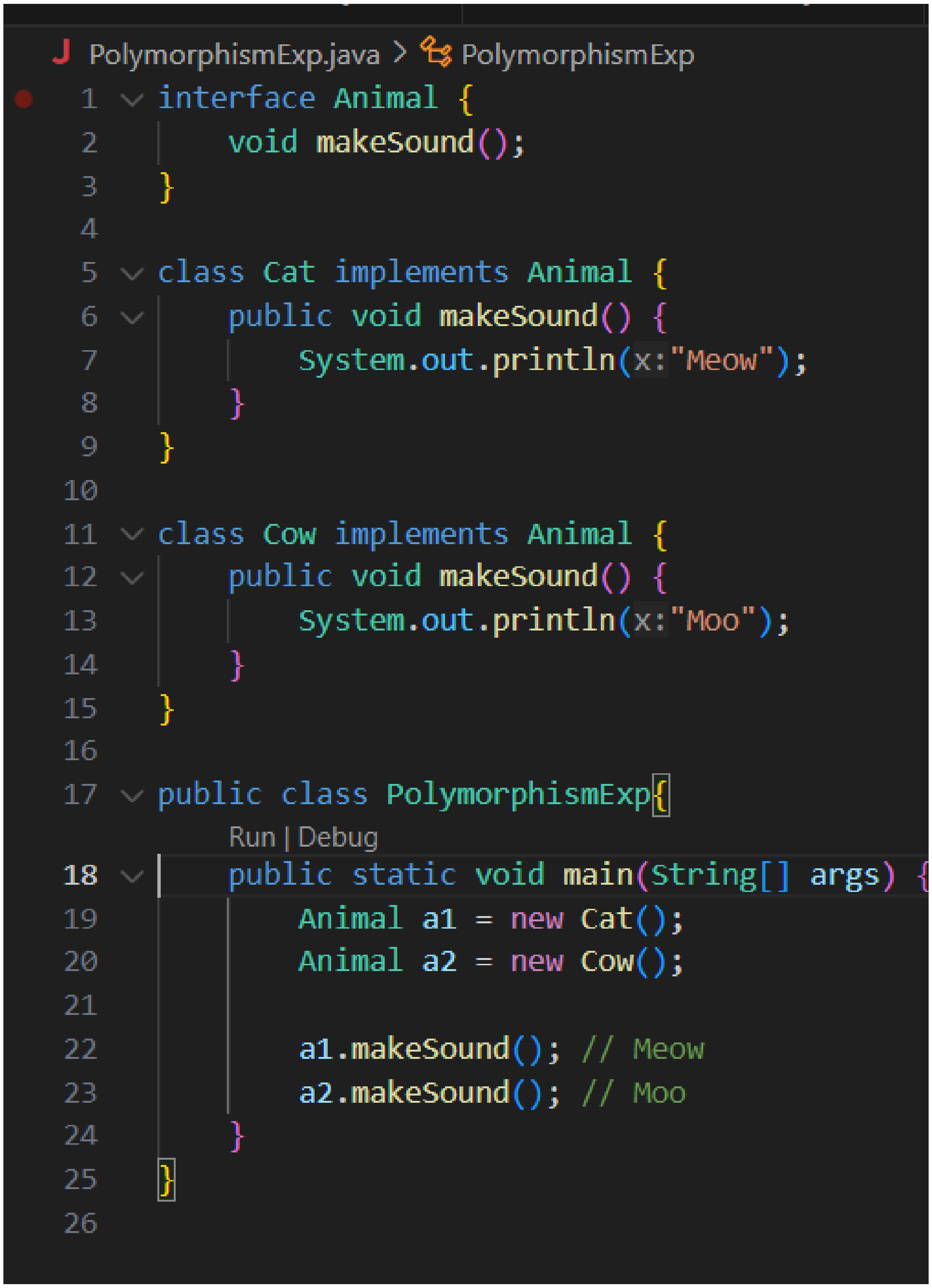
Create a program showing key differences in features and usage of both.



```java
abstract class Vehicle {
    Vehicle() {
        System.out.println(x:"Vehicle Created");
    }

    void start() {
        System.out.println(x:"Starting...");
    }

    abstract void drive();
}

interface Movable {
    void move();
}

class Car extends Vehicle implements Movable {
    void drive() {
        System.out.println(x:"Driving car");
    }

    public void move() {
        System.out.println(x:"Car moves");
    }
}

public class AbstractvsInterface {
    public static void main(String[] args) {
        Car c = new Car();
        c.start();
        c.drive();
        c.move();
    }
}
```

15. Use Interface for Polymorphism

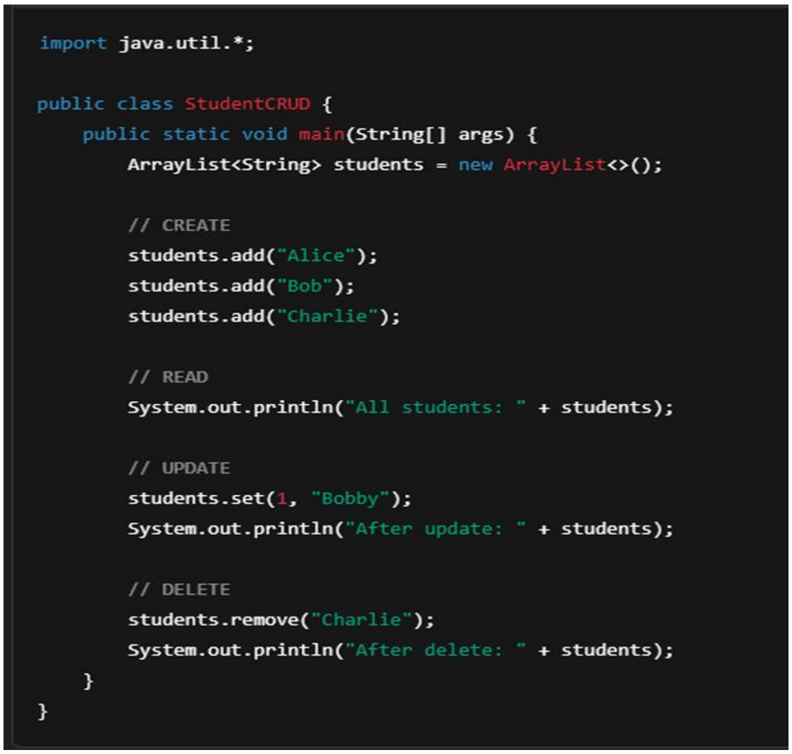
Demonstrate how different implementations of an interface can be used interchangeably.



```
interface Animal {
    void makeSound();
}

class Cat implements Animal {
    public void makeSound() {
        System.out.println(x:"Meow");
    }
}

class Cow implements Animal {
    public void makeSound() {
        System.out.println(x:"Moo");
    }
}

public class PolymorphismExp{
    Run | Debug
    public static void main(String[] args) {
        Animal a1 = new Cat();
        Animal a2 = new Cow();

        a1.makeSound(); // Meow
        a2.makeSound(); // Moo
    }
}

```

16. Perform CRUD using ArrayList

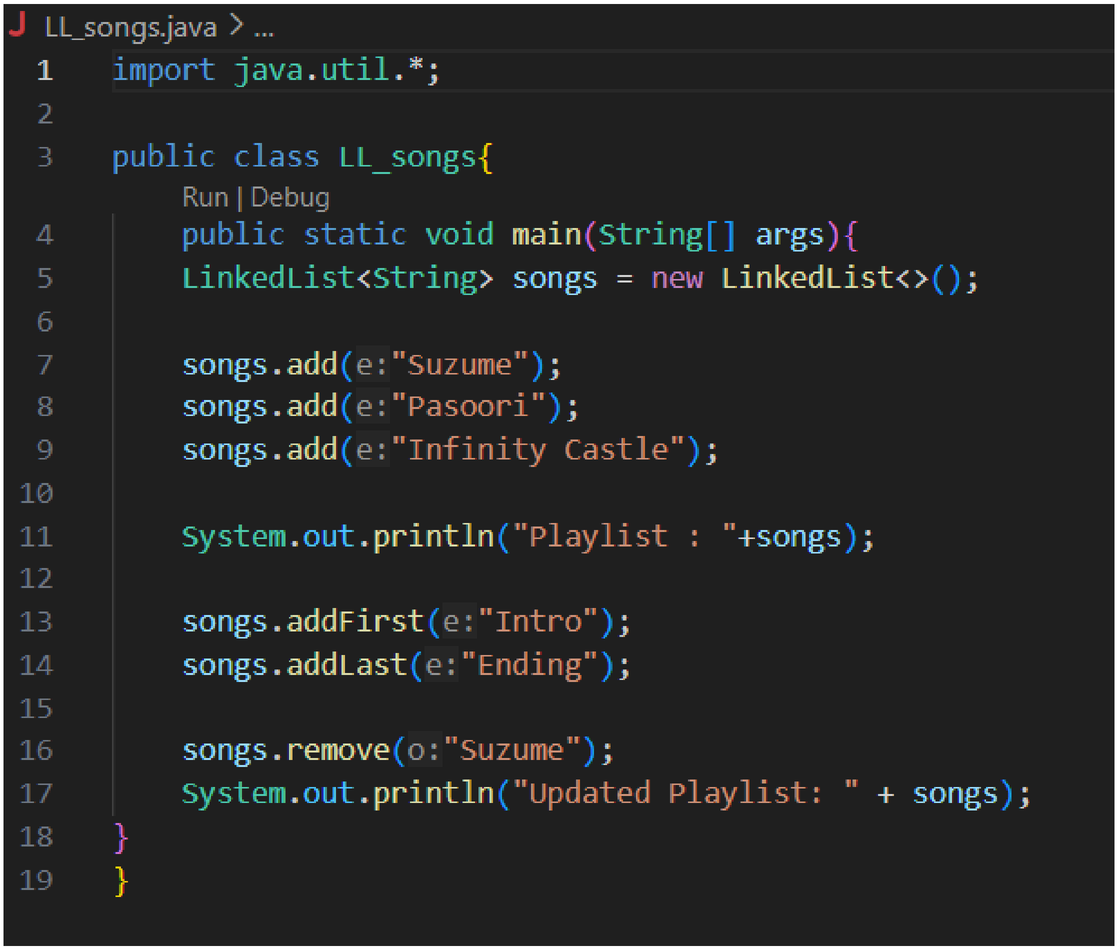
Add, retrieve, update, and remove student records using ArrayList.



```
import java.util.*;

public class StudentCRUD {
    public static void main(String[] args) {
        ArrayList<String> students = new ArrayList<>();

        // CREATE
        students.add("Alice");
        students.add("Bob");
        students.add("Charlie");

        // READ
        System.out.println("All students: " + students);

        // UPDATE
        students.set(1, "Bobby");
        System.out.println("After update: " + students);

        // DELETE
        students.remove("Charlie");
        System.out.println("After delete: " + students);
    }
}
```

17. LinkedList Example for Playlist

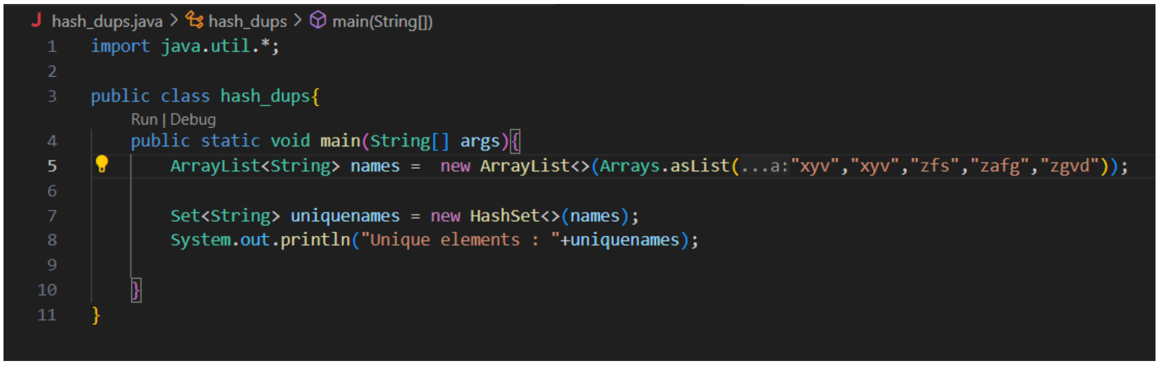
Manage songs in a playlist using LinkedList and show add/remove operations.



```java
import java.util.*;

public class LL_songs{
    Run | Debug
    public static void main(String[] args){
    LinkedList<String> songs = new LinkedList<>();

    songs.add(e:"Suzume");
    songs.add(e:"Pasoori");
    songs.add(e:"Infinity Castle");

    System.out.println("Playlist :  "+songs);

    songs.addFirst(e:"Intro");
    songs.addLast(e:"Ending");

    songs.remove(o:"Suzume");
    System.out.println("Updated Playlist: " + songs);
}
}
```

18. Remove Duplicates using HashSet

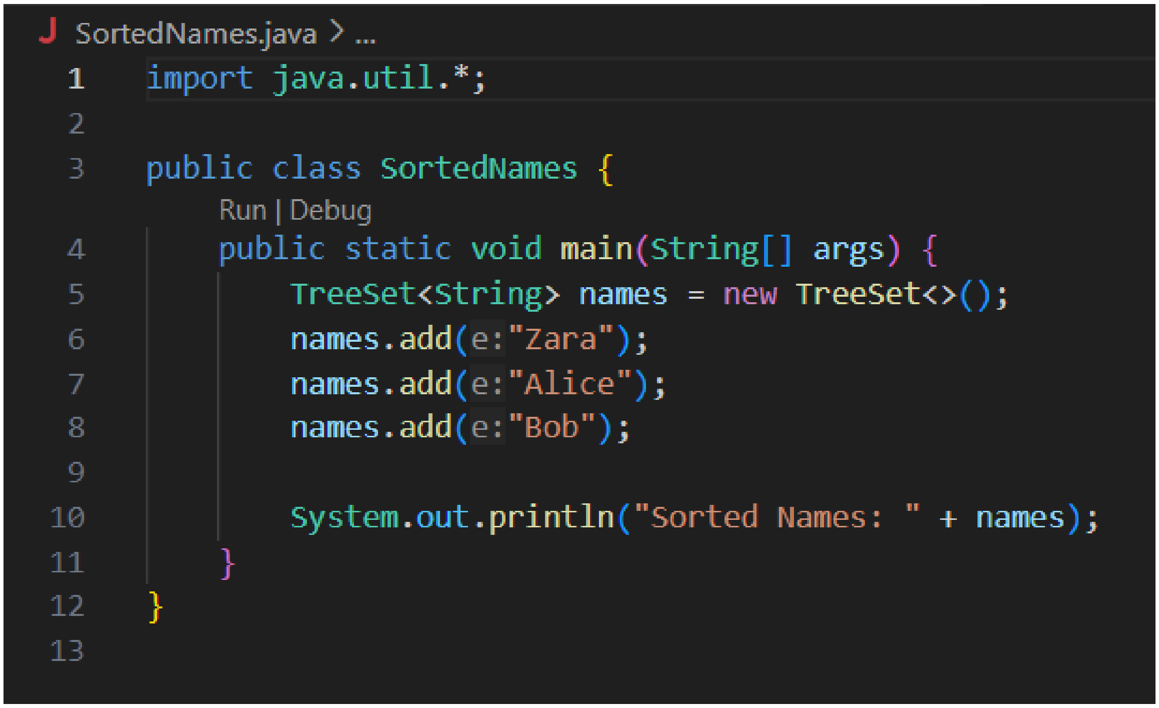
Input a list of names and store unique ones using HashSet.



```java
import java.util.*;

public class hash_dups{
    Run | Debug
    public static void main(String[] args){
        ArrayList<String> names =  new ArrayList<>(Arrays.asList(...a:"xyv","xyv","zfs","zafg","zgvd"));

        Set<String> uniquenames = new HashSet<>(names);
        System.out.println("Unique elements : "+uniquenames);

    }
}
```

19. Sort Data using TreeSet

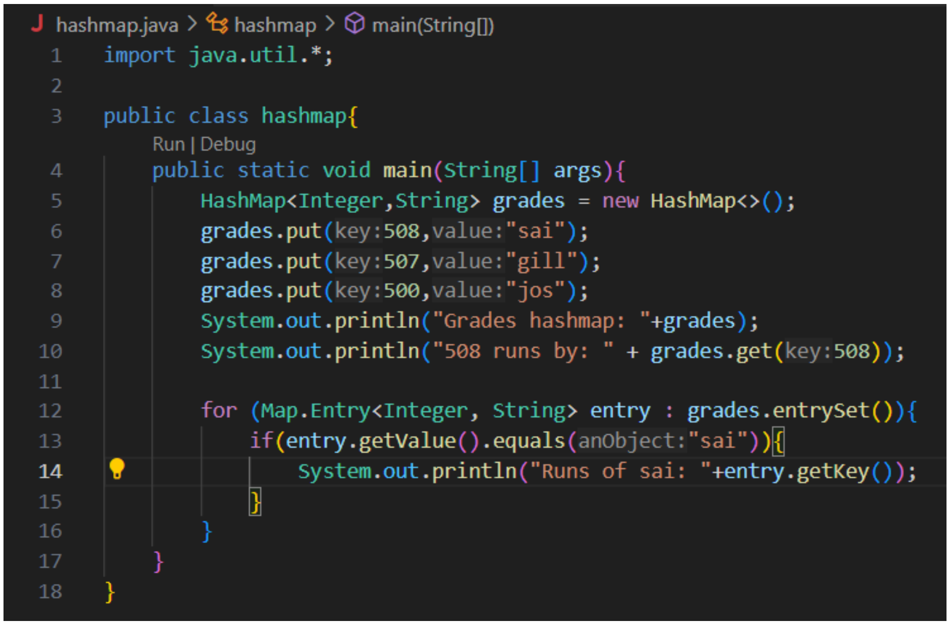
Insert names in TreeSet and show sorted order output.



```java
import java.util.*;

public class SortedNames {
    public static void main(String[] args) {
        TreeSet<String> names = new TreeSet<>();
        names.add(e:"Zara");
        names.add(e:"Alice");
        names.add(e:"Bob");

        System.out.println("Sorted Names: " + names);
    }
}

```

20. HashMap Example - Student Grades

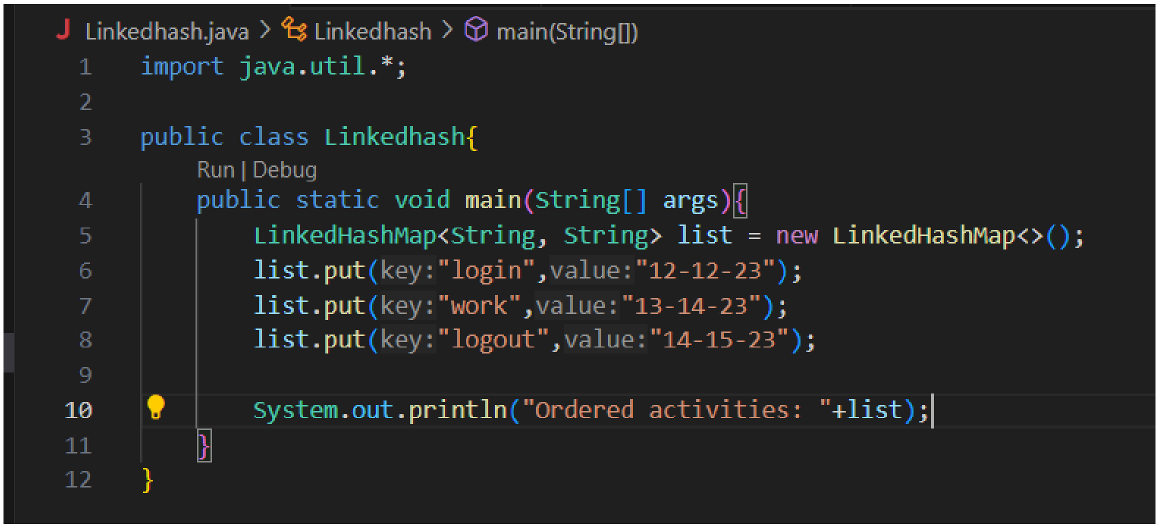
Store and retrieve students' grades using roll numbers as keys.



```java
import java.util.*;

public class hashmap{
    public static void main(String[] args){
        HashMap<Integer,String> grades = new HashMap<>();
        grades.put(key:508,value:"sai");
        grades.put(key:507,value:"gill");
        grades.put(key:500,value:"jos");
        System.out.println("Grades hashmap: "+grades);
        System.out.println("508 runs by: " + grades.get(key:508));

        for (Map.Entry<Integer, String> entry : grades.entrySet()){
            if(entry.getValue().equals(anObject:"sai")){
                System.out.println("Runs of sai: "+entry.getKey());
            }
        }
    }
}
```

21. LinkedHashMap for Recent Activities

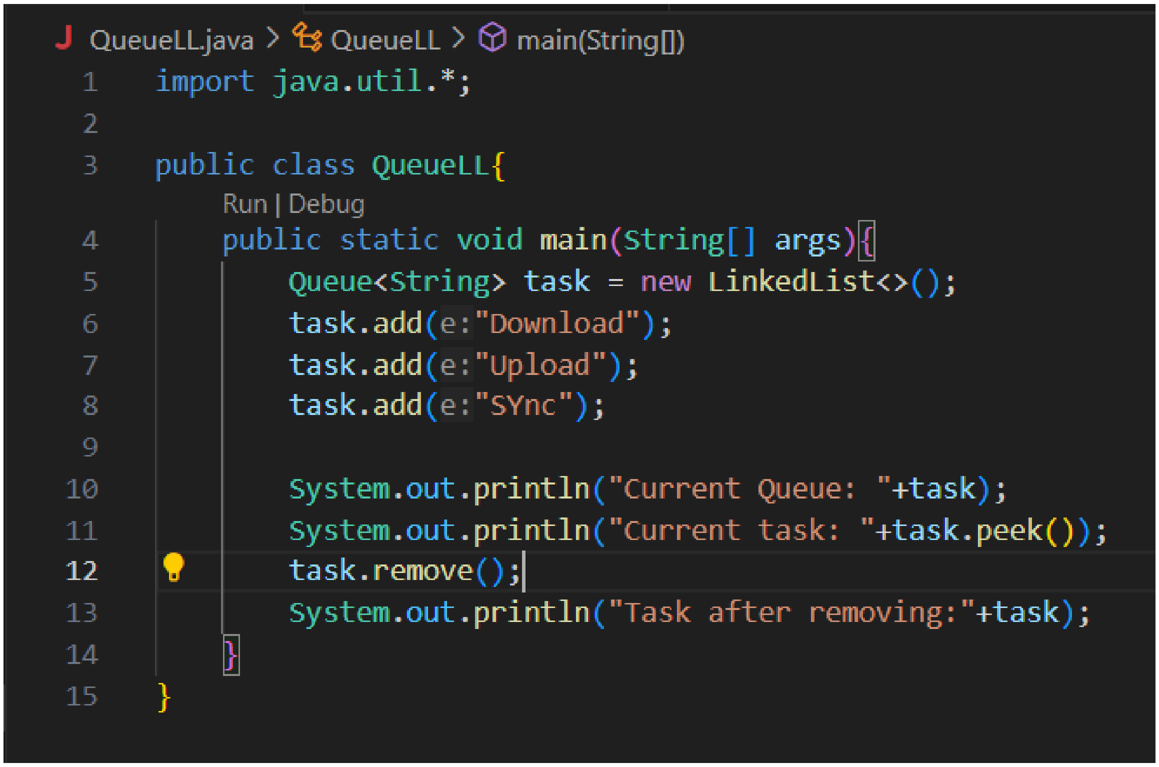
Record user activity timestamps while maintaining insertion order.



```java
import java.util.*;

public class Linkedhash{
    public static void main(String[] args){
        LinkedHashMap<String, String> list = new LinkedHashMap<>();
        list.put(key:"login",value:"12-12-23");
        list.put(key:"work",value:"13-14-23");
        list.put(key:"logout",value:"14-15-23");

        System.out.println("Ordered activities: "+list);
    }
}
```

22. Implement Queue with LinkedList

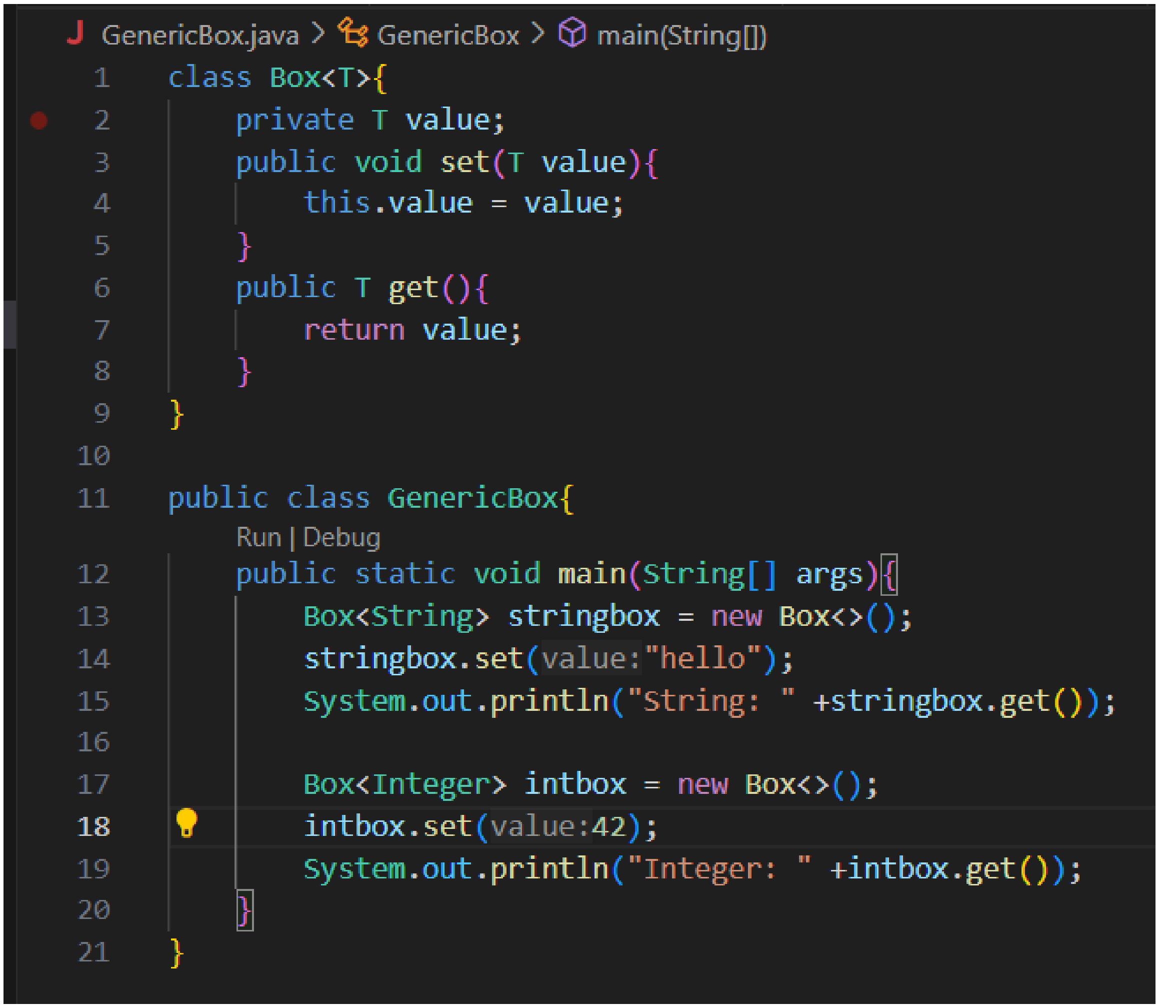
Simulate a task queue with enqueue, dequeue operations using LinkedList.



```java
import java.util.*;

public class QueueLL{
    public static void main(String[] args){
        Queue<String> task = new LinkedList<>();
        task.add(e:"Download");
        task.add(e:"Upload");
        task.add(e:"SYnc");

        System.out.println("Current Queue: "+task);
        System.out.println("Current task: "+task.peek());
        task.remove();
        System.out.println("Task after removing:"+task);
    }
}
```

23. Create a Generic Box Class

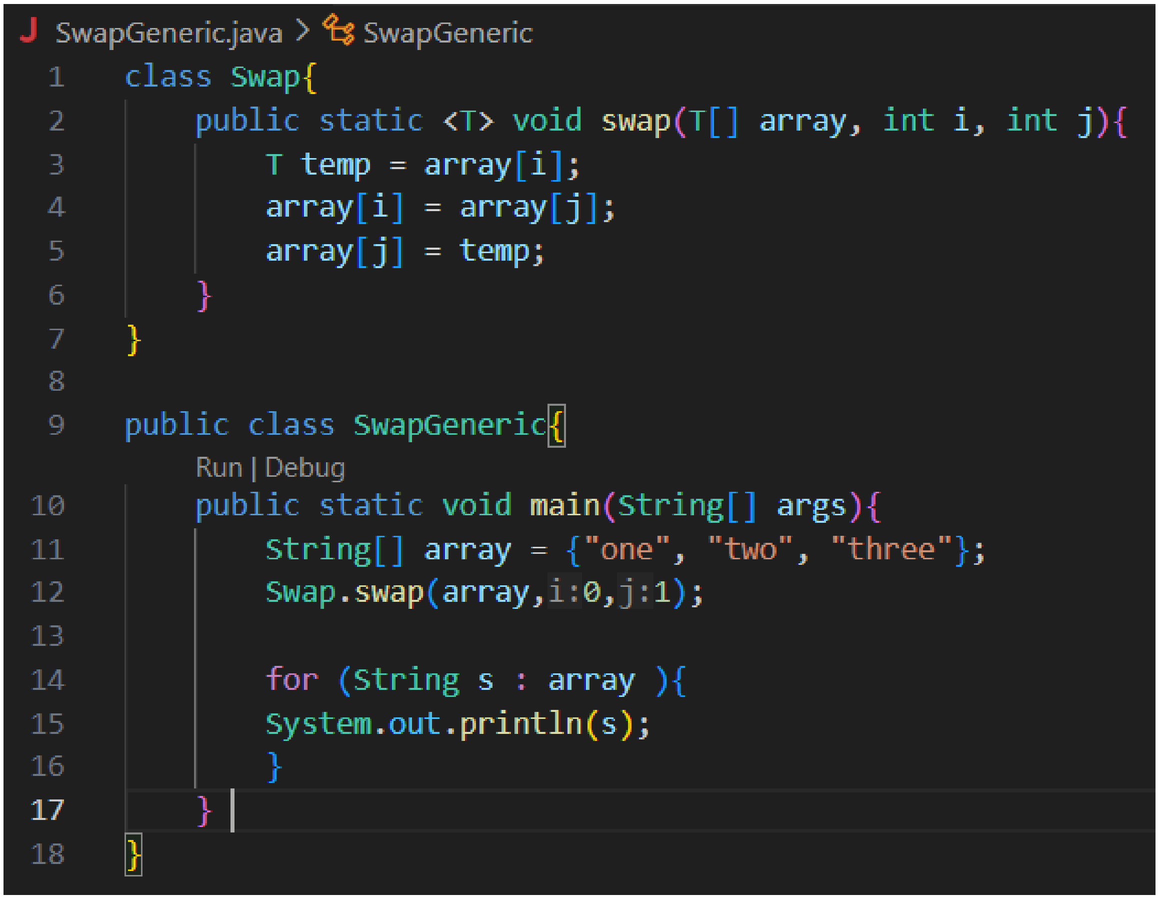
Create a class that stores objects of any type and prints the content.



```java
class Box<T>{
    private T value;
    public void set(T value){
        this.value = value;
    }
    public T get(){
        return value;
    }
}

public class GenericBox{
    public static void main(String[] args){
        Box<String> stringbox = new Box<>();
        stringbox.set(value:"hello");
        System.out.println("String: " +stringbox.get());

        Box<Integer> intbox = new Box<>();
        intbox.set(value:42);
        System.out.println("Integer: " +intbox.get());
    }
}
```

24. Write a Generic Swap Method

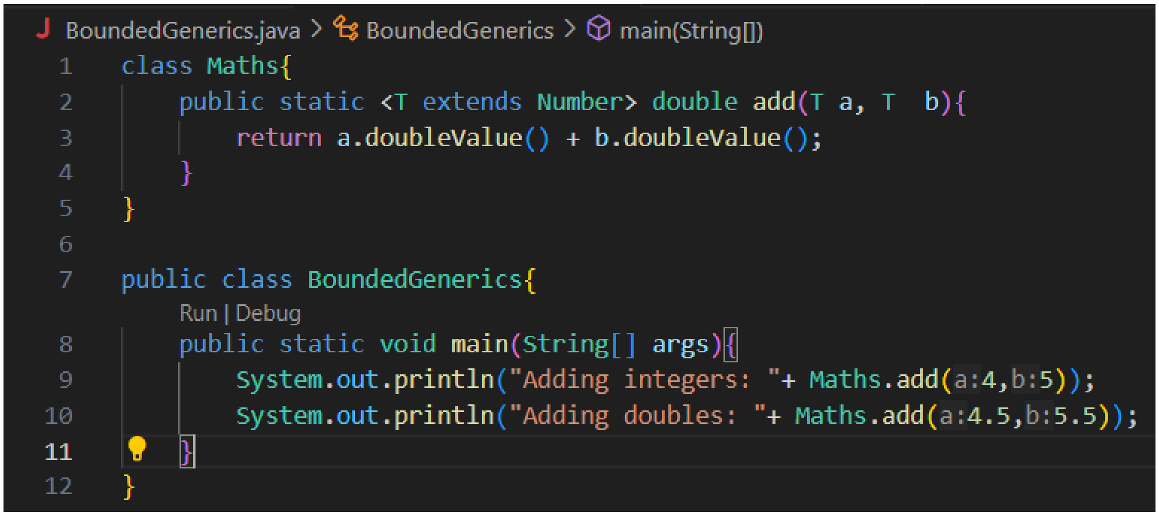
Create a method that swaps two elements of any type (e.g., integers, strings).



```java
class Swap{
    public static <T> void swap(T[] array, int i, int j){
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}

public class SwapGeneric{
    public static void main(String[] args){
        String[] array = {"one", "two", "three"};
        Swap.swap(array,i:0,j:1);

        for (String s : array ){
        System.out.println(s);
        }
    }
}
```

25. Bounded Generics Example

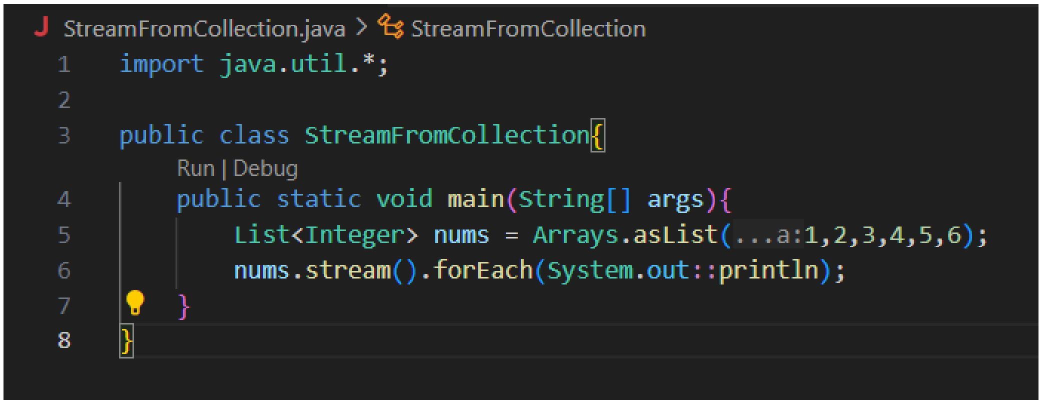
Create a generic method to print numeric values only using bounded type parameters.



```java
class Maths{
    public static <T extends Number> double add(T a, T  b){
        return a.doubleValue() + b.doubleValue();
    }
}

public class BoundedGenerics{
    public static void main(String[] args){
        System.out.println("Adding integers: "+ Maths.add(a:4,b:5));
        System.out.println("Adding doubles: "+ Maths.add(a:4.5,b:5.5));
    }
}
```

26. Stream from Collection

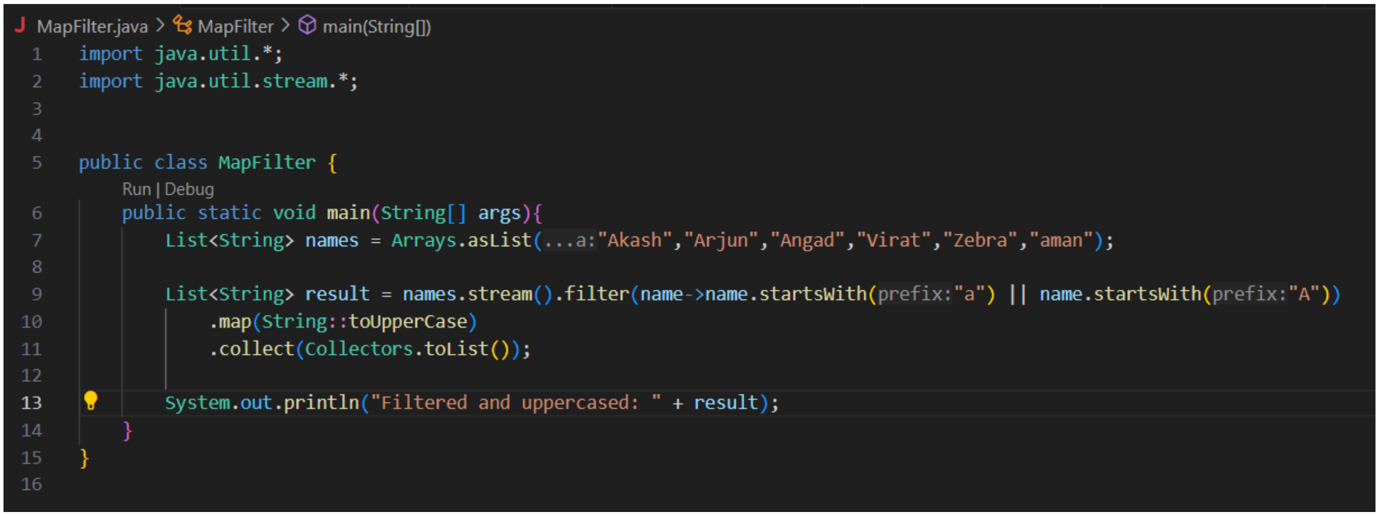
Convert a list of integers to stream and print all elements.



```java
import java.util.*;

public class StreamFromCollection{
    public static void main(String[] args){
        List<Integer> nums = Arrays.asList(...a:1,2,3,4,5,6);
        nums.stream().forEach(System.out::println);
    }
}
```

27. Map and Filter Stream Operations

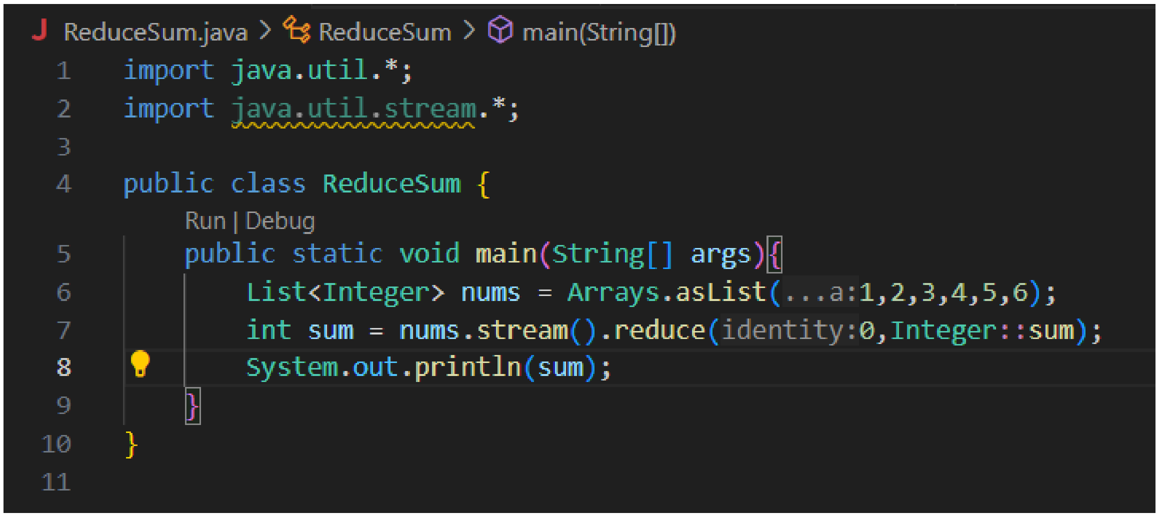
Given a list of names, filter names starting with 'A' and convert them to uppercase.



```java
import java.util.*;
import java.util.stream.*;


public class MapFilter {
    public static void main(String[] args){
        List<String> names = Arrays.asList(...a:"Akash","Arjun","Angad","Virat","Zebra","aman");

        List<String> result = names.stream().filter(name->name.startsWith(prefix:"a") || name.startsWith(prefix:"A"))
            .map(String::toUpperCase)
            .collect(Collectors.toList());

        System.out.println("Filtered and uppercased: " + result);
    }
}

```

28. Use Reduce for Sum

Use reduce() to calculate the sum of a list of integers.

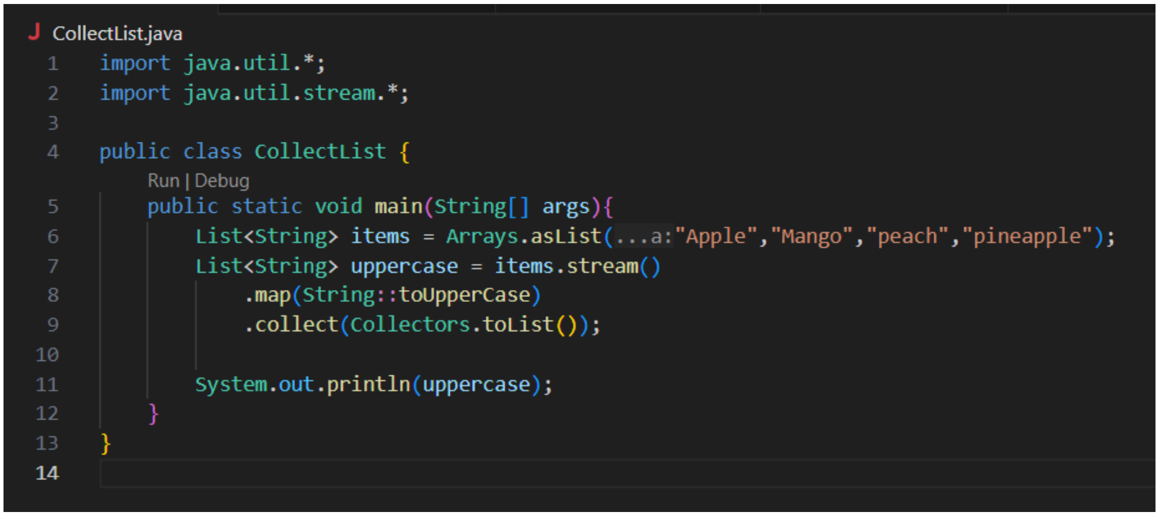


ReduceSum.java > ReduceSum > main(String[])

```
import java.util.*;
import java.util.stream.*;

public class ReduceSum {
    Run | Debug
    public static void main(String[] args){
        List<Integer> nums = Arrays.asList(...a:1,2,3,4,5,6);
        int sum = nums.stream().reduce(identity:0,Integer::sum);
        System.out.println(sum);
    }
}

```

29. Collect Stream to List

Convert a list of strings into a stream, modify, and collect it back to list.

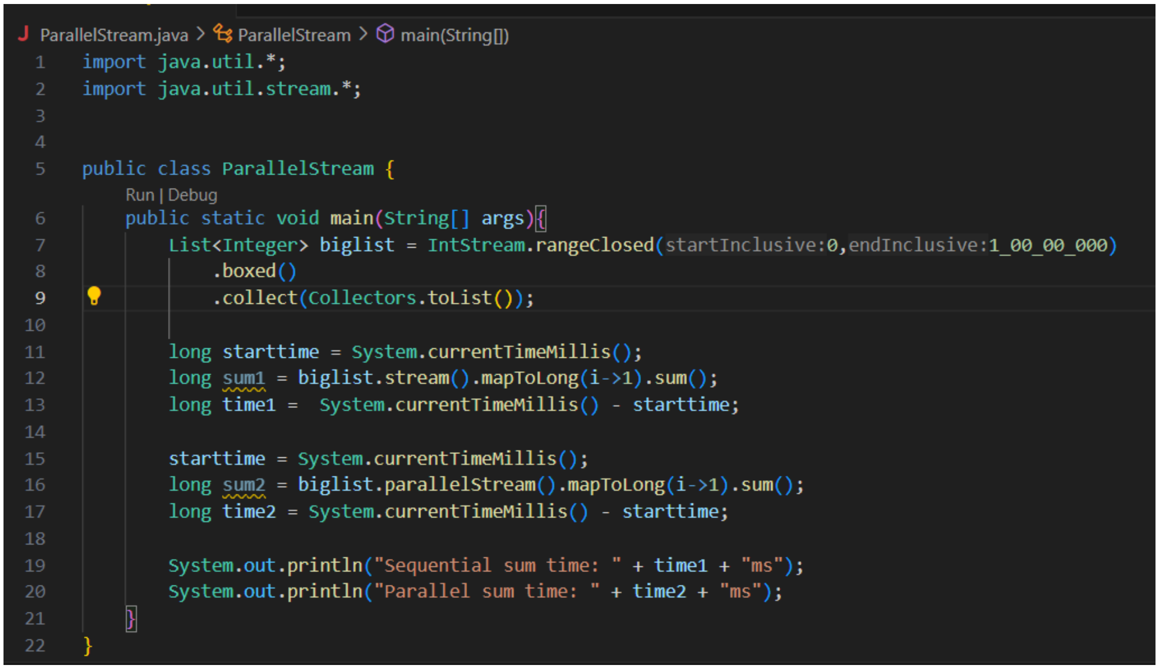


CollectList.java

```
import java.util.*;
import java.util.stream.*;

public class CollectList {
    Run | Debug
    public static void main(String[] args){
        List<String> items = Arrays.asList(...a:"Apple","Mango","peach","pineapple");
        List<String> uppercase = items.stream()
            .map(String::toUpperCase)
            .collect(Collectors.toList());

        System.out.println(uppercase);
    }
}

```

30. Parallel Stream Usage

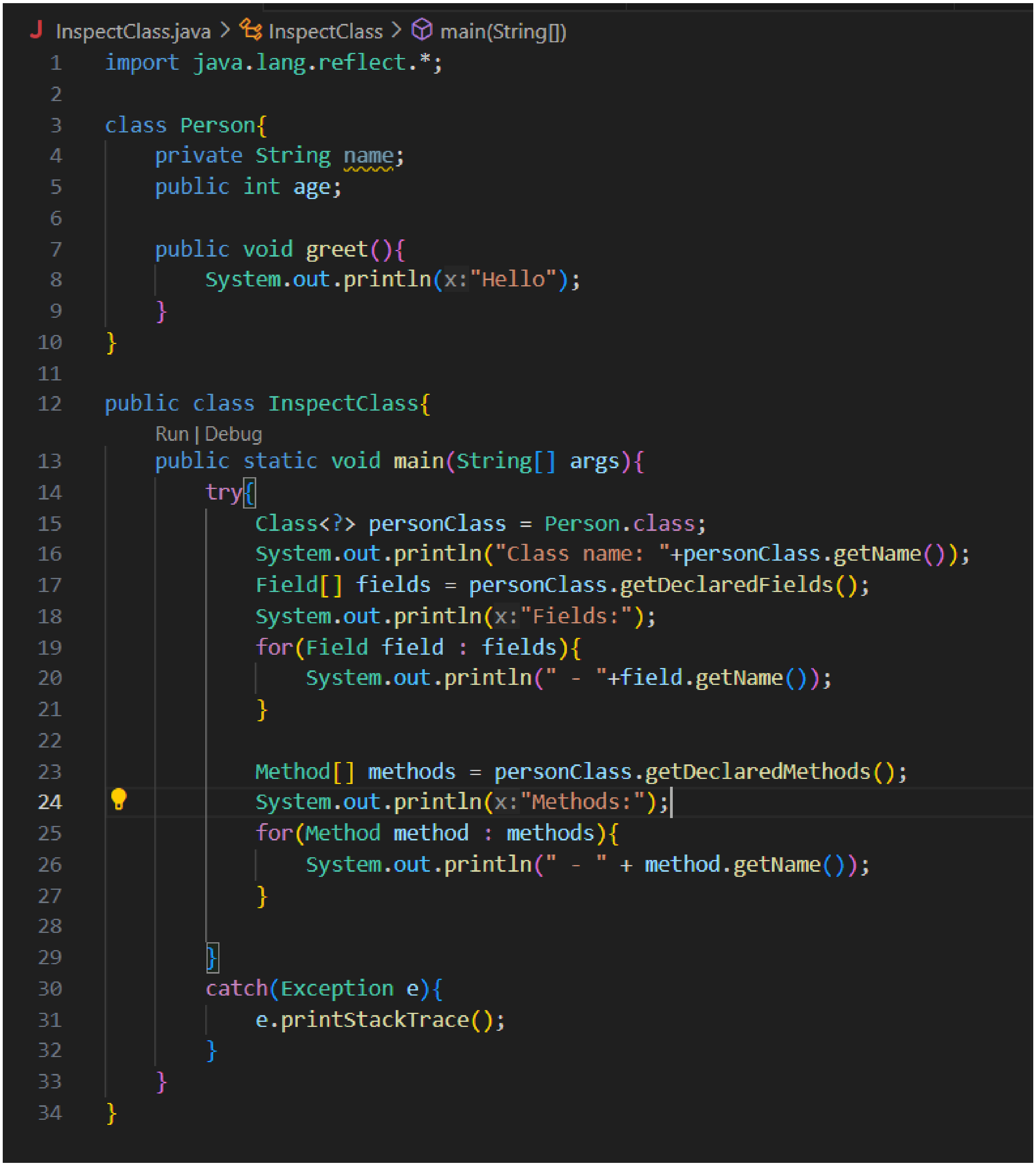
Use parallelStream() to process a large dataset and compare time taken with normal stream.



```java
import java.util.*;
import java.util.stream.*;


public class ParallelStream {
    public static void main(String[] args){
        List<Integer> biglist = IntStream.rangeClosed(startInclusive:0,endInclusive:1_00_00_000)
            .boxed()
            .collect(Collectors.toList());

        long starttime = System.currentTimeMillis();
        long sum1 = biglist.stream().mapToLong(i->1).sum();
        long time1 =  System.currentTimeMillis() - starttime;

        starttime = System.currentTimeMillis();
        long sum2 = biglist.parallelStream().mapToLong(i->1).sum();
        long time2 = System.currentTimeMillis() - starttime;

        System.out.println("Sequential sum time: " + time1 + "ms");
        System.out.println("Parallel sum time: " + time2 + "ms");
    }
}
```

31. Inspect Class using Reflection

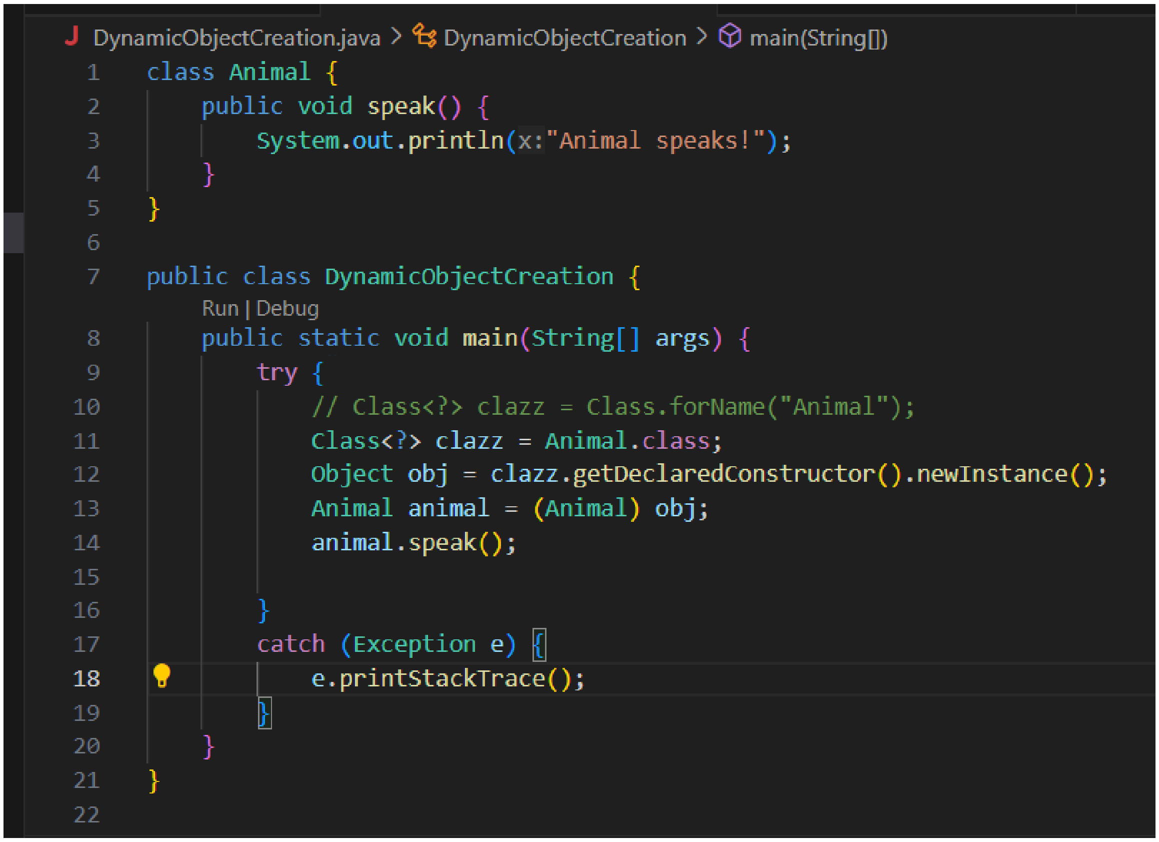
Write a program to get class name, fields, and method names using reflection.



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

class Person{
    private String name;
    public int age;

    public void greet(){
        System.out.println(x:"Hello");
    }
}

public class InspectClass{
    public static void main(String[] args){
        try{
            Class<?> personClass = Person.class;
            System.out.println("Class name: "+personClass.getName());
            Field[] fields = personClass.getDeclaredFields();
            System.out.println(x:"Fields:");
            for(Field field : fields){
                System.out.println(" - "+field.getName());
            }

            Method[] methods = personClass.getDeclaredMethods();
            System.out.println(x:"Methods:");
            for(Method method : methods){
                System.out.println(" - " + method.getName());
            }

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

32. Dynamic Object Creation using Reflection

Create an object of a class using Class.forName() and newInstance().

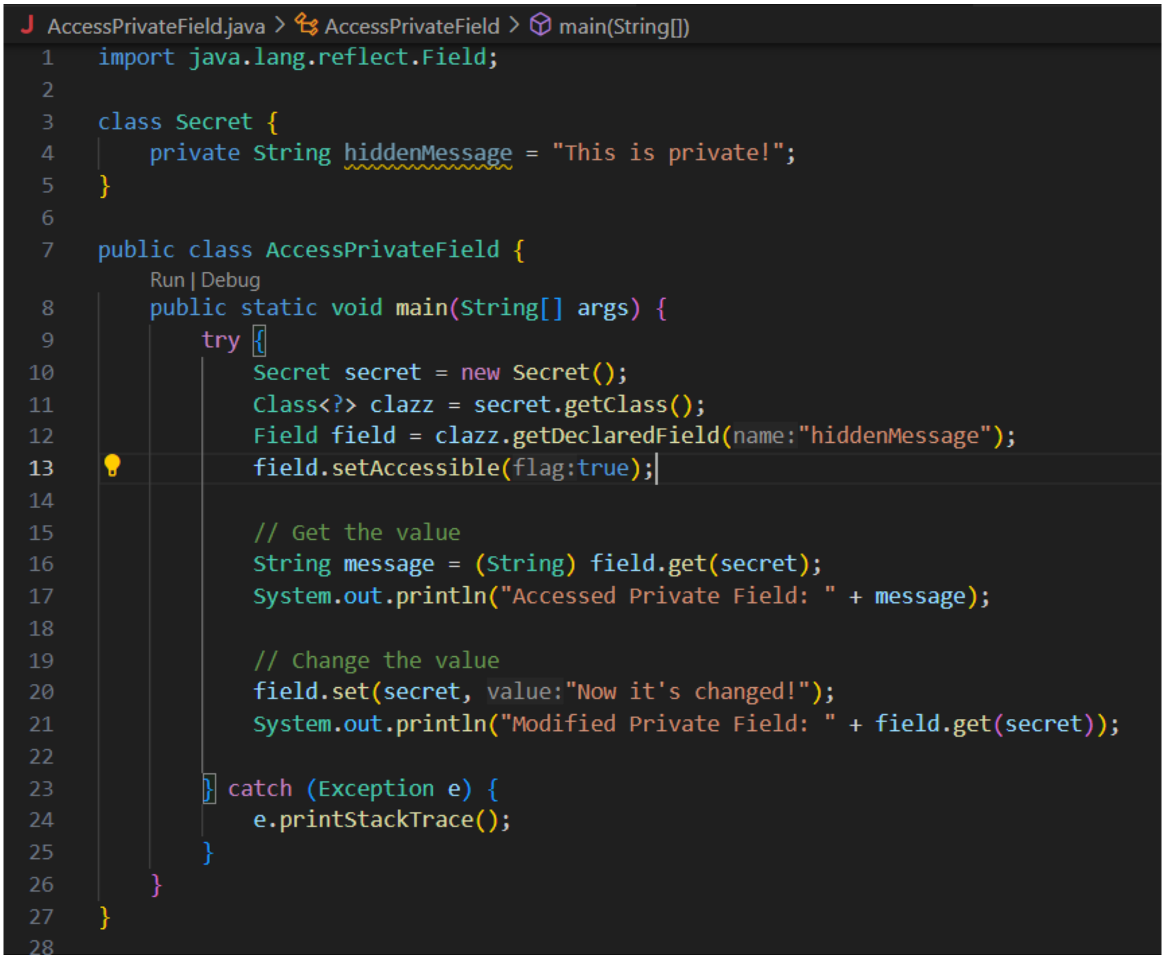


DynamicObjectCreation.java > DynamicObjectCreation > main(String[])
Run | Debug

```
class Animal {
    public void speak() {
        System.out.println(x:"Animal speaks!");
    }
}

public class DynamicObjectCreation {
    public static void main(String[] args) {
        try {
            // Class<?> clazz = Class.forName("Animal");
            Class<?> clazz = Animal.class;
            Object obj = clazz.getDeclaredConstructor().newInstance();
            Animal animal = (Animal) obj;
            animal.speak();

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

```

33. Access Private Field with Reflection

Use reflection to modify and access a private field of a class.


AccessPrivateField.java > AccessPrivateField > main(String[])
Run | Debug

```java
import java.lang.reflect.Field;

class Secret {
    private String hiddenMessage = "This is private!";
}

public class AccessPrivateField {
    public static void main(String[] args) {
        try {
            Secret secret = new Secret();
            Class<?> clazz = secret.getClass();
            Field field = clazz.getDeclaredField(name:"hiddenMessage");
            field.setAccessible(flag:true);

            // Get the value
            String message = (String) field.get(secret);
            System.out.println("Accessed Private Field: " + message);

            // Change the value
            field.set(secret, value:"Now it's changed!");
            System.out.println("Modified Private Field: " + field.get(secret));

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