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.

```
//singleton class means only one object of this class can exist
//if other object is instantiated, it refer to the original one
class Vehicle{
    private static Vehicle instance;
    private Vehicle(){}  // the constructor of singleton class is always private so that user cannot create objects directly using the 'new' keyword
    public static Vehicle getInstance(){
        if(instance == null){   // incase the object is not made before
            instance = new Vehicle();
        }
        return instance;
    }
}
public class car{
    public static void main(String[] args){
        Vehicle myCar = Vehicle.getInstance();
        Vehicle yourCar = Vehicle.getInstance();
        System.out.println(myCar == yourCar);
    }
}
```

2. Implement Factory Design Pattern

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

```
interface Shape{
    void draw();
}
class Circle implements Shape{
    public void draw(){
        System.out.println("Drawing Circle");
    }
}
class Rectangle implements Shape{
    public void draw(){
        System.out.println("Drawing Rectangle");
    }
}
class ShapeMaker{
    public static Shape createShape(String type){
        if(type.equalsIgnoreCase("circle")){
            return new Circle();
        }else if(type.equalsIgnoreCase("rectangle")){
            return new Rectangle();
        }
        return null;
    }
}
public class shapeFactory{
    public static void main(String[] args) {
        Shape shape1 = ShapeMaker.createShape("CIRCLE");
        Shape shape2 = ShapeMaker.createShape("RECTANGLE");
        shape1.draw();
        shape2.draw();
    }
}
```

3. Implement Observer Design Pattern

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

```
interface PaymentMethod{
    void pay(double amt);
}
class CreditCardPayment implements PaymentMethod{
    public void pay(double amt){
        System.out.println("Paid $ " + amt + " using Credit Card");
    }
}
class DeditCardPayment implements PaymentMethod{
    public void pay(double amt){
        System.out.println("Paid $ " + amt + " using Dedit Card");
    }
}
class UPI implements PaymentMethod{
    public void pay(double amt){
        System.out.println("Paid $ " + amt + " using UPI");
    }
}
class PaymentProcessor{
    private PaymentMethod method;
    public PaymentProcessor(PaymentMethod method){
        this.method = method;
    }
    public void processPayment(double amt){
        method.pay(amt);
    }
}
public class ocp_paymentApp{
    public static void main(String[] args){
        PaymentProcessor processor1 = new PaymentProcessor(new CreditCardPayment());
        processor1.processPayment(500.0);
        PaymentProcessor processor2 = new PaymentProcessor(new DeditCardPayment());
        processor2.processPayment(230.0);
    }
}
```

4. Implement Strategy Design Pattern

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

```
import java.util.*;
interface SortStrategy{
    void sort(int[] numbers);
}
class BubbleSort implements SortStrategy{
    public void sort(int[] numbers){
```

```
        for(int i=0; i<numbers.length - 1; i++){
            for(int j=0; j<numbers.length - i - 1; j++){
                if(numbers[j + 1] < numbers[j]){
                    int temp = numbers[j];
                    numbers[j] = numbers[j+1];
                    numbers[j+1] = temp;
                }
            }
        }
        for(int i=0; i<numbers.length; i++){
            System.out.print(numbers[i] + " ");
        }
        System.out.println("");
    }
}
class QuickSort implements SortStrategy{
    public void sort(int[] numbers){
        quickSort(numbers, 0, numbers.length - 1);
        for(int i=0; i<numbers.length; i++){
            System.out.print(numbers[i] + " ");
        }
        System.out.println("");
    }
    private void quickSort(int[] numbers, int low, int high){
        if(low < high){
            int pivot = partition(numbers, low, high);
            quickSort(numbers, low, pivot - 1);
            quickSort(numbers, pivot + 1, high);
        }
    }
    private int partition(int[] numbers, int low, int high){
        int pivot = numbers[high];
        int i = low - 1;
        for(int j = low; j<high; j++){
            if(numbers[j] < pivot){
                i++;
                int temp = numbers[i];
                numbers[i] = numbers[j];
                numbers[j] = temp;
            }
        }
        int temp = numbers[i + 1];
        numbers[i + 1] = numbers[high];
        numbers[high] = temp;
        return i + 1;
    }
}
class SortingStrategy{
    private SortStrategy strat;
    public void setStrategy(SortStrategy strat){
        this.strat = strat;
    }
    public void executeSort(int[] numbers){
        strat.sort(numbers);
    }
}
public class strategy{
    public static void main(String[] args) {
        int[] numbers = {2, 3, 5, 234, 65, 765};
        int[] arr1 = Arrays.copyOf(numbers, numbers.length);
        int[] arr2 = Arrays.copyOf(numbers, numbers.length);
        SortingStrategy context = new SortingStrategy();
        context.setStrategy(new BubbleSort());
        context.executeSort(arr1);
        context.setStrategy(new QuickSort());
        context.executeSort(arr2);
    }
}
```

5. Apply SOLID Principles - Case Study

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

```
import java.util.*;
class Book{
    private String title;
    private boolean isBorrowed;
    public Book(String title){
        this.title = title;
        this.isBorrowed = false;
    }
    public String getTitle(){
        return title;
    }
    public boolean checkIfBorrowed(){
        return isBorrowed;
    }
    public void borrow(){
        isBorrowed = true;
    }
    public void returnBook(){
        isBorrowed = false;
    }
}
interface BorrowPolicy{
    boolean canBorrow(Book book);
}
class SimpleBorrowPolicy implements BorrowPolicy{
    public boolean canBorrow(Book book){
        return !book.checkIfBorrowed();
    }
}
class LibraryManager{
    private BorrowPolicy borrowPolicy;
```

```
    public LibraryManager(BorrowPolicy policy){
        this.borrowPolicy = policy;
    }
    public void borrowBook(Book book){
        if(borrowPolicy.canBorrow(book)){
            book.borrow();
            System.out.println("You borrowed "+ book.getTitle());
        }else{
            System.out.println("Already borrowed!");
        }
    }
    public void returnBook(Book book){
        book.returnBook();
        System.err.println("You returned " + book.getTitle());
    }
}
public class librarySystem{
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        List<Book> books = new ArrayList<>();
        books.add(new Book("1984"));
        books.add(new Book("Crime and Punishment"));
        BorrowPolicy policy = new SimpleBorrowPolicy();
        LibraryManager manager = new LibraryManager(policy);
        System.err.println("Library System");
        while (true) {
            System.out.print("\n1.Borrow Book\n2.Return Book\nElse exit\n");
            int choice = scanner.nextInt();
            System.err.print("\nEnter Book Number: ");
            int bookInd = scanner.nextInt();
            if(bookInd > 2 || bookInd < 0){
                break;
            }
            Book selectedBook = books.get(bookInd);
            if(choice == 1){
                manager.borrowBook(selectedBook);
            }else if(choice == 2){
                manager.returnBook(selectedBook);
            }else{
                break;
            }
        }
        scanner.close();
    }
}
```

6. Apply Interface Segregation Principle

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

```
interface Printer{
    void print_text(String doc);
}
interface ScannerDevice{
    void scan(String doc);
}
interface Fax{
    void fax(String doc);
}
class basicprinter implements Printer{
    public void print_text(String doc){
        System.out.println("Printing: " + doc);
    }
}
class SimpleScanner implements ScannerDevice{
    public void scan(String doc){
        System.out.println("Scanning: " + doc);
    }
}
class AllMachines implements Printer, ScannerDevice, Fax{
    public void print_text(String doc){
        System.out.println("All-in-one Print: " + doc);
    }
    public void scan(String doc){
        System.out.println("All-in-one Scan: " + doc);
    }
    public void fax(String doc){
        System.out.println("All-in-one Fax: " + doc);
    }
}
public class officeDevice{
    public static void main(String[] args){
        Printer printer = new basicprinter();
        printer.print_text("JAVA_EXP.pdf");
        ScannerDevice scanner = new SimpleScanner();
        System.out.println();
        scanner.scan("NLTP_EXP.pdf");

        AllMachines allMachines = new AllMachines();
        allMachines.print_text("IPCV.docx");
        allMachines.fax("faxxx.pdf");
    }
}
```

7. Apply Dependency Inversion Principle

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

```
//dependency inversion principle
interface NotificationSender{
    void send(String message);
}
class emailSender implements NotificationSender{
    public void send(String message){
        System.out.println("Email sent: " + message);
    }
```

```
}
class SMSSender implements NotificationSender{
    public void send(String message){
        System.out.println("SMS sent: " + message);
    }
}
class NotificationService{
    private NotificationSender sender;
    //constructor injection (DIP)
    public NotificationService(NotificationSender sender){
        this.sender = sender;
    }
    public void notifyUser(String message){
        sender.send(message);
    }
}
public class notificationApp{
    public static void main(String[] args){
        NotificationSender emailSender = new emailSender();
        NotificationService emailService = new NotificationService(emailSender);
        emailService.notifyUser("Your order is accepted!");

        NotificationSender smsSender = new SMSSender();
        NotificationService smsService = new NotificationService(smsSender);
        smsService.notifyUser("OTP is 9492");
    }
}
```

8. Apply Liskov Substitution Principle

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

```
// Liskov Substitution Principle -> Objects of a superclass should be replaceable with objects of its subclasses
interface Shape{
    int getArea();
}
class Rectangle implements Shape{
    protected int width;
    protected int height;
    public Rectangle(int width, int height){
        this.width = width;
        this.height = height;
    }
    public void setWidth(int w){
        this.width = w;
    }
    public void setHeight(int h){
        this.height = h;
    }
    public int getArea(){
        return width * height;
    }
}
class square implements Shape{
    private int side;
    public square(int side){
        this.side = side;
    }
    public void setSide(int side){
        this.side = side;
    }
    public int getArea(){
        return side * side;
    }
}
public class shape_demo_lsp{
    public static void printArea(Shape shape){
        System.out.println(shape.getArea());
    }
    public static void main(String[] args) {
        Shape rect = new Rectangle(5, 3);
        Shape square = new square(6);
        printArea(rect);
        printArea(square);
    }
}
```

9. Apply Open/Closed Principle

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

```
interface PaymentMethod{
    void pay(double amt);
}
class CreditCardPayment implements PaymentMethod{
    public void pay(double amt){
        System.out.println("Paid $ " + amt + " using Credit Card");
    }
}
class DeditCardPayment implements PaymentMethod{
    public void pay(double amt){
        System.out.println("Paid $ " + amt + " using Dedit Card");
    }
}
class UPI implements PaymentMethod{
    public void pay(double amt){
        System.out.println("Paid $ " + amt + " using UPI");
    }
}
class PaymentProcessor{
    private PaymentMethod method;
    public PaymentProcessor(PaymentMethod method){
        this.method = method;
    }
    public void processPayment(double amt){
        method.pay(amt);
```

```
    }
}
public class ocp_paymentApp{
    public static void main(String[] args){
        PaymentProcessor processor1 = new PaymentProcessor(new CreditCardPayment());
        processor1.processPayment(500.0);
        PaymentProcessor processor2 = new PaymentProcessor(new DeditCardPayment());
        processor2.processPayment(230.0);
    }
}
```

10. Apply Single Responsibility Principle

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

```
class UserDataProcessor{
    public String processName(String name){
        return name.trim().toUpperCase();
    }
}
class UserDataHandler{
    public void display(String message){
        System.out.println("Processed: " + message.toUpperCase());
    }
}
public class srp{
    public static void main(String[] args) {
        String raw = "  AlicE SmITh";
        UserDataProcessor processor = new UserDataProcessor();
        String processed = processor.processName(raw);
        UserDataHandler outputHandler = new UserDataHandler();
        outputHandler.display(processed);
    }
}
```

11. Use Interface with Default Method

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

```
interface greeting{
    default void greet(){
        System.out.println("Default Greeting");
    }
}
class customGreeting implements greeting{
    public void greet(){
        System.out.println("Custom greeting");
    }
}
class defaultGreet implements greeting{

}
public class interface_default{
    public static void main(String[] args) {
        customGreeting customGreeter = new customGreeting();
        customGreeter.greet();
        defaultGreet defaultGreeter = new defaultGreet();
        defaultGreeter.greet();
    }
}
```

12.Abstract Class with Constructor

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

```
abstract class Person{
    protected String name;
    public Person(String name){
        this.name = name;
    }
    public abstract void displayInfo();
}
class Employee extends Person{
    private String role;
    public Employee(String name, String role){
        super(name);
        this.role = role;
    }
    @Override
    public void displayInfo(){
        System.out.println("Name: " + name + ", Role: " + role);
    }
}
public class abstractClass{
    public static void main(String[] args){
        Employee emp = new Employee("Alice", "Developer");
        emp.displayInfo();
    }
}
```

13.Multiple Interfaces in One Class

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

```
interface Flyable {
    void fly();
}
interface Swimmable {
    void swim();
}
class Duck implements Flyable, Swimmable {

    @Override
    public void fly() {
        System.out.println("Duck is flying.");
    }
    @Override
    public void swim() {
        System.out.println("Duck is swimming.");
    }
```

```
}
public class multipleInterface {
    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.fly();
        duck.swim();
    }
}
```

14.Compare Abstract Class and Interface

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

```
abstract class Vehicle{
    String brand;
    public Vehicle(String brand){
        this.brand = brand;
    }
    public void start(){
        System.out.println(brand + " is starting");
    }
    public abstract void drive();
}
interface Electric{
    void charge();
    default void batteryStatus(){
        System.out.println("Battery status 89%");
    }
}
class ElectricCar extends Vehicle implements Electric{
    public ElectricCar(String brand){
        super(brand);
    }
    @Override
    public void drive(){
        System.out.println(brand + " is driving");
    }
    @Override
    public void charge(){
        System.out.println(brand + " is charging");
    }
}
public class abstractNInterface{
    public static void main(String[] args){
        ElectricCar tesla = new ElectricCar("Tesla");
        tesla.start();
        tesla.drive();
        tesla.charge();
        tesla.batteryStatus();
    }
}
```

15.Use Interface for Polymorphism

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

```
interface PaymentMethod {
    void pay(double amount);
}
class CreditCard implements PaymentMethod {
    @Override
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using Credit Card.");
    }
}
class UPI implements PaymentMethod {
    @Override
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using UPI.");
    }
}
class PaymentProcessor{
    public void PaymentMethod(PaymentMethod method, double amt){
        method.pay(amt);
    }
}
public class polymorphism{
    public static void main(String[] args) {
        PaymentProcessor processor = new PaymentProcessor();
        PaymentMethod cc = new CreditCard();
        PaymentMethod upi = new UPI();
        processor.PaymentMethod(cc, 10.23);
        processor.PaymentMethod(upi, 982.12);
    }
}
```

16.Perform CRUD using ArrayList

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

```
import java.util.*;
class StudentRecord{
    private List<String> students = new ArrayList<>();
    public void addStudent(String name){
        students.add(name);
    }
    public String getStudent(String name){
        for(String s : students){
            if(s.equalsIgnoreCase(name)){
                return s;
            }
        }
        return null;
    }
    public void updateStudent(String old, String newname){
        int ind = students.indexOf(old);
        if(ind != -1){
```

```
            students.set(ind, newname);
        }else{
            System.out.println("err");
        }
    }
    public void removeStudent(String name){
        if(students.remove(name)){
            System.out.println("student removed");
        }else{
            System.out.println("student not found");
        }
    }
    public void listStudents(){
        System.out.println("\nAll Students:");
        for (String name : students) {
            System.out.println(name);
        }
    }
}
public class arrlist{
    public static void main(String[] args){
        StudentRecord records = new StudentRecord();
        records.addStudent("Bruce");
        records.addStudent("Tony");
        records.addStudent("Bob");
        records.addStudent("Charlie");

        System.out.println("Retrieved: " + records.getStudent("Bob"));

        records.updateStudent("Charlie", "Charles");
        records.removeStudent("Bruce");
        records.listStudents();
    }
}
```

17.LinkedList Example for Playlist

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

```
import java.util.*;
public class playlist{
    public static void main(String[] args) {
        LinkedList<String> PlayList = new LinkedList<>();
        PlayList.add("Kiss the Ring");
        PlayList.add("Il Padrino");
        PlayList.add("Roar");
        System.out.println("Playlist: " + PlayList);
        PlayList.remove("Roar");
        System.out.println("Playlist: " + PlayList);
    }
}
```

18.Remove Duplicates using HashSet

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

```
import java.util.*;
public class hash{
    public static void main(String[] args){
        List<String> names = Arrays.asList("Narendra", "Doland", "Vladimir", "Boris", "Olaf", "Narendra");
        Set<String> uniqueNames = new HashSet<>(names);
        System.out.println(uniqueNames);
    }
}
```

19.Sort Data using TreeSet

Insert names in TreeSet and show sorted order output.

```
import java.util.*;
public class tree {
    public static void main(String[] args) {
        Set<String> names = new TreeSet<>();
        names.add("Charlie");
        names.add("Alice");
        names.add("Bob");
        names.add("Alice");
        System.out.println("Sorted names: " + names);
    }
}
```

20.HashMap Example - Student Grades

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

```
import java.util.*;
class Student{
    private String name;
    private String grade;
    public Student(String name, String grade){
        this.name = name;
        this.grade = grade;
    }
    public String getName(){
        return name;
    }
    public String getGrade(){
        return grade;
    }
    @Override
    public String toString(){
        return name + " (Grade: " + grade + ")";
    }
}
public class hashmap{
    public static void main(String[] args){
        Map<Integer, Student> students = new HashMap<>();
        students.put(101, new Student("Alice", "A"));
        students.put(102, new Student("Bob", "B"));
        students.put(103, new Student("Mike", "C"));
```

```
        for(Map.Entry<Integer, Student> entry : students.entrySet()){
            System.out.println(entry.getKey() + " : " + students.get(entry.getKey()));
        }
    }
}
```

21.LinkedHashMap for Recent Activities

Record user activity timestamps while maintaining insertion order.

```
import java.util.*;
public class linkedhash{
    public static void main(String[] args){
        Map<String, String> activitylog = new LinkedHashMap<>();
        activitylog.put("Alice", "10:00 AM");
        activitylog.put("Bob", "10:05 AM");
        activitylog.put("Charles", "11:15 AM");
        for(Map.Entry<String, String> entry : activitylog.entrySet()){
            System.out.println(entry.getKey() + " : " + entry.getValue());
        }
    }
}
```

22.Implement Queue with LinkedList

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

```
import java.util.*;
class linkedQ{
    private LinkedList<String> queue = new LinkedList<>();
    public void enqueue(String task){
        queue.add(task);
    }
    public String dequeue(){
        if(queue.isEmpty()){
            System.out.println("Queue is mt");
            return "";
        }
        String task = queue.removeFirst();
        System.out.println("Dequeue: " + task);
        return task;
    }
    public String peek() {
        return queue.peekFirst();
    }
    public boolean isEmpty(){
        return queue.isEmpty();
    }
    public void printQueue(){
        System.out.println(queue);
    }
}
```

23.Create a Generic Box Class

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

```
class Box<T>{
    private T content;
    public void setContent(T content){
        this.content = content;
    }
    public T getConent(){
        return content;
    }
    public void printContent(){
        System.out.println(content);
    }
}
public class generic{
    public static void main(String[] args) {
        Box<String> stringBox = new Box<>();
        stringBox.setContent("Hello world");
        stringBox.printContent();

        Box<Integer> intBox = new Box<>();
        intBox.setContent(1234);
        intBox.printContent();
    }
}
```

24.Write a Generic Swap Method

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

```
public class generic_swap{
    public static <T> void swap(T[] array, int i, int j){
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
    public static void main(String[] args) {
    Integer[] ints = {1,2,3,4,5};
    swap(ints, 0, 2);
    for(int n : ints){
        System.out.print(n + " ");
    }
    System.out.println("");
    String[] strings = {"A", "B", "C"};
    swap(strings, 0, 1);
    for(String s : strings){
        System.out.print(s + " ");
    }
    }
}
```

25.Bounded Generics Example

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

```
public class bounded_generics{
    public static <T extends Number> void printNumbericValue(T value){
```

```
        System.out.println("Numeric Value: "+ value);
    }
    public static void main(String[] args) {
        printNumbericValue(10);
        printNumbericValue(10.23);
        printNumbericValue(10.03f);
    }
}
```

26.Stream from Collection

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

```
import java.util.*;
public class list2stream{
    public static void main(String[] args){
        List<Integer> ints = Arrays.asList(1,2,23,3,4,5,6,7);
        ints.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.

```
import java.util.*;
import java.util.stream.*;
public class mapFilter{
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Modi", "Aryan");
        List<String> filteredNUppercased = names.stream()
                                        .filter(name ->name.startsWith("A"))
                                        .map(String::toUpperCase)
                                        .collect(Collectors.toList());
        System.out.println(filteredNUppercased);
    }
}
```

28.Use Reduce for Sum

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

```
import java.util.*;
public class reduceSum{
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(1,2,3,4,5,6, 100);
        int sum = nums.stream().reduce(0, (a, b) -> a + b);
        System.out.println(sum);
    }
}
```

29.Collect Stream to List

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

```
import java.util.*;
import java.util.stream.Collectors;
public class stream2List{
    public static void main(String[] args) {
        List<String> name = Arrays.asList("alice", "bob", "charlie", "dave");
        List<String> modifiedNames = name.stream()
                                            .map(String :: toUpperCase)
                                            .collect(Collectors.toList());
        System.out.println(modifiedNames);
    }
}
```

30. Parallel Stream Usage

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

```
import java.util.List;
import java.util.stream.*;
public class parallelStream{
    public static void main(String[] args){
        List<Integer> nums = IntStream.rangeClosed(1, 100_000)
                                    .boxed()
                                    .collect(Collectors.toList());
        long start1 = System.currentTimeMillis();
        long sum1 = nums.stream()
                        .mapToLong(n -> n * n)
                        .sum();
        long end1 = System.currentTimeMillis();

        long start2 = System.currentTimeMillis();
        long sum2 = nums.parallelStream()
                        .mapToLong(n -> n * n)
                        .sum();
        long end2 = System.currentTimeMillis();
        System.out.println("Stream sum: " + sum2 + " ,Time: " + (end1 - start1) + " ms");
        System.out.println("Parallel stream sum: " + sum2 + ", Time: " + (end2 - start2) + " ms");
    }
}
```

31.Inspect Class using Reflection

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

```
import java.lang.reflect.*;
class Student{
    private String name;
    private int age;
    public Student(){}
    public Student(String name, int age){
        this.name = name;
        this.age = age;
    }
    public void study(){}
    public String getName(){
        return name;
    }
    public int getAge(){
        return age;
    }
```

```
}
public class inspectReflection{
    public static void main(String[] args) {
        Class<?> cls = Student.class;
        System.out.println("Class name: " + cls.getName());
        System.out.println("Fields: ");
        for(Field field : cls.getDeclaredFields()){
            System.out.print(field.getName() + " - ");
        }
        System.out.println("\nMethods:");
        for(Method method : cls.getDeclaredMethods()){
            System.out.print(method + " - ");
        }
    }
}
```

32.Dynamic Object Creation using Reflection

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

```
class Student{
    public Student(){
        System.out.println("Student object created");
    }
    public void greet(){
        System.out.println("Hello Bachoooo");
    }
}
public class dynamicObject{
    public static void main(String[] args){
        try {
            Class<?> cls = Class.forName("Student");
            Object obj = cls.getDeclaredConstructor().newInstance();
            cls.getMethod("greet").invoke(obj);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

33. Access Private Field with Reflection

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

```
import java.lang.reflect.Field;
class Student{
    private String name = "Initial name";
    public String getName(){
        return name;
    }
}
public class accessPrivate{
    public static void main(String[] args) {
        try{
            Student student = new Student();
            Field field = Student.class.getDeclaredField("name");
            field.setAccessible(true);
            System.out.println("Before: " + field.get(student));
            field.set(student, "Alice");
            System.out.println("After: " + field.get(student));
            System.out.println("Via getter: " + student.getName());
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```