**Object Oriented Programming - CS3391**

**PART-B**

**UNIT I: INTRODUCTION TO OOP AND JAVA**

**1. Explain the key principles of Object-Oriented Programming (OOP) and how Java supports them.**

**Answer:** OOP is a programming paradigm based on the concept of "objects", which can contain data and methods. The four key principles of OOP are:

1. **Encapsulation**:
   * Binding data and methods into a single unit (class).
   * Java uses access specifiers like private, protected, and public to achieve encapsulation.
2. **Abstraction**:
   * Hiding complex implementation details and showing only essential features.
   * Java supports abstraction using abstract classes and interfaces.
3. **Inheritance**:
   * Mechanism by which one class inherits the properties of another class.
   * Java supports single inheritance through extends keyword and multiple through interfaces.
4. **Polymorphism**:
   * Ability to take many forms; one method behaves differently based on object.
   * Java supports polymorphism through method overloading and overriding.

Java is inherently object-oriented and follows all OOP principles strictly. Classes and objects are the core building blocks in Java.

**2. Discuss the Java buzzwords and explain how they shape the Java programming language.**

**Answer:** Java was designed with several important characteristics known as **Java Buzzwords**:

1. **Simple**: Easy to learn, with a syntax similar to C++ but without complex features like pointers.
2. **Object-Oriented**: Supports modular and reusable code using classes and objects.
3. **Platform-Independent**: Java programs are compiled into bytecode, which can run on any system with a Java Virtual Machine (JVM).
4. **Secure**: Java provides runtime checking and exception handling, preventing unsafe operations.
5. **Robust**: Strong memory management, exception handling, and type checking.
6. **Multithreaded**: Java has built-in support for multithreaded programming.
7. **Architecture-Neutral**: Bytecode is not platform-specific.
8. **Portable**: Java programs can run on any machine.
9. **High Performance**: Bytecode execution is fast due to the Just-In-Time (JIT) compiler.
10. **Distributed**: Supports networking and remote method invocation (RMI).
11. **Dynamic**: Can adapt to an evolving environment, supports loading classes at runtime.

These features make Java suitable for developing secure, portable, and high-performance applications.

**3. Explain data types, variables, and arrays in Java with suitable examples.**

**Answer:**

**Data Types:** Java has two categories of data types:

* **Primitive Data Types**: byte, short, int, long, float, double, char, boolean
* **Non-Primitive (Reference) Data Types**: Arrays, Classes, Interfaces

**Example:**

int age = 25;

double salary = 55000.5;

char grade = 'A';

boolean isPassed = true;

**Variables:**

* **Local Variables**: Declared inside methods.
* **Instance Variables**: Declared inside a class but outside methods.
* **Static Variables**: Declared with static keyword, shared by all instances.

**Arrays:**

* Arrays are containers that hold multiple values of the same data type.

**Example:**

int[] numbers = {10, 20, 30};

System.out.println(numbers[1]); // Output: 20

**4. What are control statements in Java? Explain with examples.**

**Answer:** Control statements are used to control the flow of execution in a program.

1. **Decision-making Statements**:
   * if, if-else, switch

int num = 10;

if (num > 0) {

System.out.println("Positive number");

}

1. **Looping Statements**:
   * for, while, do-while

for (int i = 0; i < 5; i++) {

System.out.println(i);

}

1. **Branching Statements**:
   * break, continue, return

for (int i = 0; i < 5; i++) {

if (i == 3) break;

System.out.println(i);

}

These control statements enable decision making, looping, and flow control in Java programs.

**5. Explain defining classes, constructors, methods, static members, and JavaDoc comments in Java.**

**Answer:**

**Defining Classes:**

* A class is a blueprint for objects.

class Student {

int id;

String name;

}

**Constructors:**

* Used to initialize objects. Same name as the class.

Student() {

id = 1;

name = "John";

}

**Methods:**

* Define behavior of objects.

void display() {

System.out.println(id + " " + name);

}

**Access Specifiers:**

* Control visibility of classes, methods, variables.
* private, public, protected, default (package-private)

**Static Members:**

* Belong to class, not instance.

static int count = 0;

**JavaDoc Comments:**

* Special comments to generate documentation.

/\*\*

\* This is a Student class.

\* @author XYZ

\*/

These components help in structuring Java programs effectively using object-oriented principles.

**UNIT II: INHERITANCE, PACKAGES AND INTERFACES**

**13-Mark Questions and Answers (with Detailed Explanation)**

**1. Explain Method Overloading and Method Overriding with examples.**

**Answer:**

* **Method Overloading** is when multiple methods in the same class have the same name but different parameters (type, number, or order). It increases readability.

class OverloadExample {

void show(int a) {

System.out.println("Argument: " + a);

}

void show(String a) {

System.out.println("String Argument: " + a);

}

}

* **Method Overriding** happens when a subclass provides a specific implementation of a method already defined in its superclass. It supports runtime polymorphism.

class Parent {

void display() {

System.out.println("Display from Parent");

}

}

class Child extends Parent {

void display() {

System.out.println("Display from Child");

}

}

**2. Describe the types of inheritance in Java with examples.**

**Answer:**

* Java supports:
  + **Single Inheritance**
  + **Multilevel Inheritance**
  + **Hierarchical Inheritance**

**Note:** Java doesn't support **Multiple Inheritance** with classes (to avoid ambiguity) but supports it via **Interfaces**.

**Example – Single Inheritance:**

class Animal {

void eat() {

System.out.println("This animal eats food.");

}

}

class Dog extends Animal {

void bark() {

System.out.println("The dog barks.");

}

}

**Example – Multilevel Inheritance:**

class Animal {

void eat() {}

}

class Dog extends Animal {

void bark() {}

}

class Puppy extends Dog {

void weep() {}

}

**3. What is Dynamic Method Dispatch? How is it implemented in Java?**

**Answer:**

* **Dynamic Method Dispatch** is the process where a call to an overridden method is resolved at runtime. It allows Java to support runtime polymorphism.

**Implementation:**

class Animal {

void sound() {

System.out.println("Animal sound");

}

}

class Dog extends Animal {

void sound() {

System.out.println("Dog barks");

}

}

class Cat extends Animal {

void sound() {

System.out.println("Cat meows");

}

}

public class Test {

public static void main(String[] args) {

Animal a;

a = new Dog();

a.sound(); // Dog barks

a = new Cat();

a.sound(); // Cat meows

}

}

**4. Explain Abstract Classes and the final keyword in inheritance with examples.**

**Answer:**

* **Abstract Classes:**
  + A class that cannot be instantiated.
  + It can contain abstract and non-abstract methods.

**Example:**

abstract class Animal {

abstract void makeSound();

void sleep() {

System.out.println("Sleeping");

}

}

class Dog extends Animal {

void makeSound() {

System.out.println("Dog barks");

}

}

* **final Keyword:**
  + Used to prevent inheritance, method overriding, or change in value (for variables).

final class A {}

// class B extends A {} // This will give error

**5. Discuss Packages and Interfaces in Java with syntax and example.**

**Answer:**

* **Packages** are containers for classes to avoid name conflicts and to control access.

**Creating Package:**

package mypackage;

public class A {

public void msg() {

System.out.println("Hello from package");

}

}

**Using Package:**

import mypackage.A;

class B {

public static void main(String[] args) {

A obj = new A();

obj.msg();

}

}

* **Interfaces:**
  + Interfaces contain only abstract methods (until Java 7) and constants.
  + A class implements an interface using implements keyword.

**Example:**

interface Printable {

void print();

}

class A implements Printable {

public void print() {

System.out.println("Hello");

}

}

**UNIT III: EXCEPTION HANDLING AND MULTITHREADING**

**Question 1: Explain the basic concepts of exception handling in Java with examples.**

**Answer:** Exception handling in Java allows a program to deal with unexpected events or errors during runtime. Java provides a robust and object-oriented way to handle exceptions.

**Key Concepts:**

* **try block**: Code that might throw an exception.
* **catch block**: Code to handle the exception.
* **finally block**: Executes code after try-catch, regardless of exception.
* **throw**: Used to explicitly throw an exception.
* **throws**: Declares exceptions a method might throw.

**Example:**

try {

int a = 10 / 0;

} catch (ArithmeticException e) {

System.out.println("Error: Division by zero");

} finally {

System.out.println("Finally block executed");

}

**Question 2: Describe multiple catch clauses and nested try statements in Java with examples.**

**Answer:** Java allows multiple catch blocks to handle different types of exceptions. Nested try blocks allow a try block inside another try block.

**Example - Multiple catch:**

try {

int[] arr = new int[5];

arr[10] = 30 / 0;

} catch (ArithmeticException e) {

System.out.println("Arithmetic Exception: " + e);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array Index Out Of Bounds: " + e);

}

**Example - Nested try:**

try {

try {

int b = 39 / 0;

} catch (ArithmeticException e) {

System.out.println("ArithmeticException caught");

}

} catch (Exception e) {

System.out.println("General Exception caught");

}

**Question 3: Explain the Java Thread Model and how to create threads in Java.**

**Answer:** Java supports multithreading, allowing concurrent execution of two or more threads.

**Java Thread Model:**

* Each thread is an independent path of execution.
* Threads share memory space.

**Creating Threads:**

1. **By extending Thread class:**

class MyThread extends Thread {

public void run() {

System.out.println("Thread is running...");

}

}

MyThread t1 = new MyThread();

t1.start();

1. **By implementing Runnable interface:**

class MyRunnable implements Runnable {

public void run() {

System.out.println("Thread is running...");

}

}

Thread t2 = new Thread(new MyRunnable());

t2.start();

**Question 4: Explain thread synchronization and inter-thread communication with examples.**

**Answer:** **Thread Synchronization:** Used to control access to shared resources. Synchronized blocks or methods prevent data inconsistency.

**Example:**

class Table {

synchronized void printTable(int n) {

for (int i = 1; i <= 5; i++) {

System.out.println(n \* i);

}

}

}

**Inter-thread Communication:** Allows threads to communicate using wait(), notify(), and notifyAll().

**Example:**

class Shared {

synchronized void print() throws InterruptedException {

wait(); // wait for notification

System.out.println("Thread resumed");

}

synchronized void resumeThread() {

notify();

}

}

**Question 5: Explain wrapper classes and autoboxing in Java.**

**Answer:** Java provides wrapper classes for each primitive type (int, char, etc.) to use them as objects.

**Wrapper Classes:**

* int -> Integer
* char -> Character
* double -> Double

**Autoboxing:** Automatic conversion of primitive to object type.

int a = 5;

Integer obj = a; // autoboxing

**Unboxing:** Conversion of object to primitive type.

Integer obj = new Integer(10);

int b = obj; // unboxing

**Use Cases:**

* Collection frameworks (like ArrayList) store only objects.
* Required in synchronization and serialization.

**UNIT IV: I/O, GENERICS, STRING HANDLING – 13 MARK QUESTIONS AND ANSWERS**

**Q1. Explain the basics of Java I/O streams with examples.**

**Answer:** Java I/O (Input and Output) is used to read and write data (input/output). Java provides the java.io package for I/O operations. It supports two types of streams:

1. **Byte Streams**: Handles input and output of 8-bit bytes. Classes: FileInputStream, FileOutputStream.
2. **Character Streams**: Handles input and output of 16-bit characters. Classes: FileReader, FileWriter.

**Example - Reading a file using FileReader:**

import java.io.\*;

public class ReadFileExample {

public static void main(String[] args) {

try {

FileReader fr = new FileReader("example.txt");

int i;

while ((i = fr.read()) != -1) {

System.out.print((char) i);

}

fr.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Example - Writing to a file using FileWriter:**

FileWriter fw = new FileWriter("output.txt");

fw.write("Welcome to Java I/O");

fw.close();

**Q2. Describe generic classes and generic methods with suitable examples.**

**Answer:** **Generics** in Java allow you to write a single class or method that can operate on different types of data.

**Generic Class:**

class Box<T> {

private T value;

public void setValue(T value) { this.value = value; }

public T getValue() { return value; }

}

**Usage:**

Box<Integer> intBox = new Box<>();

intBox.setValue(100);

System.out.println(intBox.getValue());

**Generic Method:**

public class GenericMethodExample {

public static <T> void printArray(T[] array) {

for (T element : array) {

System.out.print(element + " ");

}

System.out.println();

}

}

**Usage:**

Integer[] intArr = {1, 2, 3};

GenericMethodExample.printArray(intArr);

**Advantages:**

* Type safety
* Code reusability
* Elimination of type casting

**Q3. Explain bounded types in Java Generics with examples.**

**Answer:** Bounded types allow you to restrict the types that can be used as type arguments in a generic class or method.

**Syntax:**

<T extends SuperClass>

**Example:**

class Stats<T extends Number> {

T[] nums;

Stats(T[] o) { nums = o; }

double average() {

double sum = 0.0;

for (T num : nums) {

sum += num.doubleValue();

}

return sum / nums.length;

}

}

**Usage:**

Integer[] intArr = {1, 2, 3, 4};

Stats<Integer> intStats = new Stats<>(intArr);

System.out.println("Average: " + intStats.average());

**Note:** If T does not extend Number, it will result in a compile-time error.

**Q4. Discuss the restrictions and limitations of Generics in Java.**

**Answer:**

1. **Cannot instantiate generic types with primitives**:
2. // Invalid: List<int> list = new ArrayList<int>();
3. // Use wrapper class: List<Integer>
4. **Cannot create generic arrays**:
5. // Invalid: T[] arr = new T[10];
6. **Cannot use instanceof with parameterized types**:
7. // Invalid: if (obj instanceof List<String>)
8. **Cannot create static fields of type parameters**:
9. // Invalid: static T value;
10. **Type erasure**:
    * Generics are implemented using type erasure, so the generic type is replaced with Object during compilation.

**Conclusion:** While generics provide flexibility and type safety, developers must understand these limitations to avoid unexpected behaviors.

**Q5. Explain String handling in Java with the use of String, StringBuffer, and StringBuilder classes.**

**Answer:** Java provides different classes to handle strings:

1. **String Class**:
   * Immutable.
   * Used when constant strings are needed.
   * Stored in a string pool.

String s = "Java";

s = s.concat(" Programming"); // creates a new string

System.out.println(s);

1. **StringBuffer Class**:
   * Mutable, thread-safe.
   * Used for strings that are modified frequently in a multithreaded environment.

StringBuffer sb = new StringBuffer("Hello");

sb.append(" Java");

System.out.println(sb);

1. **StringBuilder Class**:
   * Mutable, not thread-safe (faster).
   * Preferred when thread safety is not required.

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

System.out.println(sb);

**Common Methods:**

* length(), charAt(), substring(), replace(), toString()

**Comparison:**

| **Class** | **Mutable** | **Thread Safe** | **Performance** |
| --- | --- | --- | --- |
| String | No | Yes | Low |
| StringBuffer | Yes | Yes | Medium |
| StringBuilder | Yes | No | High |

**Use Cases:**

* Use String when immutability is required.
* Use StringBuffer for multithreaded operations.
* Use StringBuilder for high-performance single-threaded operations.

**UNIT V - JAVAFX EVENT HANDLING, CONTROLS AND COMPONENTS**

**13-MARK QUESTIONS AND DETAILED ANSWERS**

**Question 1: Explain JavaFX event handling model with examples.**

**Answer:** JavaFX provides an event-driven programming model. Events represent occurrences such as key presses, mouse actions, or any other user interactions.

**Event Handling Mechanism:**

1. **Event Source**: The GUI component (e.g., Button) that generates the event.
2. **Event Object**: Encapsulates the event details.
3. **Event Handler**: Interface that handles the event (contains handle() method).

**Steps to handle events in JavaFX:**

1. Register the event handler with the event source.
2. Implement the EventHandler interface or use lambda expressions.

**Example:**

Button btn = new Button("Click Me");

btn.setOnAction(e -> {

System.out.println("Button Clicked");

});

**Types of Events:**

* ActionEvent
* MouseEvent
* KeyEvent

**KeyEvent and MouseEvent Example:**

scene.setOnKeyPressed(e -> System.out.println("Key Pressed: " + e.getCode()));

scene.setOnMouseClicked(e -> System.out.println("Mouse Clicked at: " + e.getX() + ", " + e.getY()));

**Question 2: Describe various JavaFX controls with examples.**

**Answer:** JavaFX provides many UI controls to build responsive and interactive applications.

1. **CheckBox**

CheckBox cb = new CheckBox("Accept Terms");

1. **ToggleButton**

ToggleButton tbtn = new ToggleButton("ON/OFF");

1. **RadioButton**

RadioButton rb1 = new RadioButton("Option 1");

RadioButton rb2 = new RadioButton("Option 2");

ToggleGroup group = new ToggleGroup();

rb1.setToggleGroup(group);

rb2.setToggleGroup(group);

1. **ListView**

ListView<String> listView = new ListView<>();

listView.getItems().addAll("A", "B", "C");

1. **ComboBox and ChoiceBox**

ComboBox<String> comboBox = new ComboBox<>();

comboBox.getItems().addAll("Red", "Blue", "Green");

**Text Controls:**

* TextField, TextArea, PasswordField

**ScrollPane**: Allows scrolling of content.

ScrollPane scroll = new ScrollPane(new TextArea());

**Question 3: Explain layout panes in JavaFX with examples.**

**Answer:** JavaFX layout panes help in arranging UI components.

1. **FlowPane**: Places components in a flow, wrapping them to the next line.

FlowPane flow = new FlowPane();

flow.getChildren().addAll(btn1, btn2);

1. **HBox and VBox**: Horizontal and vertical box layouts.

HBox hbox = new HBox(10);

hbox.getChildren().addAll(btn1, btn2);

1. **BorderPane**: Divides layout into Top, Bottom, Center, Left, Right.

BorderPane pane = new BorderPane();

pane.setTop(menuBar);

pane.setCenter(content);

1. **StackPane**: Stacks components on top of each other.

StackPane stack = new StackPane();

stack.getChildren().addAll(label, image);

1. **GridPane**: Arranges nodes in a grid of rows and columns.

GridPane grid = new GridPane();

grid.add(label, 0, 0);

grid.add(textField, 1, 0);

**Question 4: Write a JavaFX program using Menu, MenuBar, and MenuItem.**

**Answer:** JavaFX supports menu systems using MenuBar, Menu, and MenuItem.

**Code Example:**

MenuBar menuBar = new MenuBar();

Menu fileMenu = new Menu("File");

MenuItem newItem = new MenuItem("New");

MenuItem exitItem = new MenuItem("Exit");

fileMenu.getItems().addAll(newItem, exitItem);

menuBar.getMenus().add(fileMenu);

exitItem.setOnAction(e -> Platform.exit());

**Explanation:**

* MenuBar holds the menu.
* Menu is a dropdown containing MenuItems.
* Events can be attached to each item.

**Question 5: Create a GUI application using various JavaFX components.**

**Answer:** **Problem Statement**: Create a login form with username and password using JavaFX.

**JavaFX Application:**

Label userLabel = new Label("Username:");

TextField userField = new TextField();

Label passLabel = new Label("Password:");

PasswordField passField = new PasswordField();

Button loginBtn = new Button("Login");

loginBtn.setOnAction(e -> {

if(userField.getText().equals("admin") && passField.getText().equals("1234")) {

System.out.println("Login Successful");

} else {

System.out.println("Login Failed");

}

});

GridPane grid = new GridPane();

grid.setVgap(10);

grid.setHgap(10);

grid.add(userLabel, 0, 0);

grid.add(userField, 1, 0);

grid.add(passLabel, 0, 1);

grid.add(passField, 1, 1);

grid.add(loginBtn, 1, 2);

**Features Used:**

* GridPane layout
* Label, TextField, PasswordField, Button
* Event Handling using Lambda expressions