150 Important Viva Questions with Answers for Object-Oriented Programming (CSC-122)

These questions and answers are based on the CSC-122 Object-Oriented Programming course outline, covering topics such as Moving from C to Java, OOP principles, Java fundamentals, classes, inheritance, polymorphism, exception handling, GUI programming, and Java Database Connectivity. They are designed to be logical and time-consuming to prepare you for your viva.

Week 1: Moving from C to Java

1. **Explain the key differences between procedural programming in C and object-oriented programming in Java.**
   - **Answer**: Procedural programming in C focuses on procedures or functions that manipulate data, with a top-down approach. Data and functions are separate, and programs are structured around sequential tasks. In contrast, Java's OOP approach organizes code around objects, which encapsulate data and behavior. Key differences include:
     - **Encapsulation**: Java uses classes to bundle data and methods, while C uses structs with no inherent methods.
     - **Inheritance**: Java supports class inheritance for code reuse, unavailable in C.
     - **Polymorphism**: Java allows method overriding and overloading, while C relies on function pointers for similar behavior.
     - **Memory Management**: Java uses automatic garbage collection, while C requires manual memory management with `malloc` and `free`.
     - **Platform Independence**: Java runs on the JVM, making it platform-independent, whereas C compiles to machine-specific code.
     - Example: In C, a program might use functions to process a `struct` for a student's data. In Java, a `Student` class encapsulates attributes (e.g., `name`, `id`) and methods (e.g., `getName()`).
2. **Why is Java considered a platform-independent language? Discuss the role of JVM in achieving this.**
   - **Answer**: Java is platform-independent because its compiled code (bytecode) can run on any system with a Java Virtual Machine (JVM). The JVM interprets bytecode into machine-specific instructions, abstracting hardware and OS differences. Steps:
     - Java source code (`.java`) is compiled into bytecode (`.class`) by the Java compiler (`javac`).
     - The JVM executes bytecode, translating it to native code for the host system.
     - The JVM ensures consistent behavior across platforms (e.g., Windows, Linux).

        - Example: A Java program written on Windows runs unchanged on Linux if the JVM is installed.
        - The JVM's role includes class loading, memory management, and exception handling, ensuring portability.

3. **Describe the history of Java and its evolution as a dominant programming language.**
    - **Answer**: Java was developed by Sun Microsystems, initiated by James Gosling in 1991 as "Oak" for embedded systems. Renamed Java, it was released in 1995 as a general-purpose language with the slogan "write once, run anywhere." Key milestones:
        - **1995**: Java 1.0 introduced applets and basic OOP features.
        - **2000s**: Java 2 Enterprise Edition (J2EE) popularized web applications.
        - **2004**: Java 5 added generics, enums, and annotations.
        - **2014**: Java 8 introduced lambda expressions and streams for functional programming.
        - **Recent**: Regular updates (e.g., Java 17) add features like records and pattern matching.
        - Java's dominance stems from platform independence, robust libraries, and use in web, mobile (Android), and enterprise applications.
4. **What are the Java buzzwords? Explain how each contributes to Java's design philosophy.**
    - **Answer**: Java's buzzwords, outlined by Sun Microsystems, describe its design principles:
        - **Simple**: Java avoids complex C++ features like pointers and multiple inheritance.
        - **Object-Oriented**: Emphasizes encapsulation, inheritance, and polymorphism via classes.
        - **Portable**: Bytecode runs on any JVM-equipped platform.
        - **Robust**: Features like garbage collection and exception handling prevent crashes.
        - **Secure**: Sandboxing and bytecode verification protect against malicious code.
        - **Multithreaded**: Built-in support for concurrent programming.
        - **High-Performance**: JIT compilation optimizes bytecode execution.
        - **Distributed**: Supports networking and RMI for distributed systems.
        - **Dynamic**: Runtime class loading enables flexibility.
        - These principles make Java versatile, reliable, and widely adopted.
5. **Differentiate between JDK, JRE, and JVM with a detailed example of their interaction during program execution.**
    - **Answer**:
        - **JDK (Java Development Kit)**: A software development kit containing tools (`javac`, `jar`) and the JRE for compiling and running Java programs.
        - **JRE (Java Runtime Environment)**: A runtime environment with the JVM, libraries, and files needed to run Java applications but not compile them.

        - **JVM (Java Virtual Machine)**: An abstract machine that executes bytecode, manages memory, and provides platform independence.
        - **Interaction Example**:
            - You write a Java program (`HelloWorld.java`).
            - The JDK's `javac` compiles it to `HelloWorld.class` (bytecode).
            - The JRE's JVM loads the bytecode, verifies it, and executes it using native code via the JIT compiler.
            - Libraries in the JRE (e.g., `java.lang`) provide functions like `System.out.println`.
        - Without the JDK, you can run but not compile; without the JRE, the JVM cannot execute.

6. **How does Java's memory management differ from C's manual memory management?**
    - **Answer**: Java uses automatic memory management via garbage collection, while C requires manual management:
        - **Java**: The JVM allocates memory for objects on the heap. The garbage collector (GC) automatically reclaims memory for objects no longer referenced, preventing memory leaks. Example: Creating an object with `new` is automatically managed.
        - **C**: Programmers use `malloc` to allocate memory and `free` to deallocate it. Errors like dangling pointers or memory leaks are common if mismanaged.
        - **Example**: In Java, `String s = new String("test"); s = null;` allows the GC to reclaim memory. In C, `char *s = malloc(5);` requires `free(s);` to avoid leaks.
        - Java's approach is safer but less performant for low-level tasks; C offers control but is error-prone.
7. **Why is Java preferred over C for large-scale applications? Provide specific examples.**
    - **Answer**: Java is preferred for large-scale applications due to:
        - **Platform Independence**: Runs on any JVM-equipped system, ideal for distributed systems like web servers.
        - **Robustness**: Exception handling and type safety reduce errors.
        - **Scalability**: Libraries like Spring and Hibernate support enterprise applications.
        - **OOP Features**: Encapsulation and inheritance improve modularity.
        - **Examples**:
            - **Web Applications**: Java powers platforms like LinkedIn using Spring Boot.
            - **Enterprise Systems**: Banking systems use Java for secure, scalable transactions.
            - **Android Apps**: Java (and Kotlin) is used for mobile app development.
        - C is better for system-level programming (e.g., OS kernels) but lacks Java's high-level abstractions.

8. **Explain how Java's "write once, run anywhere" principle is implemented in practice.**
    - **Answer**: The "write once, run anywhere" (WORA) principle is achieved via:
        - **Bytecode**: Java source code is compiled to platform-independent bytecode by `javac`.
        - **JVM**: Each platform has a JVM that interprets bytecode into native instructions.
        - **Standard Libraries**: Java's API ensures consistent behavior across platforms.
        - **Practice**: A Java program written on Windows is compiled to bytecode, copied to Linux, and run using the Linux JVM without modification.
        - **Example**: A Java Swing application runs identically on macOS and Windows with the same `.class` files.
        - Challenges include JVM version compatibility and platform-specific bugs.
9. **Compare the compilation and execution process of a C program with a Java program.**
    - **Answer**:
        - **C Program**:
            - **Compilation**: The C compiler (`gcc`) translates source code (`.c`) to machine code (`.exe` or `.o`) specific to the platform.
            - **Execution**: The OS directly runs the machine code, interacting with hardware.
            - **Example**: `gcc program.c -o program; ./program` runs the executable.
        - **Java Program**:
            - **Compilation**: The Java compiler (`javac`) converts source code (`.java`) to bytecode (`.class`), which is platform-independent.
            - **Execution**: The JVM interprets or compiles bytecode to native code at runtime.
            - **Example**: `javac HelloWorld.java; java HelloWorld` executes via the JVM.
        - **Key Difference**: C produces native code, while Java uses bytecode and the JVM for portability.
10. **Discuss the limitations of Java compared to C in terms of performance and low-level operations.**
    - **Answer**:
        - **Performance**: Java's JVM introduces overhead (bytecode interpretation, JIT compilation), making it slower than C's native code for CPU-intensive tasks like game engines.
        - **Low-Level Operations**: Java lacks direct memory access (no pointers), restricting hardware control. C's pointers enable fine-grained memory manipulation.
        - **Garbage Collection**: Java's GC reduces control over memory deallocation, unlike C's `free`.

        - **Example**: C is used for OS kernels (e.g., Linux) due to direct hardware access, while Java is unsuitable.
        - **Trade-Off**: Java sacrifices performance for safety and portability, making it less ideal for system programming.

Week 2: Introduction to OOP

11. **Define programming paradigms and explain how OOP differs from procedural and functional programming.**
    - **Answer**: A programming paradigm is a style of structuring code. Key paradigms:
        - **Procedural**: Organizes code into functions that manipulate data (e.g., C). Focuses on sequential steps.
        - **Functional**: Treats computation as mathematical functions, avoiding state changes (e.g., Haskell).
        - **OOP**: Organizes code around objects that combine data and behavior (e.g., Java).
        - **Differences**:
            - OOP uses classes/objects, enabling encapsulation, inheritance, and polymorphism.
            - Procedural programming separates data and functions, lacking inheritance.
            - Functional programming avoids mutable state, unlike OOP's stateful objects.
        - **Example**: In OOP, a `Car` class encapsulates `speed` and `drive()`. In procedural C, `drive()` is a function acting on a `struct Car`.
12. **What is abstraction in OOP? Provide a real-world example implemented in Java.**
    - **Answer**: Abstraction hides complex implementation details, exposing only essential features. In Java, it's achieved via abstract classes or interfaces.
        - **Example**: A `Vehicle` abstract class defines a `move()` method without implementation. Subclasses like `Car` provide specific behavior.

```
abstract class Vehicle {
    abstract void move();
}
class Car extends Vehicle {
    void move() {
        System.out.println("Car moves on roads");
    }
}
```

        - **Real-World**: A car's dashboard (speedometer) abstracts engine details, showing only speed.
13. **Explain the three core OOP principles (Encapsulation, Inheritance, Polymorphism) with Java code snippets.**
    - **Answer**:
        - **Encapsulation**: Bundles data and methods, restricting access via modifiers.

```
class BankAccount {
    private double balance;
```

```
            public void deposit(double amount) {
                if (amount > 0) balance += amount;
            }
            public double getBalance() { return balance; }
        }
        ```
        - **Inheritance**: Allows a class to inherit properties from another.

        ```
        class Animal { void eat() { System.out.println("Eats food"); } }
        class Dog extends Animal { void bark() { System.out.println("Barks"); } }
        ```
        - **Polymorphism**: Enables objects to be treated as instances of a parent class with specific behaviors.

        ```
        class Shape { void draw() { System.out.println("Drawing shape"); } }
        class Circle extends Shape { void draw() { System.out.println("Drawing circle"); } }
        ```

14. **How do encapsulation and inheritance work together to improve code modularity?**
    - **Answer**: Encapsulation hides data (private fields, public methods), ensuring controlled access. Inheritance allows code reuse by extending classes. Together:
        - Encapsulation protects a parent class's data, which subclasses inherit.
        - Inheritance reuses encapsulated logic, reducing redundancy.
        - **Example**: A `Person` class encapsulates `name` and `getName()`. A `Student` class inherits and adds `grade`, reusing `getName()` while protecting its own data.
        - This modularity separates concerns, easing maintenance and extension.
15. **Discuss the advantages of OOP over procedural programming with respect to software maintenance.**
    - **Answer**: OOP improves maintenance via:
        - **Modularity**: Classes encapsulate logic, making changes localized.
        - **Reusability**: Inheritance and polymorphism reuse code.
        - **Scalability**: OOP supports large systems with clear object hierarchies.
        - **Debugging**: Encapsulation isolates errors to specific classes.
        - **Example**: Updating a `BankAccount` class's interest calculation doesn't affect `Customer` class, unlike procedural code where global functions may require widespread changes.
        - Procedural code is harder to maintain in large projects due to tight coupling.
16. **Explain how polymorphism enables flexible and reusable code in Java.**
    - **Answer**: Polymorphism allows objects of different classes to be treated as a common type, enabling flexible method calls. Types:
        - **Compile-Time**: Method overloading resolves calls based on parameters.
        - **Runtime**: Method overriding allows subclasses to provide specific implementations.
        - **Example**:

        ```
        class Animal { void sound() { System.out.println("Some sound"); } }
        class Cat extends Animal { void sound() { System.out.println("Meow"); } }
        Animal animal = new Cat();
        ```

        - `animal.sound(); // Outputs "Meow"`
        - **Benefit**: Code can handle new subclasses without modification, enhancing reusability.

17. **Provide an example where abstraction and encapsulation are used together in a Java program.**
    - **Answer**:

    ```
    abstract class Appliance {
        abstract void operate();
    }
    class WashingMachine extends Appliance {
        private int loadCapacity;
        public WashingMachine(int capacity) { this.loadCapacity = capacity; }
        void operate() { System.out.println("Washing " + loadCapacity + "kg load"); }
        public int getLoadCapacity() { return loadCapacity; }
    }
    ```

        - **Abstraction**: `Appliance` defines `operate()` without implementation.
        - **Encapsulation**: `WashingMachine` hides `loadCapacity`, exposing it via `getLoadCapacity()`.

18. **How does OOP facilitate teamwork in large software projects compared to procedural programming?**
    - **Answer**: OOP supports teamwork by:
        - **Modularity**: Classes divide work into independent units (e.g., one team handles `User`, another `Database`).
        - **Encapsulation**: Hides implementation, allowing teams to work without interfering.
        - **Interfaces**: Define contracts, enabling parallel development.
        - **Example**: In a banking system, one team develops `Account` (encapsulated logic), another `Transaction` (using `Account`'s interface).
        - Procedural programming's global functions cause conflicts, as changes affect shared data.

19. **Write a Java program demonstrating the concept of abstraction using abstract classes.**
    - **Answer**:

    ```
    abstract class Shape {
        abstract double area();
    }
    class Rectangle extends Shape {
        double width, height;
        Rectangle(double w, double h) { width = w; height = h; }
        double area() { return width * height; }
    }
    class Circle extends Shape {
        double radius;
        Circle(double r) { radius = r; }
        double area() { return Math.PI * radius * radius; }
    }
    public class Main {
        public static void main(String[] args) {
            Shape rect = new Rectangle(5, 3);
    ```

```
            Shape circle = new Circle(2);
            System.out.println("Rectangle Area: " + rect.area());
            System.out.println("Circle Area: " + circle.area());
        }
    }
```

- **Explanation**: `Shape` abstracts the `area()` method, implemented differently by `Rectangle` and `Circle`.

20. **Discuss the challenges of transitioning from procedural to object-oriented thinking for a C programmer.**
    - **Answer**: Challenges include:
        - **Mindset Shift**: Moving from function-based to object-based design requires thinking in terms of entities (objects) rather than tasks.
        - **Encapsulation**: C programmers may struggle with access control and data hiding.
        - **Inheritance/Polymorphism**: These concepts are absent in C, requiring new design patterns.
        - **Example**: A C programmer might write a function to process a `struct`. In Java, they must design a class with methods and consider inheritance.
        - **Solution**: Practice designing classes and using OOP principles in small projects.

Week 3: Introduction to Java

21. **Write and explain the structure of your first Java application, including the `main` method.**
    - **Answer**:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- **Structure**:
    - `public class HelloWorld`: Defines a class named `HelloWorld`.
    - `public static void main(String[] args)`: The entry point, where execution begins. `public` makes it accessible, `static` allows calling without an instance, `void` indicates no return, and `args` accepts command-line arguments.
    - `System.out.println`: Prints output to the console.
- **Execution**: Compile with `javac HelloWorld.java`, run with `java HelloWorld`.

22. **Demonstrate the use of `if` and `for` control statements in a Java program with a practical example.**
    - **Answer**:

```
public class NumberCheck {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        for (int num : numbers) {
            if (num % 2 == 0) {
                System.out.println(num + " is even");
            } else {
                System.out.println(num + " is odd");
            }
        }
    }
}
```

- **Explanation**: The `for` loop iterates over an array. The `if-else` checks if each number is even or odd, printing the result.

23. **Explain the significance of block code in Java and how it affects variable scope.**
    - **Answer**: A block is a group of statements enclosed in `{}`. It defines a scope where variables are accessible only within the block.
        - **Example**:

```
public class ScopeDemo {
    public static void main(String[] args) {
        int x = 10;
        {
            int y = 20; // y is local to this block
            System.out.println("x: " + x + ", y: " + y);
        }
        // System.out.println(y); // Error: y is out of scope
    }
}
```

- **Significance**: Blocks limit variable visibility, preventing naming conflicts and reducing memory usage.

24. **What are lexical issues in Java? Provide examples of naming conventions and their importance.**
    - **Answer**: Lexical issues involve syntax rules like identifiers, whitespace, and case sensitivity.
        - **Naming Conventions**:
            - Classes: CamelCase (e.g., `MyClass`).
            - Methods/Variables: camelCase (e.g., `myMethod`, `myVariable`).
            - Constants: UPPER_CASE (e.g., `MAX_VALUE`).
        - **Example**:

```
public class StudentRecord {
    private String studentName;
    public void setStudentName(String name) { studentName = name; }
}
```

- **Importance**: Conventions improve readability, maintainability, and team collaboration.

25. **Discuss the importance of Java documentation (Javadoc) and demonstrate how to generate it for a class.**
    - **Answer**: Javadoc generates API documentation from comments, aiding developers in understanding code.
        - **Example**:

```
/**
 * Represents a student with name and ID.
 * @author YourName
*/
 public class Student {
 private String name;
 /** Sets the student's name. @param name The name to set */
 public void setName(String name) { this.name = name; }
 }
```

- **Generation**: Run `javadoc Student.java` to create HTML documentation.
- **Importance**: Documents code purpose, parameters, and usage, essential for libraries and teamwork.

26. **Compare the programming style in Java with C, focusing on readability and maintainability.**
    - **Answer**:
        - **Java**: Emphasizes OOP, with classes, methods, and strict typing. Javadoc and naming conventions enhance readability. Garbage collection reduces memory errors.
        - **C**: Procedural, with functions and manual memory management. Less structured, relying on comments for clarity.
        - **Example**: Java's `class` encapsulates logic, while C's `struct` and functions are separate, reducing maintainability.
        - **Conclusion**: Java's OOP and tools improve readability and maintenance for large projects.
27. **Write a Java program that uses nested `if` statements and explain its control flow.**
    - **Answer**:

```
public class GradeCalculator {
    public static void main(String[] args) {
        int score = 85;
        if (score >= 90) {
            System.out.println("Grade: A");
        } else {
            if (score >= 80) {
                System.out.println("Grade: B");
            } else {
                System.out.println("Grade: C or below");
            }
        }
    }
}
```

- **Control Flow**: Checks if `score >= 90`. If false, enters the `else` block, checking `score >= 80`. Outputs "Grade: B" for `score = 85`.

28. **How does Java handle whitespace compared to C? Explain with examples.**
    - **Answer**: Both Java and C ignore extra whitespace (spaces, tabs, newlines) during compilation, but:
        - **Java**: Requires whitespace for readability in OOP constructs (e.g., class definitions). Case-sensitive.
        - **C**: Similar, but less strict due to procedural nature.
        - **Example**:

```
public class Test { int x=10; } // Java: Valid, but poor style
```

        - `int main(){int x=10;return 0;} // C: Valid, but unreadable`
        - **Difference**: Java's conventions enforce clearer formatting for OOP.

29. **Explain the role of the `public static void main(String[] args)` method in Java programs.**
    - **Answer**: The `main` method is the entry point for Java programs:
        - **public**: Accessible to the JVM.
        - **static**: Callable without creating an instance.
        - **void**: Returns no value.
        - **String[] args**: Accepts command-line arguments.
        - **Example**:

        ```
        public class Main {
            public static void main(String[] args) {
                System.out.println("Args: " + args.length);
            }
        }
        ```

        - Running `java Main test` prints "Args: 1".
30. **Write a Java program that demonstrates proper commenting and documentation practices.**
    - **Answer**:

    ```
    /**
     * A simple calculator class for basic arithmetic.
     * @author YourName
     */
    public class Calculator {
        /**
         * Adds two integers.
         * @param a First integer
         * @param b Second integer
         * @return Sum of a and b
         */
        public int add(int a, int b) {
            // Compute sum
            return a + b;
        }
        public static void main(String[] args) {
            Calculator calc = new Calculator();
            System.out.println("Sum: " + calc.add(5, 3)); // Test addition
        }
    }
    ```

        - **Practices**: Javadoc for class/methods, inline comments for logic.

Week 4: Fundamental Elements of Language

31. **List and explain all primitive data types in Java with their memory sizes and use cases.**
    - **Answer**:
        - **byte**: 8 bits, -128 to 127. Use: Small integers (e.g., flags).
        - **short**: 16 bits, -32,768 to 32,767. Use: Memory-constrained integers.
        - **int**: 32 bits, -2^31 to 2^31-1. Use: General-purpose integers.

        - **long**: 64 bits, -2^63 to 2^63-1. Use: Large numbers (e.g., timestamps).
        - **float**: 32 bits, ±3.4E38. Use: Approximate decimals (e.g., graphics).
        - **double**: 64 bits, ±1.7E308. Use: Precise decimals (e.g., calculations).
        - **char**: 16 bits, Unicode 0 to 65,535. Use: Characters (e.g., text).
        - **boolean**: Size unspecified, true/false. Use: Conditions.
        - **Example**: `int age = 25; double salary = 50000.50;`

32. **What are literals in Java? Provide examples of integer, floating-point, and string literals.**
    - **Answer**: Literals are fixed values in code.
        - **Integer**: `int x = 42;` (decimal), `int y = 0x2A;` (hex).
        - **Floating-Point**: `double d = 3.14;` (decimal), `float f = 1.2e3;` (exponential).
        - **String**: `String s = "Hello";`
        - **Example**:

          ```
          int num = 100;
          double pi = 3.14159;
          String greeting = "Hi";
          ```
33. **Explain escape sequences in Java with a program that uses at least three different sequences.**
    - **Answer**: Escape sequences are special characters prefixed with `\` for formatting strings.
        - **Example**:

          ```
          public class EscapeDemo {
              public static void main(String[] args) {
                  System.out.println("Line 1\nLine 2"); // \n: Newline
                  System.out.println("Tab\tSpace"); // \t: Tab
                  System.out.println("Quote: \"Text\""); // \": Double quote
              }
          }
          ```
        - **Output**:

          ```
          Line 1
          Line 2
          Tab     Space
          Quote: "Text"
          ```
34. **Discuss the scope and lifetime of variables in Java with examples of local and instance variables.**
    - **Answer**:
        - **Local Variables**: Declared in methods/blocks, exist only during execution.
        - **Instance Variables**: Declared in a class, exist for the object's lifetime.
        - **Example**:

          ```
          public class VariableDemo {
              int instanceVar = 10; // Instance variable
              void method() {
                  int localVar = 20; // Local variable
                  System.out.println("Local: " + localVar + ", Instance: " + instanceVar);
          ```

```
    }
    public static void main(String[] args) {
        VariableDemo obj = new VariableDemo();
        obj.method();
        // System.out.println(localVar); // Error: localVar out of scope
    }
}
```

35. **Demonstrate type conversion and casting in Java with a program that handles both widening and narrowing conversions.**
    - **Answer**:

    ```
    public class TypeConversion {
        public static void main(String[] args) {
            // Widening (automatic)
            int i = 100;
            double d = i; // int to double
            System.out.println("Widening:,OA: " + d);
            // Narrowing (explicit)
            double x = 3.14;
            int y = (int) x; // double to int
            System.out.println("Narrowing: " + y);
        }
    }
    ```

        - **Explanation**: Widening (int to double) is safe; narrowing (double to int) requires casting and may lose precision.
36. **What is automatic type promotion in expressions? Provide a Java code example.**
    - **Answer**: In expressions, Java promotes smaller types (e.g., `byte`, `short`) to `int` or larger types.
        - **Example**:

        ```
        public class Promotion {
            public static void main(String[] args) {
                byte b1 = 10, b2 = 20;
                int result = b1 * b2; // byte promoted to int
                System.out.println("Result: " + result);
            }
        }
        ```

        - **Explanation**: `byte` operands are promoted to `int` during multiplication.
37. **Write a Java program to create and manipulate a two-dimensional array.**
    - **Answer**:

    ```
    public class TwoDArray {
        public static void main(String[] args) {
            int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
            for (int i = 0; i < matrix.length; i++) {
                for (int j = 0; j < matrix[i].length; j++) {
                    System.out.print(matrix[i][j] + " ");
                }
                System.out.println();
            }
        }
    }
    ```

        - **Output**:

        ```
        1 2 3
        4 5 6
        ```

38. **Explain the concept of uneven multidimensional arrays in Java with a code example.**
    - **Answer**: Uneven (jagged) arrays have rows of different lengths.
        - **Example**:

        ```
        public class JaggedArray {
            public static void main(String[] args) {
                int[][] jagged = new int[3][];
                jagged[0] = new int[]{1, 2};
                jagged[1] = new int[]{3, 4, 5};
                jagged[2] = new int[]{6};
                for (int[] row : jagged) {
                    for (int num : row) {
                        System.out.print(num + " ");
                    }
                    System.out.println();
                }
            }
        }
        ```

        - **Output**:

        ```
        1 2
        3 4 5
        6
        ```
39. **Discuss the differences between arrays in Java and C, focusing on memory allocation.**
    - **Answer**:
        - **Java**: Arrays are objects, dynamically allocated on the heap. Size is fixed at creation, and bounds are checked.
        - **C**: Arrays are contiguous memory blocks, allocated on stack or heap. No bounds checking, risking buffer overflows.
        - **Example**: In Java, `int[] arr = new int[5];` creates a heap object. In C, `int arr[5];` is stack-allocated.
        - **Memory**: Java's garbage collector frees arrays; C requires `free()` for heap arrays.
40. **Write a Java program that demonstrates array initialization and iteration using enhanced `for` loops.**
    - **Answer**:

    ```
    public class ArrayDemo {
        public static void main(String[] args) {
            int[] numbers = {10, 20, 30, 40};
            for (int num : numbers) {
                System.out.println("Number: " + num);
            }
        }
    }
    ```

        - **Explanation**: The enhanced `for` loop simplifies iteration over arrays.

Week 5: Operators and Control Statements

41. **Explain the difference between arithmetic, bitwise, relational, and logical operators in Java with examples.**
    - **Answer**:
        - **Arithmetic**: `+`, `-`, `*`, `/`, `%` (e.g., `5 + 3 = 8`).
        - **Bitwise**: `&`, `|`, `^`, `~`, `<<`, `>>` (e.g., `5 & 3 = 1`).
        - **Relational**: `==`, `!=`, `<`, `>`, `<=`, `>=` (e.g., `5 > 3` is `true`).
        - **Logical**: `&&`, `||`, `!` (e.g., `true && false = false`).
        - **Example**:
        - `int a = 5, b = 3;`
        - `System.out.println("Arithmetic: " + (a + b));`
        - `System.out.println("Bitwise: " + (a & b));`
        - `System.out.println("Relational: " + (a > b));`
        - `System.out.println("Logical: " + (a > 0 && b > 0));`
42. **Write a Java program that uses the ternary operator to determine the maximum of three numbers.**
    - **Answer**:

```
public class MaxNumber {
    public static void main(String[] args) {
        int a = 10, b = 20, c = 15;
        int max = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b :
c);
        System.out.println("Maximum: " + max);
    }
}
```

        - **Explanation**: The ternary operator `?:` selects the larger value.
43. **Discuss the precedence and associativity of operators in Java with a complex expression example.**
    - **Answer**: Precedence determines operator evaluation order; associativity defines left-to-right or right-to-left evaluation.
        - **Example**:
        - `int x = 2 + 3 * 4 - 5 / 2; // * and / have higher precedence than + and -`
        - `System.out.println(x); // 3 * 4 = 12, 5 / 2 = 2, 2 + 12 - 2 = 12`
        - **Precedence**: `*`, `/` > `+`, `-`. Associativity: Left-to-right for most operators.
44. **Explain the role of selection, iteration, and jump statements in Java with code snippets.**
    - **Answer**:
        - **Selection**: `if`, `switch` choose code paths.
        - `if (x > 0) System.out.println("Positive");`
        - **Iteration**: `for`, `while`, `do-while` repeat code.
        - `for (int i = 0; i < 3; i++) System.out.println(i);`
        - **Jump**: `break`, `continue`, `return` alter control flow.
        - `for (int i = 0; i < 5; i++) if (i == 3) break; // Exits loop`
45. **Write a Java program that uses `switch` statements to handle multiple user inputs.**
    - **Answer**:

```
public class Menu {
    public static void main(String[] args) {
        int choice = 2; // Simulated input
        switch (choice) {
            case 1:
                System.out.println("Option 1 selected");
                break;
            case 2:
                System.out.println("Option 2 selected");
                break;
            default:
                System.out.println("Invalid choice");
        }
    }
}
```

46. **Demonstrate the use of `break` and `continue` statements in a loop with a practical example.**
    - **Answer**:

```
public class LoopControl {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            if (i == 3) continue; // Skip 3
            if (i == 5) break; // Exit at 5
            System.out.println("Number: " + i);
        }
    }
}
```

- **Output**:

```
Number: 1
Number: 2
Number: 4
```

47. **Explain how Java's bitwise operators can be used for low-level data manipulation.**
    - **Answer**: Bitwise operators manipulate bits:
        - `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), `<<` (left shift), `>>` (right shift).
        - **Example**:

```
int a = 5; // 0101
int b = 3; // 0011
System.out.println("AND: " + (a & b)); // 0001 = 1
System.out.println("Left Shift: " + (a << 1)); // 1010 =
10
```

- **Use**: Optimize flags, masks, or bit-level operations.

48. **Write a Java program that combines logical and relational operators to validate user input.**
    - **Answer**:

```
public class InputValidation {
    public static void main(String[] args) {
        int age = 25;
        if (age >= 18 && age <= 60) {
            System.out.println("Valid age for voting");
        } else {
            System.out.println("Invalid age");
        }
    }
}
```

49. **Discuss the advantages of using `do-while` loops over `while` loops in specific scenarios.**

- **Answer**: `do-while` guarantees at least one execution, unlike `while`.
  - **Scenario**: User input validation.
  - **Example**:

```
int input;
do {
    System.out.println("Enter a positive number:");
    input = 0; // Simulated input
} while (input <= 0);
```

  - **Advantage**: Ensures prompt displays at least once.

50. **Write a Java program that demonstrates nested loops to print a pattern (e.g., a triangle).**
    - **Answer**:

```
public class Pattern {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }
}
```

  - **Output**:

```
*
* *
* * *
* * * *
* * * * *
```

Week 7: Classes & Methods

51. **Explain the role of constructors in Java and demonstrate a parameterized constructor.**
    - **Answer**: Constructors initialize objects, called during creation with `new`.
      - **Example**:

```
class Student {
    String name;
    int id;
    Student(String name, int id) {
        this.name = name;
        this.id = id;
    }
    void display() {
        System.out.println("Name: " + name + ", ID: " +
id);
    }
}
public class Main {
    public static void main(String[] args) {
        Student s = new Student("Alice", 101);
        s.display();
    }
}
```

52. **Write a Java program that uses the `this` keyword to resolve instance variable hiding.**
    - **Answer**:

    ```
    class Box {
        int width;
        Box(int width) {
            this.width = width; // Resolves name conflict
        }
        void display() {
            System.out.println("Width: " + width);
        }
    }
    public class Main {
        public static void main(String[] args) {
            Box box = new Box(10);
            box.display();
        }
    }
    ```

53. **Discuss the purpose of the `new` operator in Java with a code example.**
    - **Answer**: The `new` operator allocates memory for an object and calls its constructor.
        - **Example**:

        ```
        class Car {
            String model;
            Car(String model) { this.model = model; }
        }
        public class Main {
            public static void main(String[] args) {
                Car car = new Car("Toyota");
                System.out.println("Model: " + car.model);
            }
        }
        ```

54. **Explain garbage collection in Java and the role of the `finalize()` method.**
    - **Answer**: Garbage collection reclaims memory from unreachable objects.
        - **finalize()**: Called before an object is collected, for cleanup.
        - **Example**:

        ```
        class Resource {
            protected void finalize() {
                System.out.println("Resource cleaned up");
            }
        }
        public class Main {
            public static void main(String[] args) {
                Resource r = new Resource();
                r = null;
                System.gc(); // Suggest GC
            }
        }
        ```

        - **Note**: `finalize()` is deprecated; use `try-with-resources` instead.

55. **Write a Java program that demonstrates method overloading with different parameter types.**
    - **Answer**:

    ```
    class Calculator {
        int add(int a, int b) { return a + b; }
        double add(double a, double b) { return a + b; }
    }
    public class Main {
        public static void main(String[] args) {
            Calculator calc = new Calculator();
            System.out.println("Int Sum: " + calc.add(2, 3));
            System.out.println("Double Sum: " + calc.add(2.5, 3.5));
        }
    }
    ```

56. **Discuss the differences between instance and static methods in Java with examples.**
    - **Answer**:
        - **Instance Methods**: Require an object, access instance variables.
        - **Static Methods**: Class-level, no object needed, access static variables.
        - **Example**:

        ```
        class Demo {
            int x = 10;
            static int y = 20;
            void instanceMethod() { System.out.println("x: " + x); }
            static void staticMethod() { System.out.println("y: " + y); }
        }
        public class Main {
            public static void main(String[] args) {
                Demo.staticMethod();
                Demo obj = new Demo();
                obj.instanceMethod();
            }
        }
        ```

57. **Explain how Java handles object creation and destruction compared to C++.**
    - **Answer**:
        - **Java**: Objects created with `new`, destroyed by garbage collection. No manual deallocation.
        - **C++**: Objects created with `new`, destroyed with `delete`. Manual memory management.
        - **Example**: In Java, `Object o = new Object(); o = null;` allows GC. In C++, `Object* o = new Object(); delete o;`.
        - **Difference**: Java is safer; C++ offers control but risks leaks.

58. **Write a Java program that uses a constructor to initialize an object's state.**
    - **Answer**:

    ```
    class Employee {
        String name;
        int salary;
        Employee(String name, int salary) {
            this.name = name;
            this.salary = salary;
        }
    ```

```
    void display() {
        System.out.println(name + ": " + salary);
    }
}
public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee("Bob", 50000);
        emp.display();
    }
}
```

59. **Demonstrate the use of access specifiers (`public`, `private`, `protected`) in a Java class.**
    - **Answer**:

```
class AccessDemo {
    public int publicVar = 1;
    private int privateVar = 2;
    protected int protectedVar = 3;
    void display() {
        System.out.println("Public: " + publicVar + ", Private: " + privateVar + ", Protected: " + protectedVar);
    }
}
public class Main {
    public static void main(String[] args) {
        AccessDemo obj = new AccessDemo();
        obj.display();
    }
}
```

60. **Explain the concept of recursion in Java with a program to calculate factorial.**
    - **Answer**:

```
class Factorial {
    int fact(int n) {
        if (n <= 1) return 1;
        return n * fact(n - 1);
    }
}
public class Main {
    public static void main(String[] args) {
        Factorial f = new Factorial();
        System.out.println("Factorial of 5: " + f.fact(5));
    }
}
```

- **Explanation**: `fact(5)` computes `5 * fact(4)`, recursively until `fact(1)`.

Week 8: Encapsulation

61. **Define encapsulation and explain how it is achieved in Java with access specifiers.**
    - **Answer**: Encapsulation hides data, exposing it via methods. Achieved with `private` fields and `public` getters/setters.
        - **Example**:

```
class Person {
    private String name;
```

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

62. **Write a Java program that demonstrates encapsulation using getter and setter methods.**
    - **Answer**:

    ```
    class Book {
        private String title;
        public String getTitle() { return title; }
        public void setTitle(String title) { this.title = title; }
    }
    public class Main {
        public static void main(String[] args) {
            Book book = new Book();
            book.setTitle("Java Guide");
            System.out.println("Title: " + book.getTitle());
        }
    }
    ```

63. **Discuss the benefits of encapsulation in terms of data security and code maintenance.**
    - **Answer**:
        - **Security**: Prevents direct access to fields, ensuring valid data via setters.
        - **Maintenance**: Changes to fields don't affect external code using getters/setters.
        - **Example**: A `private balance` in a `BankAccount` class ensures deposits are positive.

64. **Explain the difference between `public`, `private`, and `protected` access specifiers with examples.**
    - **Answer**:
        - **public**: Accessible everywhere.
        - **private**: Accessible only within the class.
        - **protected**: Accessible in the class, subclasses, and same package.
        - **Example**:

        ```
        class Test {
            public int a = 1;
            private int b = 2;
            protected int c = 3;
        }
        ```

65. **Write a Java program that uses encapsulation to model a bank account with private fields.**
    - **Answer**:

    ```
    class BankAccount {
        private double balance;
        public void deposit(double amount) {
            if (amount > 0) balance += amount;
        }
        public double getBalance() { return balance; }
    }
    public class Main {
        public static void main(String[] args) {
            BankAccount acc = new BankAccount();
            acc.deposit(1000);
    ```

```
            System.out.println("Balance: " + acc.getBalance());
        }
    }
```

66. **Discuss how encapsulation prevents unauthorized access to an object's state.**
    - **Answer**: Encapsulation uses `private` fields, accessible only via controlled methods, preventing invalid modifications.
        - **Example**: A `private password` field is only set via a `setPassword` method with validation.
67. **Explain the role of references in Java and how they differ from pointers in C.**
    - **Answer**:
        - **Java References**: Point to objects, managed by the JVM, no direct memory access.
        - **C Pointers**: Store memory addresses, allow direct manipulation.
        - **Example**: In Java, `String s = "test";` is a reference. In C, `char *s = "test";` is a pointer.
        - **Difference**: Java is safer, preventing pointer arithmetic errors.
68. **Write a Java program that demonstrates passing objects as parameters to methods.**
    - **Answer**:

```
class Point {
    int x;
    Point(int x) { this.x = x; }
}
class Test {
    void modify(Point p) { p.x += 10; }
}
public class Main {
    public static void main(String[] args) {
        Point p = new Point(5);
        Test t = new Test();
        t.modify(p);
        System.out.println("x: " + p.x);
    }
}
```

69. **Discuss the concept of nested classes in Java with a practical example.**
    - **Answer**: Nested classes are defined within another class.
        - **Example**:

```
class Outer {
    class Inner {
        void display() { System.out.println("Inner class"); }
    }
}
public class Main {
    public static void main(String[] args) {
        Outer.Inner inner = new Outer().new Inner();
        inner.display();
    }
}
```

70. **Explain the difference between static and non-static nested classes in Java.**
    - **Answer**:

    - **Static Nested**: Associated with the outer class, no outer instance needed.
    - **Non-Static Nested (Inner)**: Requires an outer class instance.
    - **Example**:

      ```
      class Outer {
          static class StaticNested { void show() { System.out.println("Static"); } }
          class Inner { void show() { System.out.println("Inner"); } }
      }
      ```

Week 9: Inheritance

71. **Define inheritance and explain its advantages in Java with a real-world example.**
    - **Answer**: Inheritance allows a class to inherit properties from another.
        - **Advantages**: Code reuse, extensibility.
        - **Example**:

          ```
          class Vehicle { void move() { System.out.println("Moving"); } }
          class Car extends Vehicle { void honk() { System.out.println("Honk"); } }
          ```
72. **Write a Java program that demonstrates single inheritance using the `extends` keyword.**
    - **Answer**:

      ```
      class Animal { void eat() { System.out.println("Eating"); } }
      class Dog extends Animal { void bark() { System.out.println("Barking"); } }
      public class Main {
          public static void main(String[] args) {
              Dog d = new Dog();
              d.eat();
              d.bark();
          }
      }
      ```
73. **Explain the role of the `super` keyword in Java with a program that uses it.**
    - **Answer**: `super` accesses parent class members or constructors.
        - **Example**:

          ```
          class Parent { int x = 10; }
          class Child extends Parent {
              int x = 20;
              void display() { System.out.println("Super x: " + super.x); }
          }
          public class Main {
              public static void main(String[] args) {
                  Child c = new Child();
                  c.display();
              }
          }
          ```
74. **Discuss multilevel inheritance in Java with a code example involving three classes.**
    - **Answer**:

      ```
      class A { void methodA() { System.out.println("A"); } }
      ```

```java
class B extends A { void methodB() { System.out.println("B"); } }
class C extends B { void methodC() { System.out.println("C"); } }
public class Main {
    public static void main(String[] args) {
        C c = new C();
        c.methodA();
        c.methodB();
        c.methodC();
    }
}
```

75. **Write a Java program that demonstrates method overriding in an inheritance hierarchy.**
    - **Answer**:

    ```java
    class Animal { void sound() { System.out.println("Some sound"); } }
    class Cat extends Animal { void sound() { System.out.println("Meow"); } }
    public class Main {
        public static void main(String[] args) {
            Animal cat = new Cat();
            cat.sound();
        }
    }
    ```

76. **Explain how constructors are called in an inheritance hierarchy with a code example.**
    - **Answer**: Parent class constructors are called before subclass constructors using `super()`.
        - **Example**:

        ```java
        class Parent {
            Parent() { System.out.println("Parent Constructor"); }
        }
        class Child extends Parent {
            Child() { System.out.println("Child Constructor"); }
        }
        public class Main {
            public static void main(String[] args) {
                Child c = new Child();
            }
        }
        ```

77. **Discuss the limitations of inheritance in Java, such as single inheritance.**
    - **Answer**:
        - **Single Inheritance**: Java allows a class to inherit from only one parent, unlike C++.
        - **Workaround**: Use interfaces for multiple inheritance.
        - **Example**: A class cannot extend two classes but can implement multiple interfaces.

78. **Write a Java program that uses inheritance to model a vehicle class hierarchy.**
    - **Answer**:

    ```java
    class Vehicle { void move() { System.out.println("Moving"); } }
    ```

- `class Car extends Vehicle { void drive() { System.out.println("Driving car"); } }`
- `class Bike extends Vehicle { void ride() { System.out.println("Riding bike"); } }`
- `public class Main {`
- `    public static void main(String[] args) {`
- `        Car car = new Car();`
- `        Bike bike = new Bike();`
- `        car.move();`
- `        car.drive();`
- `        bike.move();`
- `        bike.ride();`
- `    }`
- `}`

79. **Explain the difference between method overriding and method overloading with examples.**
    - **Answer**:
        - **Overriding**: Subclass redefines a parent class method.
        - `class A { void show() { System.out.println("A"); } }`
        - `class B extends A { void show() { System.out.println("B"); } }`
        - **Overloading**: Same method name, different parameters.
        - `class C {`
        - `    void show(int x) { System.out.println("Int"); }`
        - `    void show(String s) { System.out.println("String"); }`
        - `}`
80. **Discuss how inheritance promotes code reusability and extensibility in Java.**
    - **Answer**: Inheritance reuses parent class code and allows subclasses to extend functionality.
        - **Example**: A `Shape` class defines `draw()`. Subclasses like `Circle` reuse and specialize it.
        - **Benefit**: Reduces duplication, supports adding new subclasses.

Week 10: Polymorphism

81. **Define polymorphism and explain its types (compile-time and runtime) in Java.**
    - **Answer**:
        - **Polymorphism**: Objects of different types can be treated as a common type.
        - **Compile-Time**: Method overloading, resolved at compile time.
        - **Runtime**: Method overriding, resolved at runtime via dynamic dispatch.
        - **Example**:
        - `class Test {`
        - `    void show(int x) { System.out.println("Int"); }`
        - `    void show(String s) { System.out.println("String"); } // Overloading`
        - `}`
82. **Write a Java program that demonstrates method overloading (compile-time polymorphism).**
    - **Answer**:

```
class Overload {
    void display(int x) { System.out.println("Int: " + x); }
    void display(String s) { System.out.println("String: " + s);
}
}
public class Main {
    public static void main(String[] args) {
        Overload obj = new Overload();
        obj.display(5);
        obj.display("Test");
    }
}
```

83. **Explain runtime polymorphism in Java with a program using method overriding.**
    - **Answer**:

    ```
    class Animal { void sound() { System.out.println("Sound"); } }
    class Dog extends Animal { void sound() {
    System.out.println("Woof"); } }
    public class Main {
        public static void main(String[] args) {
            Animal animal = new Dog();
            animal.sound(); // Woof
        }
    }
    ```

84. **Discuss the role of abstract classes in achieving polymorphism in Java.**
    - **Answer**: Abstract classes define methods for subclasses to implement, enabling polymorphism.
        - **Example**:

        ```
        abstract class Shape { abstract void draw(); }
        class Circle extends Shape { void draw() {
        System.out.println("Circle"); } }
        ```

85. **Write a Java program that uses an abstract class to model a shape hierarchy.**
    - **Answer**:

    ```
    abstract class Shape { abstract double area(); }
    class Rectangle extends Shape {
        double width, height;
        Rectangle(double w, double h) { width = w; height = h; }
        double area() { return width * height; }
    }
    public class Main {
        public static void main(String[] args) {
            Shape s = new Rectangle(5, 3);
            System.out.println("Area: " + s.area());
        }
    }
    ```

86. **Explain the use of the `final` keyword in preventing method overriding and inheritance.**
    - **Answer**:
        - **Final Method**: Cannot be overridden.
        - **Final Class**: Cannot be extended.
        - **Example**:

        ```
        class Parent {
            final void show() { System.out.println("Parent"); }
        }
        ```

        - `final class FinalClass { }`

87. **Discuss how polymorphism improves code flexibility in large-scale applications.**
    - **Answer**: Polymorphism allows new subclasses to be added without modifying existing code, using a common interface.
        - **Example**: A payment system can process `CreditCard` or `PayPal` via a `Payment` interface.
88. **Write a Java program that demonstrates polymorphic behavior using interfaces.**
    - **Answer**:

```
interface Printable { void print(); }
class Document implements Printable {
    public void print() { System.out.println("Printing document"); }
}
public class Main {
    public static void main(String[] args) {
        Printable p = new Document();
        p.print();
    }
}
```

89. **Explain the difference between abstract classes and interfaces in achieving polymorphism.**
    - **Answer**:
        - **Abstract Classes**: Can have implemented methods, single inheritance.
        - **Interfaces**: Only method signatures, multiple inheritance.
        - **Example**: An abstract `Vehicle` has `move()`. An interface `Drivable` defines `drive()`.
90. **Discuss the advantages and disadvantages of polymorphism in Java programming.**
    - **Answer**:
        - **Advantages**: Flexibility, extensibility, reusable code.
        - **Disadvantages**: Can increase complexity, slight performance overhead due to dynamic dispatch.
        - **Example**: Polymorphic code handles new types but may require careful design.

Week 11: Packages & Interfaces

91. **Explain the purpose of packages in Java and how they organize code.**
    - **Answer**: Packages group related classes, preventing name conflicts and improving modularity.
        - **Example**: `java.util` contains utility classes like `ArrayList`.
92. **Write a Java program that creates and uses a custom package.**
    - **Answer**:

```
// File: mypackage/MyClass.java
package mypackage;
public class MyClass {
    public void show() { System.out.println("Custom package"); }
}
// File: Main.java
import mypackage.MyClass;
```

```
public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.show();
    }
}
```

93. **Discuss the process of importing packages in Java with examples of `import` statements.**
    - **Answer**:
        - **Import Single Class**: `import java.util.ArrayList;`
        - **Import Package**: `import java.util.*;`
        - **Example**:

        ```
        import java.util.ArrayList;
        public class Main {
            public static void main(String[] args) {
                ArrayList<String> list = new ArrayList<>();
                list.add("Test");
            }
        }
        ```

94. **Explain the concept of interfaces in Java and their role in achieving abstraction.**
    - **Answer**: Interfaces define method signatures for classes to implement, hiding implementation details.
        - **Example**:

        ```
        interface Movable { void move(); }
        class Car implements Movable { public void move() { System.out.println("Car moves"); } }
        ```

95. **Write a Java program that defines and implements an interface with multiple methods.**
    - **Answer**:

    ```
    interface Vehicle {
        void start();
        void stop();
    }
    class Bike implements Vehicle {
        public void start() { System.out.println("Bike started"); }
        public void stop() { System.out.println("Bike stopped"); }
    }
    public class Main {
        public static void main(String[] args) {
            Bike bike = new Bike();
            bike.start();
            bike.stop();
        }
    }
    ```

96. **Discuss how interfaces support multiple inheritance in Java with a code example.**
    - **Answer**:

    ```
    interface A { void methodA(); }
    interface B { void methodB(); }
    class Test implements A, B {
        public void methodA() { System.out.println("A"); }
        public void methodB() { System.out.println("B"); }
    }
    public class Main {
    ```

```
o      public static void main(String[] args) {
o          Test t = new Test();
o          t.methodA();
o          t.methodB();
o      }
o  }
```

97. **Explain the difference between partial and complete implementation of an interface.**
    - **Answer**:
        - **Complete**: All methods implemented.
        - **Partial**: Some methods unimplemented, class must be abstract.
        - **Example**:
        - `interface Test { void a(); void b(); }`
        - `abstract class Partial implements Test { public void a() { } }`
98. **Write a Java program that demonstrates extending an interface.**
    - **Answer**:

```
o  interface Base { void baseMethod(); }
o  interface Extended extends Base { void extendedMethod(); }
o  class Impl implements Extended {
o      public void baseMethod() { System.out.println("Base"); }
o      public void extendedMethod() {
   System.out.println("Extended"); }
o  }
```

99. **Discuss the benefits of using packages for code modularity and reusability.**
    - **Answer**:
        - **Modularity**: Groups related classes.
        - **Reusability**: Packages can be shared across projects.
        - **Example**: `java.util` provides reusable utilities.
100. **Explain how interfaces enable polymorphic behavior in Java applications.**
    - **Answer**: Interfaces allow objects of different classes to be treated as the interface type.
        - **Example**:
        - `interface Drawable { void draw(); }`
        - `class Circle implements Drawable { public void draw() { System.out.println("Circle"); } }`

Week 12: Exception Handling

101. **Define exception handling in Java and explain its importance in robust programming.**
    - **Answer**: Exception handling manages runtime errors, preventing crashes.
        - **Importance**: Ensures graceful error recovery.
        - **Example**: Catching a `FileNotFoundException` when opening a file.
102. **Write a Java program that demonstrates the use of `try` and `catch` blocks.**
    - **Answer**:

```
o  public class ExceptionDemo {
o      public static void main(String[] args) {
o          try {
o              int x = 10 / 0;
o           } catch (ArithmeticException e) {
o               System.out.println("Error: " + e.getMessage());
o           }
o       }
o   }
```

103. **Explain the difference between checked and unchecked exceptions in Java with examples.**

- **Answer**:
    - **Checked**: Must be declared or caught (e.g., `IOException`).
    - **Unchecked**: Runtime exceptions, optional handling (e.g., `NullPointerException`).
    - **Example**:

```
▪   try { new java.io.FileInputStream("file.txt"); } catch
    (java.io.FileNotFoundException e) { }
▪   int x = Integer.parseInt("abc"); // NumberFormatException
```

104. **Write a Java program that uses the `throw` keyword to create a custom exception.**

- **Answer**:

```
o   class CustomException extends Exception {
o       CustomException(String msg) { super(msg); }
o   }
o   public class Main {
o       public static void main(String[] args) {
o           try {
o               throw new CustomException("Custom error");
o           } catch (CustomException e) {
o               System.out.println("Caught: " + e.getMessage());
o           }
o       }
o   }
```

105. **Discuss the role of the `throws` keyword in method declarations with a code example.**

- **Answer**: `throws` declares exceptions a method may throw.
    - **Example**:

```
▪   class Test {
▪       void method() throws IOException {
▪           throw new IOException("IO error");
▪       }
▪   }
```

106. **Explain the purpose of the `finally` block in exception handling with a practical example.**

- **Answer**: `finally` executes regardless of exception, for cleanup.
    - **Example**:

```
▪   public class FinallyDemo {
▪       public static void main(String[] args) {
▪           try {
▪               System.out.println("Try");
▪           } catch (Exception e) {
▪               System.out.println("Catch");
▪           } finally {
▪               System.out.println("Finally");
▪           }
```

▪ }
▪ }

107. **Write a Java program that handles multiple exceptions in a single `try` block.**

- **Answer**:

```
public class MultiException {
    public static void main(String[] args) {
        try {
            int[] arr = new int[2];
            arr[10] = 5; // ArrayIndexOutOfBoundsException
            int x = Integer.parseInt("abc"); //
NumberFormatException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array error");
        } catch (NumberFormatException e) {
            System.out.println("Number error");
        }
    }
}
```

108. **Discuss the hierarchy of exception classes in Java, including `Throwable`, `Error`, and `Exception`.**

- **Answer**:
  - **Throwable**: Root class for all errors and exceptions.
  - **Error**: Serious issues (e.g., `OutOfMemoryError`).
  - **Exception**: Recoverable issues (e.g., `IOException`, `RuntimeException`).
  - **Example**: `IOException` is a checked `Exception`; `NullPointerException` is an unchecked `RuntimeException`.

109. **Explain how exception handling in Java differs from error handling in C.**

- **Answer**:
  - **Java**: Structured with `try-catch`, automatic stack unwinding.
  - **C**: Manual error codes or `setjmp/longjmp`, less robust.
  - **Example**: Java catches `IOException`; C checks `fopen` return value.

110. **Write a Java program that demonstrates chaining exceptions using `initCause()`.**

- **Answer**:

```
public class ExceptionChain {
    public static void main(String[] args) {
        try {
            try {
                throw new IOException("IO error");
            } catch (IOException e) {
                throw new RuntimeException("Wrapped", e);
            }
        } catch (RuntimeException e) {
            System.out.println("Cause: " + e.getCause());
        }
    }
}
```

Week 13: File I/O

111. **Explain the concept of streams in Java and differentiate between byte and character streams.**
- **Answer**:
  - **Byte Streams**: Handle raw bytes (e.g., `FileInputStream`).
  - **Character Streams**: Handle Unicode characters (e.g., `FileReader`).
  - **Example**: `FileInputStream` for binary files, `FileReader` for text.

112. **Write a Java program that writes data to a file using `FileOutputStream`.**
- **Answer**:

```
import java.io.*;
public class WriteFile {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream("output.txt");
        String data = "Hello, Java!";
        fos.write(data.getBytes());
        fos.close();
    }
}
```

113. **Write a Java program that reads data from a file using `FileInputStream`.**
- **Answer**:

```
import java.io.*;
public class ReadFile {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("output.txt");
        int byteData;
        while ((byteData = fis.read()) != -1) {
            System.out.print((char) byteData);
        }
        fis.close();
    }
}
```

114. **Discuss the advantages of using `BufferedReader` over `FileReader` for text files.**
- **Answer**:
  - **BufferedReader**: Buffers input, reducing system calls, faster for large files.
  - **FileReader**: Reads directly, slower for frequent access.
  - **Example**: `BufferedReader` reads lines efficiently with `readLine()`.

115. **Write a Java program that appends data to an existing file using `FileWriter`.**
- **Answer**:

```
import java.io.*;
public class AppendFile {
    public static void main(String[] args) throws IOException {
        FileWriter fw = new FileWriter("output.txt", true);
        fw.write("\nAppended text");
        fw.close();
    }
}
```

116. **Explain the role of the `Serializable` interface in Java file I/O with a code example.**

- **Answer**: `Serializable` enables object serialization to files.
  - **Example**:

```
import java.io.*;
class Student implements Serializable {
    String name;
    Student(String name) { this.name = name; }
}
public class Main {
    public static void main(String[] args) throws IOException {
        Student s = new Student("Alice");
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("student.ser"));
        oos.writeObject(s);
        oos.close();
    }
}
```

117. **Write a Java program that copies the contents of one file to another.**
    - **Answer**:

```
import java.io.*;
public class CopyFile {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("source.txt");
        FileOutputStream fos = new FileOutputStream("dest.txt");
        int byteData;
        while ((byteData = fis.read()) != -1) {
            fos.write(byteData);
        }
        fis.close();
        fos.close();
    }
}
```

118. **Discuss the differences between file I/O in Java and C with respect to error handling.**
    - **Answer**:
      - **Java**: Uses exceptions (e.g., `IOException`) for errors.
      - **C**: Uses return codes (e.g., `NULL` from `fopen`) or `errno`.
      - **Example**: Java throws `FileNotFoundException`; C checks `fopen` result.

119. **Explain how Java handles file permissions and security during I/O operations.**
    - **Answer**: Java respects OS file permissions. `File` class methods like `canRead()` check access.
      - **Example**: Attempting to read a restricted file throws `SecurityException`.

120. **Write a Java program that reads and displays the contents of a CSV file.**
    - **Answer**:

```
import java.io.*;
public class ReadCSV {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader("data.csv"));
        String line;
```

```
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
            br.close();
        }
    }
```

Week 14: Graphical User Interface

121. **Explain the difference between AWT and Swing in Java GUI programming.**
    - **Answer**:
        - **AWT**: Native components, platform-dependent, lightweight.
        - **Swing**: Pure Java, platform-independent, richer components.
        - **Example**: `JFrame` (Swing) vs. `Frame` (AWT).

122. **Write a Java program that creates a simple GUI using `JFrame` and `JButton`.**
    - **Answer**:

```
import javax.swing.*;
public class SimpleGUI {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Simple GUI");
        JButton button = new JButton("Click Me");
        frame.add(button);
        frame.setSize(200, 100);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

123. **Discuss the role of layout managers in Java GUI design with examples of `FlowLayout` and `GridLayout`.**
    - **Answer**:
        - **FlowLayout**: Arranges components in a row.
        - **GridLayout**: Arranges components in a grid.
        - **Example**:

```
import javax.swing.*;
import java.awt.*;
public class LayoutDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Layouts");
        frame.setLayout(new FlowLayout());
        frame.add(new JButton("Button 1"));
        frame.add(new JButton("Button 2"));
        frame.setSize(200, 100);
        frame.setVisible(true);
    }
}
```

124. **Write a Java program that demonstrates event handling with a button click.**
    - **Answer**:

```
import javax.swing.*;
import java.awt.event.*;
public class ButtonEvent {
```

```
o       public static void main(String[] args) {
o           JFrame frame = new JFrame("Button Event");
o           JButton button = new JButton("Click");
o           button.addActionListener(e -> System.out.println("Button
   clicked"));
o           frame.add(button);
o           frame.setSize(200, 100);
o           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
o           frame.setVisible(true);
o       }
o   }
```

125. **Explain the steps to create a Java GUI application using Swing components.**
   - **Answer**:
     - Create a `JFrame`.
     - Set layout manager.
     - Add components (e.g., `JButton`, `JLabel`).
     - Attach event listeners.
     - Set frame properties (size, visibility).
     - **Example**: See question 124.

126. **Write a Java program that uses `BorderLayout` to arrange multiple components.**
   - **Answer**:

```
o   import javax.swing.*;
o   import java.awt.*;
o   public class BorderLayoutDemo {
o       public static void main(String[] args) {
o           JFrame frame = new JFrame("BorderLayout");
o           frame.setLayout(new BorderLayout());
o           frame.add(new JButton("North"), BorderLayout.NORTH);
o           frame.add(new JButton("South"), BorderLayout.SOUTH);
o           frame.setSize(300, 200);
o           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
o           frame.setVisible(true);
o       }
o   }
```

127. **Discuss the advantages of using Swing over AWT for modern GUI applications.**
   - **Answer**:
     - **Portability**: Swing is platform-independent.
     - **Rich Components**: More widgets (e.g., `JTable`, `JTree`).
     - **Customizable**: Look-and-feel support.
     - **Example**: Swing's `JButton` looks consistent across OSes.

128. **Explain the concept of composition vs. inheritance in Java GUI design with examples.**
   - **Answer**:
     - **Composition**: Add components to a container.
     - **Inheritance**: Extend `JFrame`.
     - **Example**: