

# Subject Name Solutions

4341602 – Summer 2023

Semester 1 Study Material

*Detailed Solutions and Explanations*

## Question 1(a) [3 marks]

Differentiate between Procedure-Oriented Programming (POP) and Object-Oriented Programming (OOP).

Solution		
Aspect	POP	OOP
Focus	Functions/Procedures	Objects and Classes
Data Security	Less secure, global data	More secure, data encapsulation
Problem Solving	Top-down approach	Bottom-up approach
Code Reusability	Limited	High through inheritance
Examples	C, Pascal	Java, C++, Python
<ul style="list-style-type: none"><li>• <b>POP:</b> Program divided into functions, data flows between functions</li><li>• <b>OOP:</b> Program organized around objects that contain both data and methods</li></ul>		

## Mnemonic

“POP Functions, OOP Objects”

## Question 1(b) [4 marks]

Explain Super keyword in inheritance with suitable example.

Solution		
<b>Super keyword</b> is used to access parent class members from child class.		
Table 1: Super keyword uses		
Use	Purpose	Example
<b>super()</b>	Call parent constructor	super(name, age)
<b>super.method()</b>	Call parent method	super.display()
<b>super.variable</b>	Access parent variable	super.name

### Code Block:

```
1 class Animal {
2     String name = "Animal";
3     void eat() { System.out.println("Animal eats"); }
4 }
5
6 class Dog extends Animal {
7     String name = "Dog";
8     void eat() {
9         super.eat(); // calls parent method
10        System.out.println("Dog eats bones");
11    }
12    void display() {
13        System.out.println(super.name); // prints "Animal"
14    }
15 }
```

### Mnemonic

“Super calls Parent”

### Question 1(c) [7 marks]

Define: Method Overriding. List out Rules for method overriding. Write a java program that implements method overriding.

### Solution

**Method Overriding:** Child class provides specific implementation of parent class method with same signature.

Table 2: Method Overriding Rules

Rule	Description
<b>Same name</b>	Method name must be identical
<b>Same parameters</b>	Parameter list must match exactly
<b>IS-A relationship</b>	Must have inheritance
<b>Access modifier</b>	Cannot reduce visibility
<b>Return type</b>	Must be same or covariant

### Code Block:

```
1 class Shape {
2     void draw() {
3         System.out.println("Drawing a shape");
4     }
5 }
6
7 class Circle extends Shape {
8     @Override
9     void draw() {
10        System.out.println("Drawing a circle");
11    }
12 }
13
14 class Main {
15     public static void main(String[] args) {
16         Shape s = new Circle();
17         s.draw(); // Output: Drawing a circle
18     }
19 }
```

### Mnemonic

“Override Same Signature”

## Question 1(c OR) [7 marks]

**Describe: Interface. Write a java program using interface to demonstrate multiple inheritance.**

### Solution

**Interface:** Blueprint containing abstract methods and constants. Classes implement interfaces to achieve multiple inheritance.

Table 3: Interface Features

Feature	Description
<b>Abstract methods</b>	No implementation (before Java 8)
<b>Constants</b>	All variables are public static final
<b>Multiple inheritance</b>	Class can implement multiple interfaces
<b>Default methods</b>	Concrete methods (Java 8+)

#### Code Block:

```
1 interface Flyable {
2     void fly();
3 }
4
5 interface Swimmable {
6     void swim();
7 }
8
9 class Duck implements Flyable, Swimmable {
10     public void fly() {
11         System.out.println("Duck flies");
12     }
13
14     public void swim() {
15         System.out.println("Duck swims");
16     }
17 }
18
19 class Main {
20     public static void main(String[] args) {
21         Duck d = new Duck();
22         d.fly();
23         d.swim();
24     }
25 }
```

#### Mnemonic

“Interface Multiple Implementation”

### Question 2(a) [3 marks]

Explain the Java Program Structure with example.

#### Solution

**Java Program Structure** consists of package, imports, class declaration, and main method.  
**Diagram:**

```
1 +-----+
2 | Package statement|
3 +-----+
4 | Import statements|
5 +-----+
6 | Class declaration|
7 | +-----+ |
8 | | Variables  | |
9 | +-----+ |
10 | | Methods    | |
11 | +-----+ |
12 +-----+
```

#### Code Block:

```
1 package com.example;           // Package
2 import java.util.*;            // Import
3
4 public class HelloWorld {       // Class
5     static int count = 0;       // Variable
6
7     public static void main(String[] args) { // Main method
8         System.out.println("Hello World");
9     }
10 }
```

```
9     }  
0 }
```

#### Mnemonic

“Package Import Class Main”

### Question 2(b) [4 marks]

Explain static keyword with suitable example.

#### Solution

**Static keyword** belongs to class rather than instance. Memory allocated once.

Table 4: Static Uses

Type	Description	Example
<b>Static variable</b>	Shared by all objects	static int count
<b>Static method</b>	Called without object	static void display()
<b>Static block</b>	Executes before main	static { }

#### Code Block:

```
1 class Student {  
2     static String college = "GTU"; // static variable  
3     String name;  
4  
5     static void showCollege() { // static method  
6         System.out.println(college);  
7     }  
8  
9     static { // static block  
0         System.out.println("Static block executed");  
1     }  
2 }  
3  
4 class Main {  
5     public static void main(String[] args) {  
6         Student.showCollege(); // No object needed  
7     }  
8 }
```

#### Mnemonic

“Static Shared by Class”

### Question 2(c) [7 marks]

Define: Constructor. List out types of it. Explain Parameterized and copy constructor with suitable example.

#### Solution

**Constructor:** Special method to initialize objects, same name as class, no return type.

Table 5: Constructor Types

Type	Description	Example
<b>Default</b>	No parameters	Student()
<b>Parameterized</b>	With parameters	Student(String name)
<b>Copy</b>	Creates copy of object	Student(Student s)

#### Code Block:

```

1  class Student {
2      String name;
3      int age;
4
5      // Parameterized constructor
6      Student(String n, int a) {
7          name = n;
8          age = a;
9      }
10
11     // Copy constructor
12     Student(Student s) {
13         name = s.name;
14         age = s.age;
15     }
16
17     void display() {
18         System.out.println(name + " " + age);
19     }
20 }
21
22 class Main {
23     public static void main(String[] args) {
24         Student s1 = new Student("John", 20); // Parameterized
25         Student s2 = new Student(s1);         // Copy
26         s1.display();
27         s2.display();
28     }
29 }

```

#### Mnemonic

“Constructor Initializes Objects”

### Question 2(a OR) [3 marks]

Explain the Primitive Data Types and User Defined Data Types in java.

#### Solution

**Primitive Data Types:** Built-in types provided by Java language. **User Defined Types:** Custom types created by programmer using classes.

Table 6: Data Types

Category	Types	Size	Example
<b>Primitive</b>	byte, short, int, long	1,2,4,8 bytes	int x = 10;
<b>Primitive</b>	float, double	4,8 bytes	double d = 3.14;
<b>Primitive</b>	char, boolean	2,1 bytes	char c = 'A';
<b>User Defined</b>	Class, Interface, Array	Variable	Student s;

- **Primitive:** Stored in stack, faster access
- **User Defined:** Stored in heap, complex operations

### Mnemonic

“Primitive Built-in, User Custom”

### Question 2(b OR) [4 marks]

Explain this keyword with suitable example.

#### Solution

This keyword refers to current object instance, used to distinguish between instance and local variables.

Table 7: This keyword uses

Use	Purpose	Example
<b>this.variable</b>	Access instance variable	this.name = name;
<b>this.method()</b>	Call instance method	this.display();
<b>this()</b>	Call constructor	this(name, age);

#### Code Block:

```
1 class Student {
2     String name;
3     int age;
4
5     Student(String name, int age) {
6         this.name = name;    // this distinguishes
7         this.age = age;      // instance from parameter
8     }
9
10    void setData(String name) {
11        this.name = name;    // this refers to current object
12    }
13
14    void display() {
15        System.out.println(this.name + " " + this.age);
16    }
17 }
```

### Mnemonic

“This Current Object”

### Question 2(c OR) [7 marks]

Define Inheritance. List out types of it. Explain multilevel and hierarchical inheritance with suitable example.

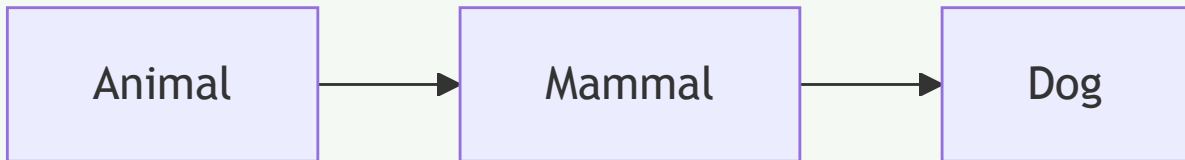
#### Solution

**Inheritance:** Mechanism where child class acquires properties and methods of parent class.

Table 8: Inheritance Types

Type	Description	Structure
<b>Single</b>	One parent, one child	$A \rightarrow B$
<b>Multilevel</b>	Chain of inheritance	$A \rightarrow B \rightarrow C$
<b>Hierarchical</b>	One parent, multiple children	$A \rightarrow B, A \rightarrow C$
<b>Multiple</b>	Multiple parents (via interfaces)	$B, C \rightarrow A$

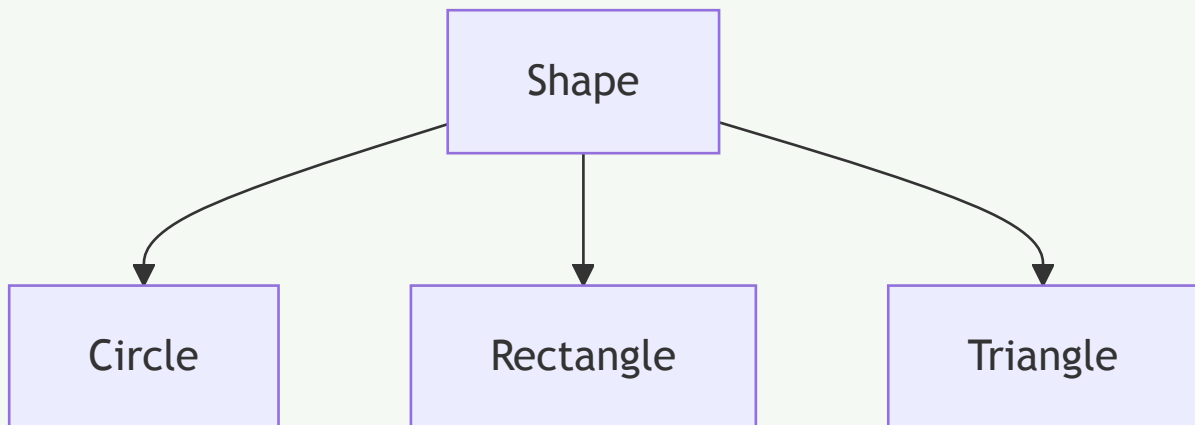
Diagram - Multilevel:



Code Block - Multilevel:

```
1 class Animal {  
2     void eat() { System.out.println("Animal eats"); }  
3 }  
4  
5 class Mammal extends Animal {  
6     void breathe() { System.out.println("Mammal breathes"); }  
7 }  
8  
9 class Dog extends Mammal {  
0     void bark() { System.out.println("Dog barks"); }  
1 }
```

Diagram - Hierarchical:



Code Block - Hierarchical:

```
1 class Shape {  
2     void draw() { System.out.println("Drawing shape"); }  
3 }  
4  
5 class Circle extends Shape {  
6     void drawCircle() { System.out.println("Drawing circle"); }  
7 }  
8  
9 class Rectangle extends Shape {  
0     void drawRectangle() { System.out.println("Drawing rectangle"); }  
1 }
```

### Mnemonic

“Inheritance Shares Properties”

### Question 3(a) [3 marks]

Explain Type Conversion and Casting in java.

## Solution

**Type Conversion:** Converting one data type to another. **Casting:** Explicit type conversion by programmer.

Table 9: Type Conversion

Type	Description	Example
<b>Implicit (Widening)</b>	Automatic, smaller to larger	int to double
<b>Explicit (Narrowing)</b>	Manual, larger to smaller	double to int

### Code Block:

```
1 // Implicit conversion
2 int i = 10;
3 double d = i;           // int to double (automatic)
4
5 // Explicit casting
6 double x = 10.5;
7 int y = (int) x;        // double to int (manual)
8
9 // String conversion
10 String s = String.valueOf(i); // int to String
11 int z = Integer.parseInt("123"); // String to int
```

## Mnemonic

“Implicit Auto, Explicit Manual”

## Question 3(b) [4 marks]

Explain different visibility controls used in Java.

## Solution

**Visibility Controls (Access Modifiers):** Control access to classes, methods, and variables.

Table 10: Access Modifiers

Modifier	Same Class	Same Package	Subclass	Different Package
<b>private</b>				
<b>default</b>				
<b>protected</b>				
<b>public</b>				

### Code Block:

```
1 class Example {
2     private int x = 10;    // Only within class
3     int y = 20;           // Package level
4     protected int z = 30; // Package + subclass
5     public int w = 40;     // Everywhere
6
7     private void method1() { } // Private method
8     public void method2() { }  // Public method
9 }
```

## Mnemonic

“Private Package Protected Public”

### Question 3(c) [7 marks]

Define: Thread. List different methods used to create Thread. Explain Thread life cycle in detail.

#### Solution

**Thread:** Lightweight subprocess that allows concurrent execution of multiple parts of program.

Table 11: Thread Creation Methods

Method	Description	Example
<b>Extending Thread</b>	Inherit Thread class	class MyThread extends Thread
<b>Implementing Runnable</b>	Implement Runnable interface	class MyTask implements Runnable

#### Diagram: Thread Life Cycle

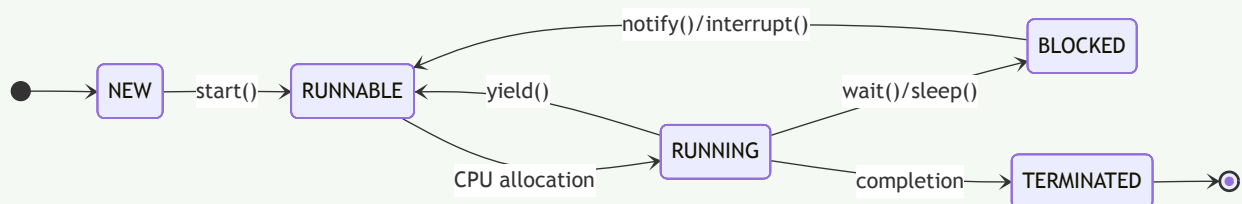


Table 12: Thread States

State	Description
<b>NEW</b>	Thread created but not started
<b>RUNNABLE</b>	Ready to run, waiting for CPU
<b>RUNNING</b>	Currently executing
<b>BLOCKED</b>	Waiting for resource or sleep
<b>TERMINATED</b>	Execution completed

#### Code Block:

```
1 // Method 1: Extending Thread
2 class MyThread extends Thread {
3     public void run() {
4         System.out.println("Thread running");
5     }
6 }
7
8 // Method 2: Implementing Runnable
9 class MyTask implements Runnable {
10    public void run() {
11        System.out.println("Task running");
12    }
13 }
14
15 class Main {
16     public static void main(String[] args) {
17         MyThread t1 = new MyThread();
18         Thread t2 = new Thread(new MyTask());
19         t1.start();
20         t2.start();
21     }
22 }
```

#### Mnemonic

“Thread Concurrent Execution”

### Question 3(a OR) [3 marks]

Explain the purpose of JVM in java.

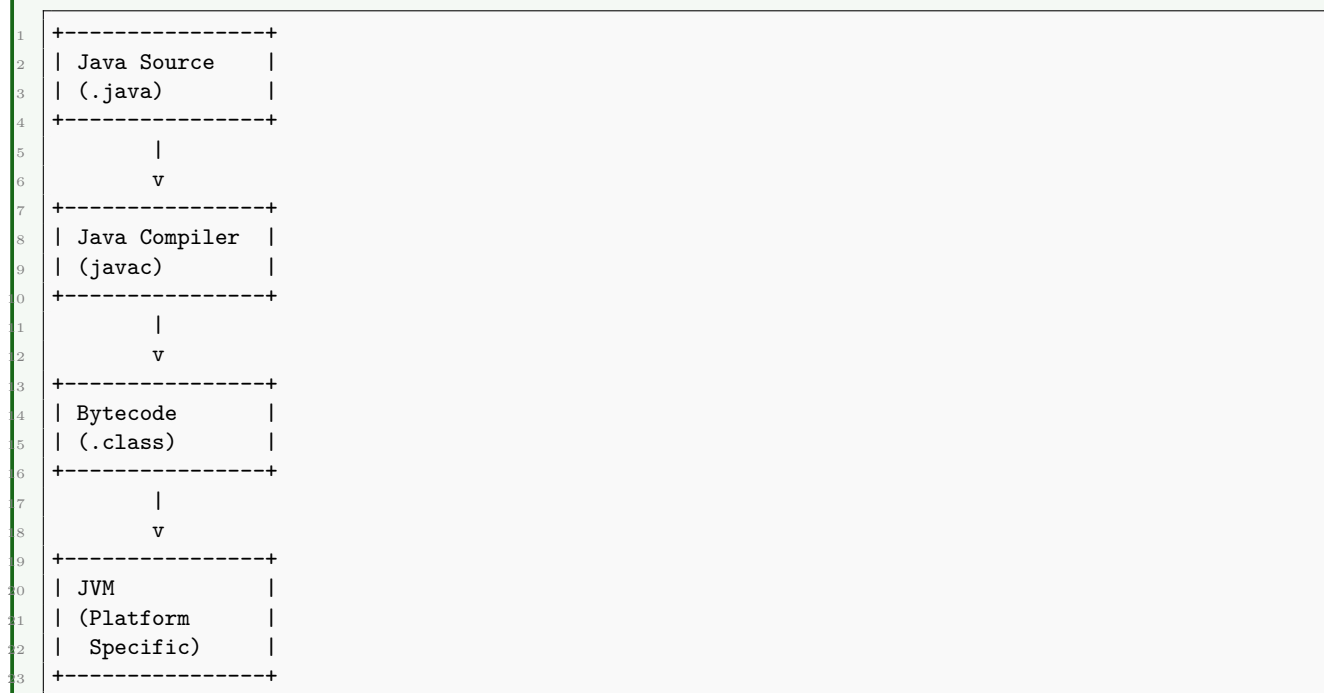
#### Solution

**JVM (Java Virtual Machine):** Runtime environment that executes Java bytecode and provides platform independence.

Table 13: JVM Components

Component	Purpose
<b>Class Loader</b>	Loads .class files into memory
<b>Execution Engine</b>	Executes bytecode
<b>Memory Area</b>	Manages heap and stack memory
<b>Garbage Collector</b>	Automatic memory management

#### Diagram:



- **Platform Independence:** “Write Once, Run Anywhere”
- **Memory Management:** Automatic garbage collection
- **Security:** Bytecode verification

#### Mnemonic

“JVM Java Virtual Machine”

### Question 3(b OR) [4 marks]

Define: Package. Write the steps to create a Package with suitable example.

#### Solution

**Package:** Collection of related classes and interfaces grouped together, providing namespace and access control.

Table 14: Package Benefits

Benefit	Description
<b>Namespace</b>	Avoid name conflicts
<b>Access Control</b>	Better encapsulation
<b>Organization</b>	Logical grouping
<b>Reusability</b>	Easy to maintain

#### Steps to create Package:

1. **Declare package** at top of file
2. **Create directory** structure matching package name
3. **Compile** with package structure
4. **Import** in other classes

#### Code Block:

```

1 // File: com/company/utilities/Calculator.java
2 package com.company.utilities;
3
4 public class Calculator {
5     public int add(int a, int b) {
6         return a + b;
7     }
8 }
9
10 // File: Main.java
11 import com.company.utilities.Calculator;
12
13 class Main {
14     public static void main(String[] args) {
15         Calculator calc = new Calculator();
16         System.out.println(calc.add(5, 3));
17     }
18 }

```

#### Directory Structure:

```

1 com/
2   company/
3     utilities/
4       Calculator.class
5 Main.class

```

#### Mnemonic

“Package Groups Classes”

### Question 3(c OR) [7 marks]

Explain Synchronization in Thread with suitable example.

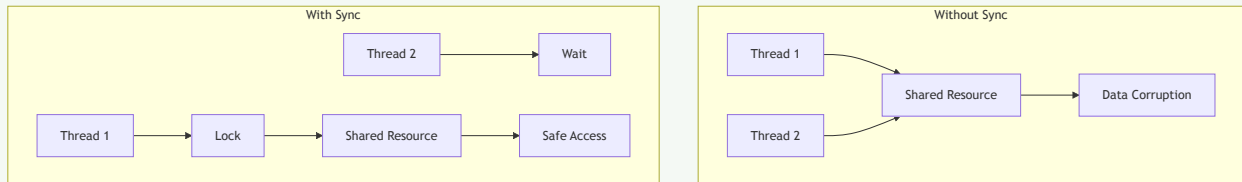
#### Solution

**Synchronization:** Mechanism to control access to shared resources by multiple threads to avoid data inconsistency.

Table 15: Synchronization Types

Type	Description	Usage
<b>Synchronized method</b>	Entire method locked	synchronized void method()
<b>Synchronized block</b>	Specific code block locked	synchronized(object) { }
<b>Static synchronization</b>	Class level locking	synchronized static void method()

## Diagram: Without vs With Synchronization



### Code Block:

```

1 class Counter {
2     private int count = 0;
3
4     // Synchronized method
5     public synchronized void increment() {
6         count++;
7     }
8
9     // Synchronized block
10    public void decrement() {
11        synchronized(this) {
12            count--;
13        }
14    }
15
16    public int getCount() {
17        return count;
18    }
19 }
20
21 class CounterThread extends Thread {
22     Counter counter;
23
24     CounterThread(Counter c) {
25         counter = c;
26     }
27
28     public void run() {
29         for(int i = 0; i < 1000; i++) {
30             counter.increment();
31         }
32     }
33 }
34
35 class Main {
36     public static void main(String[] args) throws InterruptedException {
37         Counter c = new Counter();
38         CounterThread t1 = new CounterThread(c);
39         CounterThread t2 = new CounterThread(c);
40
41         t1.start();
42         t2.start();
43
44         t1.join();
45         t2.join();
46
47         System.out.println("Final count: " + c.getCount());
48     }
49 }
  
```

### Mnemonic

“Synchronization Prevents Race Conditions”

### Question 4(a) [3 marks]

Differentiate between String class and StringBuffer class.

#### Solution

Table 16: String vs StringBuffer

Aspect	String	StringBuffer
<b>Mutability</b>	Immutable (cannot change)	Mutable (can change)
<b>Performance</b>	Slower for concatenation	Faster for concatenation
<b>Memory</b>	Creates new object each time	Modifies existing object
<b>Thread Safety</b>	Thread safe	Thread safe
<b>Methods</b>	concat(), substring()	append(), insert(), delete()

#### Code Block:

```
1 // String - Immutable
2 String s1 = "Hello";
3 s1 = s1 + " World"; // Creates new String object
4
5 // StringBuffer - Mutable
6 StringBuffer sb = new StringBuffer("Hello");
7 sb.append(" World"); // Modifies existing object
```

- **String:** Use when content doesn't change frequently
- **StringBuffer:** Use when frequent modifications needed

#### Mnemonic

"String Immutable, StringBuffer Mutable"

### Question 4(b) [4 marks]

Write a Java Program to find sum and average of 10 numbers of an array.

#### Solution

#### Code Block:

```
1 class ArraySum {
2     public static void main(String[] args) {
3         // Initialize array with 10 numbers
4         int[] numbers = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
5
6         int sum = 0;
7
8         // Calculate sum
9         for(int i = 0; i < numbers.length; i++) {
10             sum += numbers[i];
11         }
12
13         // Calculate average
14         double average = (double) sum / numbers.length;
15
16         // Display results
17         System.out.println("Array elements: ");
18         for(int num : numbers) {
19             System.out.print(num + " ");
20         }
21
22         System.out.println("\nSum: " + sum);
23     }
24 }
```

```

33     System.out.println("Average: " + average);
34 }
35 }

```

#### Output:

```

1 Array elements: 10 20 30 40 50 60 70 80 90 100
2 Sum: 550
3 Average: 55.0

```

#### Logic Steps:

1. **Initialize** array with 10 numbers
2. **Loop** through array to calculate sum
3. **Calculate** average = sum / length
4. **Display** results

#### Mnemonic

“Loop Sum Divide Average”

### Question 4(c) [7 marks]

I) Explain abstract class with suitable example. II) Explain final class with suitable example.

#### Solution

**I) Abstract Class:** Class that cannot be instantiated, contains abstract methods that must be implemented by subclasses.

Table 17: Abstract Class Features

Feature	Description
<b>Cannot instantiate</b>	No object creation
<b>Abstract methods</b>	Methods without implementation
<b>Concrete methods</b>	Methods with implementation
<b>Inheritance</b>	Subclasses must implement abstract methods

### Code Block - Abstract Class:

```
1 abstract class Shape {
2     String color;
3
4     // Abstract method
5     abstract void draw();
6
7     // Concrete method
8     void setColor(String c) {
9         color = c;
10    }
11 }
12
13 class Circle extends Shape {
14     void draw() {
15         System.out.println("Drawing Circle");
16     }
17 }
18
19 class Main {
20     public static void main(String[] args) {
21         // Shape s = new Shape(); // Error: Cannot instantiate
22         Circle c = new Circle();
23         c.draw();
24     }
25 }
```

**II) Final Class:** Class that cannot be extended (no inheritance allowed).

Table 18: Final Class Features

Feature	Description
<b>No inheritance</b>	Cannot be extended
<b>Security</b>	Prevents modification
<b>Performance</b>	Better optimization
<b>Examples</b>	String, Integer, System

### Code Block - Final Class:

```
1 final class FinalClass {
2     void display() {
3         System.out.println("This is final class");
4     }
5 }
6
7 // class SubClass extends FinalClass { } // Error: Cannot extend
8
9 class Main {
10     public static void main(String[] args) {
11         FinalClass obj = new FinalClass();
12         obj.display();
13     }
14 }
```

### Mnemonic

“Abstract Incomplete, Final Complete”

### Question 4(a OR) [3 marks]

Explain Garbage Collection in Java.

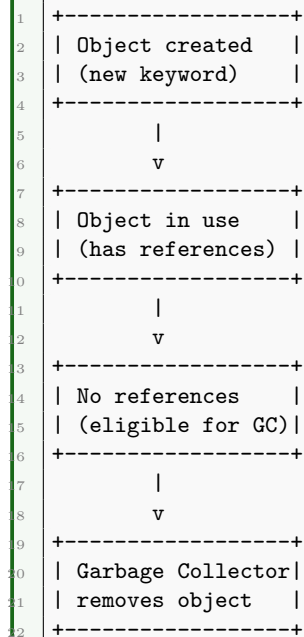
## Solution

**Garbage Collection:** Automatic memory management process that removes unused objects from heap memory.

Table 19: GC Benefits

Benefit	Description
<b>Automatic</b>	No manual memory management
<b>Memory leak prevention</b>	Removes unreferenced objects
<b>Performance</b>	Optimizes memory usage
<b>Safety</b>	Prevents memory errors

### Diagram:



- **When occurs:** When heap memory is low or `System.gc()` called
- **Process:** Mark and Sweep algorithm
- **Cannot guarantee:** Exact timing of garbage collection

## Mnemonic

“GC Automatic Memory Cleanup”

## Question 4(b OR) [4 marks]

Write a Java program to handle user defined exception for ‘Divide by Zero’ error.

## Solution

### Code Block:

```
1 // User defined exception class
2 class DivideByZeroException extends Exception {
3     public DivideByZeroException(String message) {
4         super(message);
5     }
6 }
7
8 class Calculator {
9     public static double divide(int a, int b) throws DivideByZeroException {
10         if(b == 0) {
```

```

1         throw new DivideByZeroException("Cannot divide by zero!");
2     }
3     return (double) a / b;
4 }
5 }
6
7 class Main {
8     public static void main(String[] args) {
9         try {
10             int num1 = 10;
11             int num2 = 0;
12
13             double result = Calculator.divide(num1, num2);
14             System.out.println("Result: " + result);
15
16         } catch (DivideByZeroException e) {
17             System.out.println("Error: " + e.getMessage());
18         }
19     }
20 }

```

#### Output:

```

1 Error: Cannot divide by zero!

```

#### Steps:

1. **Create** custom exception class extending Exception
2. **Throw** exception when condition occurs
3. **Handle** exception with try-catch block

#### Mnemonic

“Custom Exception Handle Error”

### Question 4(c OR) [7 marks]

Write a java program to demonstrate multiple try block and multiple catch block exception.

#### Solution

##### Code Block:

```

1 class MultipleExceptionDemo {
2     public static void main(String[] args) {
3         // First try block
4         try {
5             int[] arr = {1, 2, 3};
6             System.out.println("Array element: " + arr[5]); // ArrayIndexOutOfBoundsException
7         }
8         catch (ArrayIndexOutOfBoundsException e) {
9             System.out.println("Array index error: " + e.getMessage());
10        }
11        catch (Exception e) {
12            System.out.println("General exception: " + e.getMessage());
13        }
14
15        // Second try block
16        try {
17            String str = null;
18            System.out.println("String length: " + str.length()); // NullPointerException
19        }
20        catch (NullPointerException e) {
21            System.out.println("Null pointer error: " + e.getMessage());
22        }
23    }
24 }

```

```

33 // Third try block with multiple catch
34 try {
35     int a = 10;
36     int b = 0;
37     int result = a / b; // ArithmeticException
38
39     String s = "abc";
40     int num = Integer.parseInt(s); // NumberFormatException
41 }
42 catch(ArithmeticException e) {
43     System.out.println("Arithmetic error: " + e.getMessage());
44 }
45 catch(NumberFormatException e) {
46     System.out.println("Number format error: " + e.getMessage());
47 }
48 catch(Exception e) {
49     System.out.println("Other error: " + e.getMessage());
50 }
51 finally {
52     System.out.println("Program completed");
53 }
54 }
55 }
56 }

```

#### Output:

```

1 Array index error: Index 5 out of bounds for length 3
2 Null pointer error: null
3 Arithmetic error: / by zero
4 Program completed

```

#### Features demonstrated:

- **Multiple try blocks:** Each handles different operations
- **Multiple catch blocks:** Each handles specific exception type
- **Exception hierarchy:** General Exception catches all
- **Finally block:** Always executes

#### Mnemonic

“Multiple Try Multiple Catch”

### Question 5(a) [3 marks]

Write a program in Java to create a file and perform write operation on this file.

#### Solution

##### Code Block:

```

1 import java.io.*;
2
3 class FileWriteDemo {
4     public static void main(String[] args) {
5         try {
6             // Create file
7             File file = new File("demo.txt");
8
9             // Create FileWriter object
10            FileWriter writer = new FileWriter(file);
11
12            // Write data to file
13            writer.write("Hello World!\n");
14            writer.write("This is Java file writing demo.\n");

```

```

5      writer.write("File created successfully.");
6
7      // Close the writer
8      writer.close();
9
10     System.out.println("File created and data written successfully!");
11
12     } catch(IOException e) {
13         System.out.println("Error: " + e.getMessage());
14     }
15 }
16 }

```

#### Steps:

1. **Import** java.io package
2. **Create** File object with filename
3. **Create** FileWriter object
4. **Write** data using write() method
5. **Close** writer to save changes

#### Mnemonic

"File Writer Write Close"

### Question 5(b) [4 marks]

Explain throw and finally in Exception Handling with example.

#### Solution

**Throw:** Keyword used to explicitly throw an exception. **Finally:** Block that always executes regardless of exception occurrence.

Table 20: Throw vs Finally

Keyword	Purpose	Usage
<b>throw</b>	Explicitly throw exception	throw new Exception()
<b>finally</b>	Always execute cleanup code	finally { }

### Code Block:

```
1 class ThrowFinallyDemo {
2     public static void checkAge(int age) throws Exception {
3         if(age < 18) {
4             throw new Exception("Age must be 18 or above");
5         }
6         System.out.println("Valid age: " + age);
7     }
8
9     public static void main(String[] args) {
10        try {
11            checkAge(15); // Will throw exception
12        }
13        catch(Exception e) {
14            System.out.println("Error: " + e.getMessage());
15        }
16        finally {
17            System.out.println("Finally block always executes");
18        }
19    }
20 }
```

### Output:

```
1 Error: Age must be 18 or above
2 Finally block always executes
```

- **Throw:** Forces exception to occur
- **Finally:** Cleanup code, closes resources

### Mnemonic

“Throw Exception, Finally Always”

## Question 5(c) [7 marks]

Describe: Polymorphism. Explain run time polymorphism with suitable example in java.

### Solution

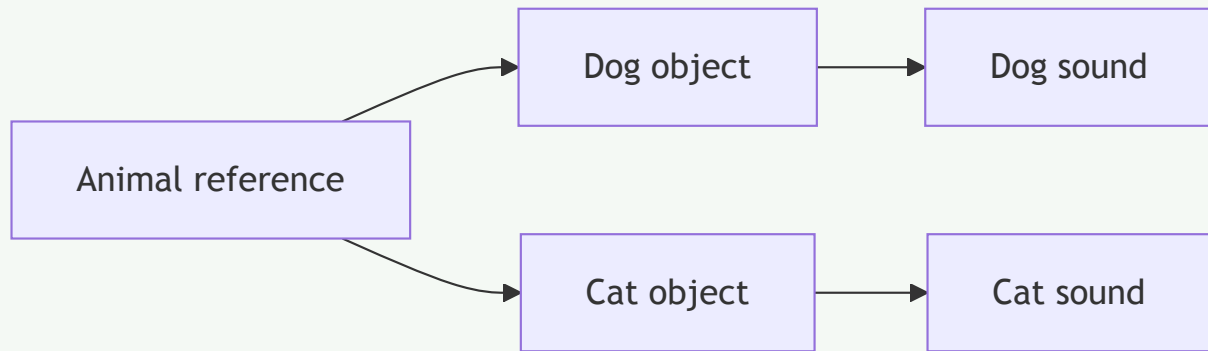
**Polymorphism:** One interface, multiple implementations. Object behaves differently based on its actual type.

Table 21: Polymorphism Types

Type	Description	When Decided
<b>Compile-time</b>	Method overloading	At compilation
<b>Run-time</b>	Method overriding	At execution

**Run-time Polymorphism:** Method call resolved at runtime based on actual object type.

**Diagram:**



**Code Block:**

```
1 class Animal {
2     void makeSound() {
3         System.out.println("Animal makes sound");
4     }
5 }
6
7 class Dog extends Animal {
8     @Override
9     void makeSound() {
10        System.out.println("Dog barks");
11    }
12 }
13
14 class Cat extends Animal {
15     @Override
16     void makeSound() {
17        System.out.println("Cat meows");
18    }
19 }
20
21 class Main {
22     public static void main(String[] args) {
23         Animal animal1 = new Dog(); // Upcasting
24         Animal animal2 = new Cat(); // Upcasting
25
26         animal1.makeSound(); // Output: Dog barks
27         animal2.makeSound(); // Output: Cat meows
28
29         // Array of animals
30         Animal[] animals = {new Dog(), new Cat(), new Dog()};
31         for(Animal a : animals) {
32             a.makeSound(); // Dynamic method dispatch
33         }
34     }
35 }
```

**Output:**

```
1 Dog barks
2 Cat meows
3 Dog barks
4 Cat meows
5 Dog barks
```

**Features:**

- **Dynamic Method Dispatch:** JVM decides which method to call at runtime
- **Upcasting:** Child object referenced by parent reference
- **Flexibility:** Same code works with different object types

### Mnemonic

“Polymorphism Many Forms Runtime”

### Question 5(a OR) [3 marks]

Write a program in Java that read the content of a file byte by byte and copy it into another file.

#### Solution

##### Code Block:

```
1 import java.io.*;
2
3 class FileCopyDemo {
4     public static void main(String[] args) {
5         try {
6             // Create input stream to read from source file
7             FileInputStream input = new FileInputStream("source.txt");
8
9             // Create output stream to write to destination file
10            FileOutputStream output = new FileOutputStream("destination.txt");
11
12            int byteData;
13
14            // Read byte by byte and copy
15            while((byteData = input.read()) != -1) {
16                output.write(byteData);
17            }
18
19            // Close streams
20            input.close();
21            output.close();
22
23            System.out.println("File copied successfully!");
24
25        } catch(IOException e) {
26            System.out.println("Error: " + e.getMessage());
27        }
28    }
29 }
```

##### Steps:

1. Create FileInputStream for reading
2. Create FileOutputStream for writing
3. Read byte by byte using read()
4. Write each byte using write()
5. Close both streams

### Mnemonic

“Read Byte Write Byte”

### Question 5(b OR) [4 marks]

Explain the different I/O Classes available with Java.

## Solution

Table 22: Java I/O Classes

Class Type	Class Name	Purpose
Byte Stream	FileInputStream	Read bytes from file
Byte Stream	FileOutputStream	Write bytes to file
Character Stream	FileReader	Read characters from file
Character Stream	FileWriter	Write characters to file
Buffered	BufferedReader	Efficient character reading
Buffered	BufferedWriter	Efficient character writing

### Diagram: I/O Class Hierarchy

```

1  +-----+
2  |   InputStream   |
3  +-----+
4      |
5      +--- FileInputStream
6      +--- BufferedInputStream
7
8  +-----+
9  |   OutputStream   |
10 +-----+
11      |
12      +--- FileOutputStream
13      +--- BufferedOutputStream
14
15 +-----+
16 |    Reader    |
17 +-----+
18      |
19      +--- FileReader
20      +--- BufferedReader
21
22 +-----+
23 |    Writer    |
24 +-----+
25      |
26      +--- FileWriter
27      +--- BufferedWriter

```

### Code Example:

```

1  // Byte streams
2  FileInputStream fis = new FileInputStream("file.txt");
3  FileOutputStream fos = new FileOutputStream("output.txt");
4
5  // Character streams
6  FileReader fr = new FileReader("file.txt");
7  FileWriter fw = new FileWriter("output.txt");
8
9  // Buffered streams
10 BufferedReader br = new BufferedReader(new FileReader("file.txt"));
11 BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"));

```

## Mnemonic

“Byte Character Buffered Streams”

### Question 5(c OR) [7 marks]

Write a java program that executes two threads. One thread displays “Java Programming” every 3 seconds, and the other displays “Semester - 4th IT” every 6 seconds.(Create the threads by extending the Thread class)

#### Solution

##### Code Block:

```
1 class JavaThread extends Thread {
2     public void run() {
3         try {
4             while(true) {
5                 System.out.println("Java Programming");
6                 Thread.sleep(3000); // Sleep for 3 seconds
7             }
8         } catch (InterruptedException e) {
9             System.out.println("JavaThread interrupted");
10        }
11    }
12 }
13
14 class SemesterThread extends Thread {
15     public void run() {
16         try {
17             while(true) {
18                 System.out.println("Semester - 4th IT");
19                 Thread.sleep(6000); // Sleep for 6 seconds
20             }
21         } catch (InterruptedException e) {
22             System.out.println("SemesterThread interrupted");
23        }
24    }
25 }
26
27 class Main {
28     public static void main(String[] args) {
29         // Create thread objects
30         JavaThread javaThread = new JavaThread();
31         SemesterThread semesterThread = new SemesterThread();
32
33         // Start both threads
34         javaThread.start();
35         semesterThread.start();
36
37         // Let threads run for 20 seconds then stop
38         try {
39             Thread.sleep(20000);
40             javaThread.interrupt();
41             semesterThread.interrupt();
42         } catch (InterruptedException e) {
43             System.out.println("Main thread interrupted");
44         }
45     }
46 }
```

##### Sample Output:

```
1 Java Programming
2 Semester - 4th IT
3 Java Programming
4 Java Programming
5 Semester - 4th IT
6 Java Programming
7 Java Programming
8 Semester - 4th IT
9 ...
```

**Features:**

- **Two separate threads:** Each with different timing
- **Thread.sleep():** Pauses execution for specified milliseconds
- **Concurrent execution:** Both threads run simultaneously
- **Extending Thread class:** Override run() method

**Execution Pattern:**

- **JavaThread:** Displays every 3 seconds
- **SemesterThread:** Displays every 6 seconds
- Both run concurrently showing different timing

**Mnemonic**

“Two Threads Different Timing”