Object Oriented Programming With Java (4341602) - Summer 2025 Solution

Milav Dabgar

May 15, 2025

Question 1(a) [3 marks]

Differentiate between Procedure Oriented Programming (POP) and object-oriented programming (OOP).

Solution

Difference:

Table 1. POP vs OOP

Aspect	**POP**	**OOP**
Approach	Top-down approach	Bottom-up approach
Focus	Functions and procedures	Objects and classes
Data Security	Less secure, global data	More secure, data encapsulation
Problem Solving	Divides into functions	Divides into objects

Key Points:
- **POP**: Functions are primary building blocks.
- **OOP**: Objects contain both data and methods.
- **Reusability**: OOP provides better code reusability.

Mnemonic

"POP Functions, OOP Objects"

Question 1(b) [4 marks]

Enlist and explain the basic concepts of OOP.

Solution

Basic OOP Concepts:
- **Encapsulation**: Binding data and methods together in a class. It hides data from outside interference.
- **Inheritance**: Creating new classes from existing classes. It promotes code reusability.
- **Polymorphism**: Same method name with different implementations (e.g., Overloading, Overriding).
- **Abstraction**: Hiding implementation details from user and showing only functionality.

Benefits:
- **Code Reusability**: Through inheritance and polymorphism.
- **Data Security**: Through encapsulation.
- **Easy Maintenance**: Modular approach makes updates easier.

Mnemonic

"Every Intelligent Person Abstracts"

Question 1(c) [7 marks]

Define Constructor. Enlist different types of Constructors and explain any 2 of them with a proper example.

Solution

Constructor Definition: A constructor is a special method that initializes objects when they are created. It has the same name as the class and no return type.

Types of Constructors:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Private Constructor

Listing 1. Types of Constructors

```java
class Student {
    String name;
    int age;

    // 1. Default Constructor
    public Student() {
        name = "Unknown";
        age = 0;
    }

    // 2. Parameterized Constructor
    public Student(String n, int a) {
        name = n;
        age = a;
    }
}

class Main {
    public static void main(String[] args) {
        Student s1 = new Student();        // Calls Default
        Student s2 = new Student("John", 20); // Calls Parameterized
    }
}
```

Key Features:

- **Automatic Invocation**: Called automatically during object creation.
- **No Return Type**: Constructors don't have a return type (not even void).

Mnemonic

"Constructors Create Objects"

Question 1(c OR) [7 marks]

Explain String class. Enlist different methods of String class and explain any 3 of them with a proper example.

Solution

String Class: String class in Java represents immutable character sequences. Once created, String objects cannot be modified.
String Methods:

Table 2. Common String Methods

Method	Purpose
`length()`	Returns string length
`charAt(index)`	Returns character at specified index
`substring(start, end)`	Extracts substring from start to end-1
`indexOf(char)`	Finds first position of character
`toUpperCase()`	Converts all characters to uppercase

Listing 2. String Methods Demo

```java
public class StringDemo {
    public static void main(String[] args) {
        String str = "Hello World";

        // 1. length() method
        System.out.println("Length: " + str.length()); // 11

        // 2. charAt() method
        System.out.println("Char at 0: " + str.charAt(0)); // H

        // 3. substring() method
        System.out.println("Substring: " + str.substring(0, 5)); // Hello
    }
}
```

Key Points:
- **Immutable**: String objects cannot be changed after creation.
- **Memory Efficient**: Uses String pool for storage optimization.

Mnemonic

"Strings Store Text"

Question 2(a) [3 marks]

Define Garbage collection. Describe the importance of Garbage collection in JAVA Programming.

Solution

Garbage Collection Definition: Automatic memory management process that reclaims memory occupied by objects that are no longer referenced (reachable) in the program.
Importance:
- **Automatic Memory Management**: Developers don't need to manually allocate and deallocate memory.
- **Prevents Memory Leaks**: Automatically frees unused memory, reducing risk of leaks.
- **Application Performance**: Optimizes memory usage by cleaning up heap space.

Benefits:
- **Productivity**: Programmer focuses on business logic, not memory.
- **Reliability**: Reduces system crashes due to memory errors.

Mnemonic

"Garbage Collector Cleans Memory"

Question 2(b) [4 marks]

List down the four ways to make an object eligible for garbage collection.

Solution

Ways for GC Eligibility: An object becomes eligible for Garbage Collection when there are no more references to it.

Table 3. GC Eligibility Methods

Method	Description
Nullifying Reference	Explicitly setting an object reference to `null`.
Reassigning Reference	Pointing a reference variable to another object.
Anonymous Objects	Creating objects without assigning them to a reference variable.
Island of Isolation	Two or more objects referencing only each other with no external access.

Examples:
1. `Student s = new Student(); s = null;`
2. `Student s1 = new Student(); Student s2 = new Student(); s1 = s2;`
3. `new Student();`

Mnemonic

"Null References Attract Islands"

Question 2(c) [7 marks]

Write a Java Program to demonstrate a static block that gets executed before main. Explain its significance.

Solution

Listing 3. Static Block Demo

```java
public class StaticBlockDemo {
    static int count;

    // Static block - Executes BEFORE main
    static {
        System.out.println("Static block executed first");
        count = 10;
        System.out.println("Count initialized to: " + count);
    }

    public static void main(String[] args) {
        System.out.println("Main method started");
        System.out.println("Count value: " + count);
    }
}
```

Output:

```
Static block executed first
Count initialized to: 10
Main method started
Count value: 10
```

Significance:

- **Early Initialization**: Executes automatically when the class is loaded into memory, before invoking the `main` method.
- **Class Loading**: It runs only once per class loading.
- **Uses**: Ideal for initializing static variables or loading native libraries/configurations.

Mnemonic

"Static Blocks Start Before Main"

Question 2(a OR) [3 marks]

Describe Minor/Incremental and Major/Full Garbage collection in JAVA.

Solution

Types of Garbage Collection:

Table 4. Minor vs Major GC

Type	Description	Frequency
Minor GC	Cleans the **Young Generation** (Eden space). Removes short-lived objects.	Frequent, Fast
Major GC	Cleans the **Old Generation** (Tenured space). Involves whole heap (Full GC).	Less Frequent, Slow

Details:

- **Minor GC**: Triggered when Eden space is full. Fast and causes little pause.
- **Major GC**: Triggered when Old Generation is full. Causes "Stop the World" pauses which can affect performance.

Mnemonic

"Minor Frequent, Major Slow"

Question 2(b OR) [4 marks]

Explicate the finalize() method in java with its advantages.

Solution

finalize() Method: It is a method defined in the `Object` class. The Garbage Collector calls this method on an object just before it destroys the object and reclaims its memory. It is used for cleanup processing.

Syntax:

```java
protected void finalize() throws Throwable {
    // Cleanup code
}
```

Advantages:

- **Resource Cleanup**: Useful for closing non-Java resources like file streams, database connections, or sockets if the programmer forgot to close them.
- **Safety Net**: Acts as a final safety check to ensure critical resources are released.

Note: It is deprecated in newer Java versions (since Java 9) in favor of `try-with-resources` and `Cleaners`.

Mnemonic

"Finalize Frees Resources"

Question 2(c OR) [7 marks]

Explain the syntax of public static void main (String[] args). Write a Java Program to print input taken as command line argument.

Solution

Main Method Syntax Explanation: `public static void main(String[] args)`

- **public**: Access modifier making it accessible from anywhere (JVM needs to call it).
- **static**: Allows JVM to call the method without creating an instance of the class.
- **void**: The method does not return any value.
- **main**: The specific identifier name JVM looks for as the starting point.
- **String[] args**: Array of String objects storing command-line arguments.

Listing 4. Command Line Arguments Demo

```java
public class CommandLineDemo {
    public static void main(String[] args) {
        System.out.println("Number of arguments: " + args.length);

        if(args.length > 0) {
            System.out.println("Command line arguments:");
            for(int i = 0; i < args.length; i++) {
                System.out.println("Arg " + i + ": " + args[i]);
            }
        } else {
            System.out.println("No arguments provided");
        }
    }
}
```

Execution: `java CommandLineDemo Hello World 123`
Output:

```
Number of arguments: 3
Command line arguments:
Arg 0: Hello
Arg 1: World
Arg 2: 123
```

Question 3(a) [3 marks]

Enlist and Explain various Java access modifier(s).

Solution

Java Access Modifiers: Access modifiers determine the scope (visibility) of a class, constructor, variable, method, or data member.

Table 5. Access Modifiers Scope

Modifier	Class	Package	Subclass	World
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
default	Yes	Yes	No	No
private	Yes	No	No	No

Usage:

- **public**: Universal access.
- **protected**: Used for inheritance; accessible within package and outside by subclasses.
- **default**: (No keyword) Package-private.
- **private**: Restricted to the defining class only.

Mnemonic

"Public Protected Default Private"

Question 3(b) [4 marks]

Describe interface in JAVA. Demonstrate inheritance of an interface with an executable example.

Solution

Interface in Java: An interface is a reference type in Java. It is similar to a class but represents a contract. It can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot be instantiated—they can only be implemented by classes or extended by other interfaces.

Listing 5. Interface Inheritance

```java
// Parent interface
interface Animal {
    void sound();
}

// Child interface inheriting from Animal
interface Mammal extends Animal {
    void walk();
}

// Class implementing the child interface
class Dog implements Mammal {
    public void sound() {
        System.out.println("Dog barks");
    }

    public void walk() {
        System.out.println("Dog walks on four legs");
    }
}

class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();
        d.walk();
    }
}
```

Mnemonic

"Interfaces Inherit Contracts"

Question 3(c) [7 marks]

Define super keyword and demonstrate the use of super keyword with an executable Java Program

Solution

super Keyword: The super keyword in Java is a reference variable used to refer to the immediate parent class object. It is used to bypass the current class's scope to access members of the parent class.
Uses:

1. **Variable Access**: To refer to immediate parent class instance variable ('super.variable').
2. **Method Call**: To invoke immediate parent class method ('super.method()').
3. **Constructor Call**: To invoke immediate parent class constructor ('super()').

Listing 6. Using super Keyword

```java
class Animal {
    String name = "Animal";

    Animal(String type) {
        System.out.println("Animal constructor: " + type);
    }

    void sound() {
        System.out.println("Animal makes sound");
    }
}

class Dog extends Animal {
    String name = "Dog";

    Dog() {
        // 1. Calling parent constructor
        super("Mammal");
        System.out.println("Dog constructor");
    }

    void sound() {
        // 2. Calling parent method
        super.sound();
        System.out.println("Dog barks");
    }

    void display() {
        // 3. Accessing parent variable
        System.out.println("Parent name: " + super.name);
        System.out.println("Child name: " + this.name);
    }
}

class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();
        d.display();
    }
}
```

Mnemonic

"Super Calls Parent"

Question 3(a OR) [3 marks]

Explain package in JAVA with workable illustration.

Solution

Package: A package in Java is a mechanism to encapsulate a group of classes, sub-packages, and interfaces.
Benefits:

- **Organization**: Categorizes classes (e.g., 'model', 'view', 'controller') for better code management.
- **Access Control**: Provides access protection (default/protected access).
- **Namespace Management**: Prevents naming conflicts.

Listing 7. Creating and Using Package

```java
// File: com/company/model/Student.java
package com.company.model;

public class Student {
    private String name;
    public void setName(String name) { this.name = name; }
}

// File: Main.java
import com.company.model.Student;

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("John");
    }
}
```

Mnemonic

"Packages Organize Classes"

Question 3(b OR) [4 marks]

Explain abstract and final keywords with a viable illustration.

Solution

Table 6. Abstract vs Final

Keyword	Purpose	Usage Level
abstract	Defines incomplete implementation (template). Cannot be instantiated.	Class, Method
final	Defines constant or unchangeable entity. Prevents modification/extension.	Class, Method, Variable

Listing 8. Abstract and Final Demo

```java
// Abstract class
abstract class Shape {
    final double PI = 3.14;  // final variable (Constant)

    abstract void draw();    // abstract method (No body)

    final void display() {   // final method (Cannot override)
        System.out.println("Displaying shape");
    }
}

// Final class (Cannot extend)
final class Circle extends Shape {
    void draw() {
        System.out.println("Drawing circle");
    }
}
```

Mnemonic

"Abstract Allows, Final Forbids"

Question 3(c OR) [7 marks]

State Dynamic Method Dispatch in Java Programming language context. Construct an executable program demonstrating Dynamic Method Dispatch.

Solution

Dynamic Method Dispatch (Runtime Polymorphism): It is a mechanism by which a call to an overridden method is resolved at runtime rather than compile-time. This is implemented using parent class reference variable referring to a child class object.

How it works:

1. Retrieve the actual object type referenced by the variable.
2. Look up the method in the Virtual Method Table (vtable).
3. Invoke the method implementation corresponding to the actual object.

Listing 9. Dynamic Dispatch Example

```java
// Base class
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}
```

```java

// Derived classes
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    void sound() {
        System.out.println("Cat meows");
    }
}

class DynamicDispatchDemo {
    public static void main(String[] args) {
        Animal ref;  // Reference variable

        // Runtime method resolution

        ref = new Dog(); // Referencing Dog object
        ref.sound();     // Calls Dog's sound()

        ref = new Cat(); // Referencing Cat object
        ref.sound();     // Calls Cat's sound()

        ref = new Animal();
        ref.sound();     // Calls Animal's sound()
    }
}
```

Output:

```
Dog barks
Cat meows
Animal makes sound
```

Mnemonic

"Dynamic Dispatch Decides Runtime"

Question 4(a) [3 marks]

Explain throw and finally keywords in Exception Handling.

Solution

throw:

- Used to explicitly throw an exception from a method or block of code.
- Mainly used to throw custom exceptions.
- Syntax: `throw new Exception("Message");`

finally:

- A block that follows `try` or `catch`.
- It **always executes** regardless of whether an exception occurred or not.
- Generally used for cleanup code (closing files, releasing resources).

Listing 10. Throw and Finally

```java
try {
    if(age < 0)
        throw new IllegalArgumentException("Invalid age");
} catch(Exception e) {
    System.out.println(e);
} finally {
    System.out.println("This always prints.");
}
```

Mnemonic

"Throw Creates, Finally Cleans"

Question 4(b) [4 marks]

Write a program demonstrating try...catch block in JAVA

Solution

Listing 11. Try-Catch Demo

```java
public class TryCatchDemo {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};
            // This line causes ArrayIndexOutOfBoundsException
            System.out.println("Array element: " + arr[5]);

            // This line will not execute
            int result = 10 / 0;

        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index error: " + e.getMessage());

        } catch(ArithmeticException e) {
            System.out.println("Math error: " + e.getMessage());

        } catch(Exception e) {
            System.out.println("General error: " + e.getMessage());
        }

        System.out.println("Program continues...");
    }
}
```

Output:

```
Array index error: Index 5 out of bounds for length 3
Program continues...
```

Mnemonic

"Try Code, Catch Errors"

Question 4(c) [7 marks]

Define ArrayIndexOutOfBoundsException Exception. Write a workable JAVA program

exhibiting it. Also mention input(s) which will raise this Exception.

Solution

Definition: `ArrayIndexOutOfBoundsException` is a runtime exception thrown when code attempts to access an array element with an illegal index. The index is either negative or greater than or equal to the size of the array.

Listing 12. ArrayIndexOutOfBounds Demo

```java
public class ArrayExceptionDemo {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50}; // Array size: 5

        try {
            // Valid access
            System.out.println("Element at 2: " + numbers[2]);

            // INVALID access - raises exception
            // Index 10 is >= length 5
            System.out.println("Element at 10: " + numbers[10]);

        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught: " + e.getMessage());
            System.out.println("Invalid index accessed!");
        }
    }
}
```

Inputs raising Exception:
- **Negative Index**: `numbers[-1]`
- **Index >= Length**: `numbers[5]` (since valid indices are 0-4)
- **Empty Array**: `numbers[0]` on an empty array

Mnemonic

"Array Bounds Break Programs"

Question 4(a OR) [3 marks]

Draw and explain the life cycle of Thread in JAVA with example.

Solution

Thread Life Cycle: A thread goes through various states during its life cycle.

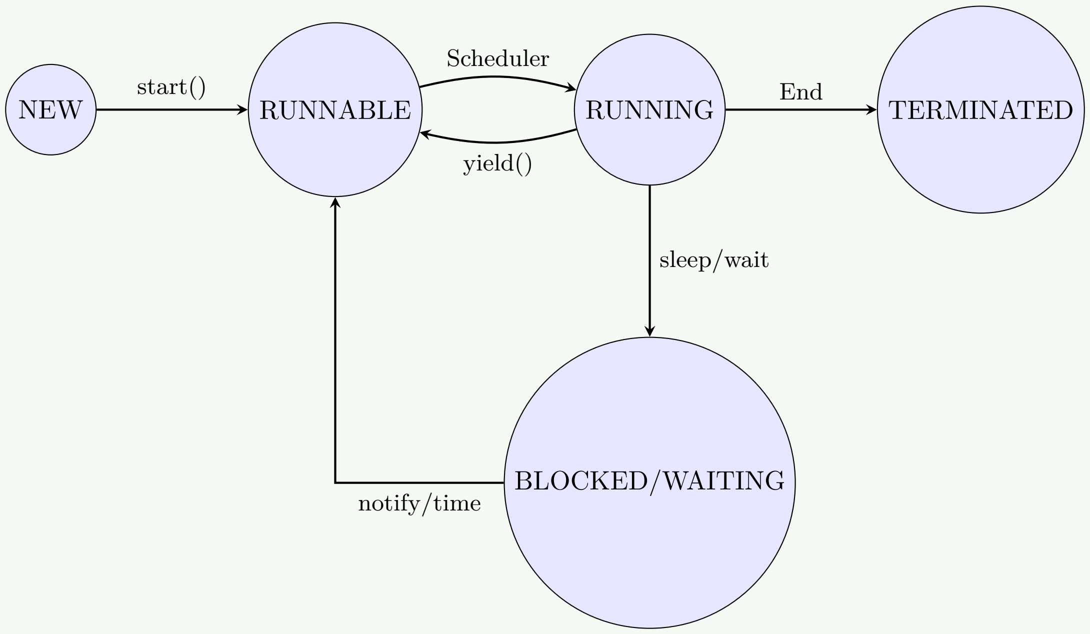

Figure 1. Thread Life Cycle

States:

- **NEW**: Thread creates but not started.
- **RUNNABLE**: Ready to run, waiting for CPU.
- **RUNNING**: Currently executing.
- **BLOCKED/WAITING**: Waiting for resource or another thread.
- **TERMINATED**: Execution completed.

Mnemonic

"New Runs, Blocks Wait, Terminates"

Question 4(b OR) [4 marks]

Explain JAVA Optional class. Describe the OfNullable() method of Optional class.

Solution

Optional Class: Introduced in Java 8, `java.util.Optional` is a container object that may or may not contain a non-null value. It is used to represent the absence of a value without using `null`, helping to avoid `NullPointerException`.
ofNullable() Method:

- **Description**: Returns an `Optional` describing the specified value, if non-null, otherwise returns an empty `Optional`.
- **Syntax**: `static <T> Optional<T> ofNullable(T value)`

Listing 13. Optional.ofNullable Demo

```java
import java.util.Optional;

public class OptionalDemo {
    public static void main(String[] args) {
        String name1 = "John";
        String name2 = null;

        // Returns Optional containing "John"
        Optional<String> opt1 = Optional.ofNullable(name1);

        // Returns empty Optional (no Exception thrown)
        Optional<String> opt2 = Optional.ofNullable(name2);

        System.out.println("opt1 present: " + opt1.isPresent()); // true
        System.out.println("opt2 present: " + opt2.isPresent()); // false

        // usage with orElse
        System.out.println(opt2.orElse("Default Name")); // prints Default Name
    }
}
```

Mnemonic

"Optional Offers Null Safety"

Question 4(c OR) [7 marks]

Write a workable JAVA program showcasing nested try...catch block.

Solution

Listing 14. Nested Try-Catch

```java
public class NestedTryCatchDemo {
    public static void main(String[] args) {
        try {
            System.out.println("Outer try block started");

            try {
                System.out.println("Inner try block started");
                int[] numbers = {10, 20};

                // Causes ArrayIndexOutOfBoundsException
                System.out.println(numbers[5]);

            } catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Inner catch: Array index error");
                // Modifying exception flow - throwing runtime exception
                throw new RuntimeException("Error from inner block");
            }

        } catch(RuntimeException e) {
            System.out.println("Outer catch: Runtime error - " + e.getMessage());

        } catch(Exception e) {
            System.out.println("Outer catch: General error");

        } finally {
            System.out.println("Outer finally: Cleanup");
        }
    }
}
```

Key Features:

- **Layered Handling**: Specific errors handled in inner block, general errors in outer.
- **Flow**: If inner catch handles it, outer catch might not be reached unless re-thrown.

Question 5(a) [3 marks]

Explain thread synchronization with an executable code in JAVA.

Solution

Thread Synchronization: Synchronization is the capability to control the access of multiple threads to any shared resource. It prevents thread interference and memory consistency errors.

Listing 15. Thread Synchronization

```java
class Counter {
    private int count = 0;

    // Synchronized method
    public synchronized void increment() {
        count++;
    }

    public int getCount() { return count; }
}

class SyncDemo extends Thread {
    Counter counter;

    SyncDemo(Counter c) { counter = c; }

    public void run() {
        for(int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}
```

Benefits:
- **Data Consistency**: Essential when multiple threads modify the same data.
- **Thread Safety**: Ensures only one thread accesses the resource at a time.

Mnemonic

"Synchronize Secures Shared Data"

Question 5(b) [4 marks]

Enlist various stream classes in JAVA. Explain anyone with an executable example.

Solution

Stream Classes:

Table 7. Common Stream Classes

Class	Purpose	Type
FileInputStream	Read bytes from file	Byte Stream
FileOutputStream	Write bytes to file	Byte Stream
BufferedReader	Buffered character reading	Character Stream
PrintWriter	Formatted text output	Character Stream

Listing 16. FileInputStream Example

```java
import java.io.*;

public class StreamDemo {
    public static void main(String[] args) {
        try {
            // Write data
            FileOutputStream fos = new FileOutputStream("test.txt");
            String data = "Hello World";
            fos.write(data.getBytes());
            fos.close();

            // Read data
            FileInputStream fis = new FileInputStream("test.txt");
            int ch;
            while((ch = fis.read()) != -1) {
                System.out.print((char)ch);
            }
            fis.close();

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

Mnemonic

"Streams Send Data"

Question 5(c) [7 marks]

Write a JAVA program extending Thread class to display odd numbers between given two integer numbers using thread.

Solution

Listing 17. Odd Number Thread

```java
class OddNumberThread extends Thread {
    private int start;
    private int end;

    public OddNumberThread(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    public void run() {
        System.out.println("Thread started: " + getName());

        for(int i = start; i <= end; i++) {
            if(i % 2 != 0) {  // Check if odd
                System.out.println("Odd number: " + i);
                try {
                    Thread.sleep(500);  // Pause
                } catch(InterruptedException e) {
                    System.out.println("Thread interrupted");
                }
            }
        }
        System.out.println("Thread completed: " + getName());
    }
}

public class OddNumberDemo {
    public static void main(String[] args) {
        // Create threads
        OddNumberThread t1 = new OddNumberThread(1, 10);

        t1.setName("OddThread-1");
        t1.start();

        try {
            t1.join(); // Wait for completion
        } catch(InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Main thread completed");
    }
}
```

Output:

```
Thread started: OddThread-1
Odd number: 1
Odd number: 3
...
Odd number: 9
Thread completed: OddThread-1
Main thread completed
```

Mnemonic

"Threads Take Turns"

Question 5(a OR) [3 marks]

Explain join() and alive() methods of Thread class in JAVA.

Solution

Thread Methods:

- **join()**: This method allows one thread to wait for the completion of another. If `t.join()` is called, the current thread pauses execution until thread `t` terminates.
- **isAlive()**: This method checks if a thread is still running. It returns `true` if the thread has started and not yet died, otherwise `false`.

Listing 18. Join and IsAlive

```java
class TestThread extends Thread {
    public void run() {
        try { sleep(500); } catch(InterruptedException e) {}
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        TestThread t = new TestThread();
        System.out.println("Before start: " + t.isAlive()); // false

        t.start();
        System.out.println("After start: " + t.isAlive()); // true

        t.join(); // Wait for completion
        System.out.println("After join: " + t.isAlive()); // false
    }
}
```

Mnemonic

"Join Waits, Alive Checks"

Question 5(b OR) [4 marks]

Define user-defined exceptions in JAVA. Write a program to show user defined exception.

Solution

User-defined Exceptions: Java allows users to define their own exception classes by extending the `Exception` class (checked) or `RuntimeException` class (unchecked). These are used to handle application-specific logical errors.

Listing 19. Custom Exception

```java
// 1. Create custom exception class
class AgeValidationException extends Exception {
    public AgeValidationException(String message) {
        super(message);
    }
}

class Person {
    public void setAge(int age) throws AgeValidationException {
        if(age < 0) {
            throw new AgeValidationException("Age cannot be negative: " + age);
        }
        System.out.println("Valid age: " + age);
    }
}

public class UserDefinedExceptionDemo {
    public static void main(String[] args) {
        Person p = new Person();
        try {
            p.setAge(-5); // Invalid
        } catch(AgeValidationException e) {
            System.out.println("Caught: " + e.getMessage());
        }
    }
}
```

Mnemonic

"Custom Exceptions Catch Specific Errors"

Question 5(c OR) [7 marks]

Write a JAVA program to copy content of file a.txt to b.txt.

Solution

Listing 20. File Copy Program

```java
import java.io.*;

public class FileCopyDemo {
    public static void main(String[] args) {
        String source = "a.txt";
        String target = "b.txt";

        // Method: Using FileInputStream and FileOutputStream
        copyFile(source, target);
    }

    public static void copyFile(String src, String dest) {
        FileInputStream fis = null;
        FileOutputStream fos = null;

        try {
            // Initialize streams
            fis = new FileInputStream(src);
            fos = new FileOutputStream(dest);

            // Read and write byte by byte
            int ch;
            while((ch = fis.read()) != -1) {
                fos.write(ch);
            }

            System.out.println("File copied successfully!");

        } catch(FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        } catch(IOException e) {
            System.out.println("IO Error: " + e.getMessage());
        } finally {
            // Close streams in finally block
            try {
                if(fis != null) fis.close();
                if(fos != null) fos.close();
            } catch(IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Logic:

1. Open source file in read mode using `FileInputStream`.
2. Open destination file in write mode using `FileOutputStream`.
3. Read byte from source and write to destination until end of file (-1).

4. Close both streams to release system resources.

Mnemonic

"Files Flow From Source To Target"