Java Programming (4343203) - Summer 2025 Solution

Milav Dabgar

May 13, 2025

Question 1(a) [3 marks]

List out the rules to name an identifier in Java with valid and invalid examples.

Solution

Rules for Java Identifiers:

Table 1. Identifier Rules

Rule	Description	Valid Example	Invalid Example
Start Character	Must begin with letter, underscore, or dollar sign	`name`, `_value`, `$cost`	`2name`, `#id`
Following Characters	Can contain letters, digits, underscore, dollar	`student123`, `user_name`	`my-var`, `class@`
Keywords Restriction	Cannot use Java reserved words	`myClass`, `userName`	`class`, `int`
Case Sensitivity	Identifiers are case-sensitive	`Name` $\neq$ `name`	-
Length	No length limit (practically reasonable)	`verylongvariablename`	-

Mnemonic

"LKC: Letters First, Keywords Never, Case Counts"

Question 1(b) [4 marks]

List out different types of operators in Java. Explain Arithmetic and Logical Operators in detail.

Solution

Java Operator Types:

Table 2. Operator Types

Operator Type	Examples
Arithmetic	+, -, *, /, %
Relational	==, !=, <, >, <=, >=
Logical	&&, ||, !
Assignment	=, +=, -=, *=, /=
Unary	++, --, +, -, !
Bitwise	&, |, ^, ~, <<, >>
Ternary	condition ? value1 : value2

Arithmetic Operators:
- **Addition (+)**: Adds two operands
- **Subtraction (-)**: Subtracts second from first
- **Multiplication (*)**: Multiplies two operands
- **Division (/)**: Divides first by second
- **Modulus (%)**: Returns remainder of division

Logical Operators:
- **AND (&&)**: Returns true if both conditions are true
- **OR (——)**: Returns true if at least one condition is true
- **NOT (!)**: Reverses the logical state

Mnemonic

"AMDR-AON: Add Subtract Multiply Divide Remainder, And Or Not"

Question 1(c) [7 marks]

Write a program in Java to reverse the digits of a number for number having three digits. Like reverse of 653 is 356.

Solution

Listing 1. Reverse Digits Program

```java
import java.util.Scanner;

public class ReverseNumber {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter 3-digit number: ");
        int num = sc.nextInt();

        int reverse = 0;
        int temp = num;

        while (temp > 0) {
            reverse = reverse * 10 + temp % 10;
            temp = temp / 10;
        }

        System.out.println("Original: " + num);
        System.out.println("Reversed: " + reverse);
    }
}
```

Algorithm:

- **Extract last digit**: Use modulus operator (%)
- **Build reversed number**: Multiply by 10 and add digit
- **Remove last digit**: Use integer division (/)
- **Repeat**: Until original number becomes 0

Mnemonic

"EBRR: Extract, Build, Remove, Repeat"

Question 1(c OR) [7 marks]

Write a program in Java to add two 3*3 matrices.

Solution

Listing 2. Matrix Addition

```java
import java.util.Scanner;

public class MatrixAddition {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int[][] matrix1 = new int[3][3];
        int[][] matrix2 = new int[3][3];
        int[][] result = new int[3][3];

        // Input first matrix
        System.out.println("Enter first matrix:");
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                matrix1[i][j] = sc.nextInt();
            }
        }

        // Input second matrix
        System.out.println("Enter second matrix:");
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                matrix2[i][j] = sc.nextInt();
            }
        }

        // Add matrices
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                result[i][j] = matrix1[i][j] + matrix2[i][j];
            }
        }

        // Display result
        System.out.println("Sum of matrices:");
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                System.out.print(result[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Matrix Addition Steps:

- **Create arrays**: Three 3x3 integer arrays
- **Input matrices**: Read values for both matrices
- **Add corresponding elements**: `result[i][j] = matrix1[i][j] + matrix2[i][j]`
- **Display result**: Print the sum matrix

Mnemonic

"CIAD: Create, Input, Add, Display"

Question 2(a) [3 marks]

Write a program in Java that shows the use of parameterized Constructor.

Solution

Listing 3. Parameterized Constructor

```java
class Student {
    private String name;
    private int rollNo;

    // Parameterized Constructor
    public Student(String name, int rollNo) {
        this.name = name;
        this.rollNo = rollNo;
    }

    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Roll No: " + rollNo);
    }
}

public class ParameterizedConstructor {
    public static void main(String[] args) {
        Student s1 = new Student("John", 101);
        s1.display();
    }
}
```

Parameterized Constructor Features:

- **Takes parameters**: Accepts values during object creation
- **Initializes instance variables**: Sets object state
- **Same name as class**: Constructor name matches class name
- **No return type**: Constructors don't have return type

Mnemonic

"PISN: Parameters Initialize Same-name No-return"

Question 2(b) [4 marks]

Give the basic syntax of the following terms with an example: (1) To create a Class, (2) To create an Object, (3) To define a Method, (4) To declare a Variable.

Solution

Java Basic Syntax:

Table 3. Syntax Examples

Component	Syntax	Example
Class Creation	`class ClassName { }`	`class Car { }`
Object Creation	`ClassName obj = new ClassName();`	`Car car = new Car();`
Method Definition	`returnType method(params) { }`	`void start() { }`
Variable Decl.	`dataType variableName;`	`int age;`

Listing 4. Complete Syntax Example

```
class Car {                          // Class Creation
    int speed;                       // Variable Declaration

    public void accelerate() {       // Method Definition
        speed += 10;
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();       // Object Creation
    }
}
```

Mnemonic

"COMV: Class Object Method Variable"

Question 2(c) [7 marks]

Write a program in Java which has a class Student having two instance variables enrollmentNo and name. Create 3 objects of Student class in main method and display student's name.

Solution

Listing 5. Student Class Example

```
class Student {
    String enrollmentNo;
    String name;

    // Constructor to initialize student data
    public Student(String enrollmentNo, String name) {
        this.enrollmentNo = enrollmentNo;
        this.name = name;
    }

    // Method to display student name
    public void displayName() {
        System.out.println("Student Name: " + name);
    }
```

```
}

public class StudentDemo {
    public static void main(String[] args) {
        // Creating 3 objects of Student class
        Student s1 = new Student("CS001", "Alice");
        Student s2 = new Student("CS002", "Bob");
        Student s3 = new Student("CS003", "Charlie");

        // Displaying student names
        s1.displayName();
        s2.displayName();
        s3.displayName();
    }
}
```

Program Structure:

- **Class definition**: Student class with instance variables
- **Constructor**: Initialize enrollmentNo and name
- **Method**: `displayName()` to show student name
- **Object creation**: Three Student objects in main method
- **Method calling**: Display names using `displayName()`

Mnemonic

"DID-CC: Define Initialize Display Create Call"

Question 2(a OR) [3 marks]

Write a program in Java that shows the use of Default Constructor.

Solution

Listing 6. Default Constructor

```
class Rectangle {
    int length;
    int width;

    // Default Constructor
    public Rectangle() {
        length = 5;
        width = 3;
        System.out.println("Default constructor called");
    }

    public void displayArea() {
        System.out.println("Area: " + (length * width));
    }
}

public class DefaultConstructor {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        r1.displayArea();
    }
}
```

Default Constructor Features:

- **No parameters**: Takes no arguments
- **Default values**: Sets default values for instance variables
- **Automatic call**: Called when object is created
- **Same name as class**: Constructor name matches class name

Mnemonic

"NDAS: No-parameters Default Automatic Same-name"

Question 2(b OR) [4 marks]

Give four Difference between Procedure Oriented Programming and Object-Oriented Programming.

Solution

POP vs OOP Comparison:

Table 4. POP vs OOP

Aspect	POP	OOP
Approach	Top-down approach	Bottom-up approach
Focus	Functions and procedures	Objects and classes
Data Security	No data hiding, global access	Data encapsulation and hiding
Problem Solving	Divide into functions	Divide into objects
Code Reusability	Limited reusability	High reusability (Inheritance)
Maintenance	Difficult to maintain	Easy to maintain and modify

Key Differences:
- **Structure**: POP uses functions, OOP uses classes
- **Security**: OOP provides better data protection
- **Reusability**: OOP supports inheritance and polymorphism
- **Maintenance**: OOP code is easier to maintain

Mnemonic

"SSRM: Structure Security Reusability Maintenance"

Question 2(c OR) [7 marks]

Write a program in Java which has a class Shape having 2 overloaded methods area (float radius) and area (float length, float width). Display the area of circle and rectangle using overloaded methods.

Solution

Listing 7. Method Overloading

```
class Shape {
    // Method to calculate area of circle
    public void area(float radius) {
        float circleArea = 3.14f * radius * radius;
        System.out.println("Area of Circle: " + circleArea);
```

```
    }

    // Overloaded method to calculate area of rectangle
    public void area(float length, float width) {
        float rectangleArea = length * width;
        System.out.println("Area of Rectangle: " + rectangleArea);
    }
}

public class MethodOverloading {
    public static void main(String[] args) {
        Shape shape = new Shape();

        // Calculate area of circle with radius 5
        shape.area(5.0f);

        // Calculate area of rectangle with length 4 and width 6
        shape.area(4.0f, 6.0f);
    }
}
```

Method Overloading Concepts:

- **Same method name**: Both methods named `area`
- **Different parameters**: One takes radius, other takes length and width
- **Compile-time polymorphism**: Method selected at compile time
- **Parameter differentiation**: Different number or type of parameters

Mnemonic

"SDCP: Same-name Different-params Compile-time Params-diff"

Question 3(a) [3 marks]

Write a program in Java to demonstrate single inheritance.

Solution

Listing 8. Single Inheritance

```
// Parent class
class Animal {
    public void eat() {
        System.out.println("Animal is eating");
    }

    public void sleep() {
        System.out.println("Animal is sleeping");
    }
}

// Child class inheriting from Animal
class Dog extends Animal {
    public void bark() {
        System.out.println("Dog is barking");
    }
}

public class SingleInheritance {
    public static void main(String[] args) {
```

```
        Dog dog = new Dog();

        // Inherited methods from Animal class
        dog.eat();
        dog.sleep();

        // Own method of Dog class
        dog.bark();
    }
}
```

Single Inheritance Features:

- **One parent**: Child class inherits from one parent class
- **extends keyword**: Used to establish inheritance relationship
- **Method inheritance**: Child class inherits parent methods
- **IS-A relationship**: Dog IS-A Animal

Mnemonic

"OEMI: One-parent Extends Method IS-A"

Question 3(b) [4 marks]

Define abstract class in JAVA with example.

Solution

An abstract class is a class that cannot be instantiated and may contain abstract methods (methods without implementation).

Listing 9. Abstract Class Example

```
// Abstract class
abstract class Vehicle {
    String brand;

    // Regular method
    public void displayBrand() {
        System.out.println("Brand: " + brand);
    }

    // Abstract method (no implementation)
    public abstract void start();
    public abstract void stop();
}

// Concrete class extending abstract class
class Car extends Vehicle {
    public Car(String brand) {
        this.brand = brand;
    }

    // Must implement abstract methods
    public void start() {
        System.out.println("Car started with key");
    }

    public void stop() {
        System.out.println("Car stopped with brake");
```

```
    }
}

public class AbstractDemo {
    public static void main(String[] args) {
        Car car = new Car("Toyota");
        car.displayBrand();
        car.start();
        car.stop();
    }
}
```

Abstract Class Features:

- **Cannot instantiate**: Cannot create objects directly
- **Abstract methods**: Methods without body
- **Concrete methods**: Regular methods with implementation
- **Must extend**: Child classes must implement abstract methods

Mnemonic

"CACM: Cannot-instantiate Abstract Concrete Must-extend"

Question 3(c) [7 marks]

Write a program in Java to implement multiple inheritance using interfaces.

Solution

Listing 10. Multiple Inheritance with Interfaces

```
// First interface
interface Flyable {
    void fly();
}

// Second interface
interface Swimmable {
    void swim();
}

// Class implementing multiple interfaces
class Duck implements Flyable, Swimmable {
    private String name;

    public Duck(String name) {
        this.name = name;
    }

    // Implementing fly method
    public void fly() {
        System.out.println(name + " is flying in the sky");
    }

    // Implementing swim method
    public void swim() {
        System.out.println(name + " is swimming in water");
    }

    public void walk() {
```

```
        System.out.println(name + " is walking on land");
    }
}

public class MultipleInheritance {
    public static void main(String[] args) {
        Duck duck = new Duck("Donald");

        // Methods from interfaces
        duck.fly();
        duck.swim();

        // Own method
        duck.walk();
    }
}
```

Multiple Inheritance via Interfaces:

- **Multiple interfaces**: Class can implement multiple interfaces
- **implements keyword**: Used to implement interfaces
- **Must implement all**: All interface methods must be implemented
- **Solves diamond problem**: Avoids ambiguity of multiple inheritance

Mnemonic

"MIMS: Multiple Implements Must-implement Solves-diamond"

Question 3(a OR) [3 marks]

Write a program in Java to demonstrate multilevel inheritance.

Solution

Listing 11. Multilevel Inheritance

```
// Grandparent class
class Animal {
    public void breathe() {
        System.out.println("Animal is breathing");
    }
}

// Parent class inheriting from Animal
class Mammal extends Animal {
    public void giveBirth() {
        System.out.println("Mammal gives birth to babies");
    }
}

// Child class inheriting from Mammal
class Dog extends Mammal {
    public void bark() {
        System.out.println("Dog is barking");
    }
}

public class MultilevelInheritance {
    public static void main(String[] args) {
        Dog dog = new Dog();
```

```
25
26          // Method from Animal class (grandparent)
27          dog.breathe();
28
29          // Method from Mammal class (parent)
30          dog.giveBirth();
31
32          // Own method of Dog class
33          dog.bark();
34      }
35  }
```

Multilevel Inheritance Features:

- **Chain of inheritance**: Child → Parent → Grandparent
- **Multiple levels**: More than two levels of inheritance
- **Transitive inheritance**: Properties passed through levels
- **extends keyword**: Each level uses extends

Mnemonic

"CMTE: Chain Multiple Transitive Extends"

Question 3(b OR) [4 marks]

Define package and write the syntax to create a package with example.

Solution

A package is a namespace that organizes related classes and interfaces, providing access protection and namespace management.

Syntax: `package packageName;`

Listing 12. Package Implementation

```
1  // File: mypackage/Calculator.java
2  package mypackage;
3
4  public class Calculator {
5      public int add(int a, int b) {
6          return a + b;
7      }
8
9      public int subtract(int a, int b) {
10         return a - b;
11     }
12 }
13
14 // File: TestCalculator.java
15 import mypackage.Calculator;
16
17 class TestCalculator {
18     public static void main(String[] args) {
19         Calculator calc = new Calculator();
20         System.out.println("Sum: " + calc.add(10, 5));
21     }
22 }
```

Package Benefits:

- **Namespace management**: Avoids naming conflicts

- **Access control**: Controls class visibility
- **Code organization**: Groups related classes
- **Reusability**: Easy to reuse packaged classes

Mnemonic

"NAOR: Namespace Access Organization Reusability"

Question 3(c OR) [7 marks]

Write a program in Java to demonstrate method overriding.

Solution

Listing 13. Method Overriding

```java
// Parent class
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
    public void move() {
        System.out.println("Animal moves");
    }
}

// Child class overriding parent methods
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks: Woof!");
    }

    @Override
    public void move() {
        System.out.println("Dog runs on four legs");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows: Meow!");
    }

    @Override
    public void move() {
        System.out.println("Cat walks silently");
    }
}

public class MethodOverriding {
    public static void main(String[] args) {
        Animal animal;

        animal = new Dog();
        animal.makeSound();  // Calls Dog's method

        animal = new Cat();
```

```
        animal.makeSound();  // Calls Cat's method
    }
}
```

Method Overriding Features:

- **Same method signature**: Same name, parameters, and return type
- **Runtime polymorphism**: Method decided at runtime
- **@Override annotation**: Optional but recommended
- **IS-A relationship**: Child class overrides parent method

Mnemonic

"SROI: Same-signature Runtime Override IS-A"

Question 4(a) [3 marks]

List and explain different types of errors in Java.

Solution

Java Error Types:

Table 5. Error Types

Error Type	Description
Compile-time	Detected during compilation (Syntax errors, missing semicolons)
Runtime	Occur during program execution (Division by zero, null pointer)
Logical	Program runs but gives wrong output (Incorrect algorithm logic)

- **Compile-time**: Prevented by compiler, must fix before running
- **Runtime**: Program crashes during execution, handled by exceptions
- **Logical**: Hardest to find, program works but results are incorrect

Mnemonic

"CRL: Compile Runtime Logic"

Question 4(b) [4 marks]

What is wrapper class? Explain use of any two wrapper class.

Solution

Wrapper classes provide object representation of primitive data types, converting primitives into objects.

Table 6. Wrapper Classes

Primitive	Wrapper Class
int	Integer
double	Double
boolean	Boolean
char	Character

Listing 14. Wrapper Class Usage

```
// Integer Wrapper
int num = 100;
Integer intObj = Integer.valueOf(num);  // Boxing
int value = intObj.intValue();          // Unboxing
int parsed = Integer.parseInt("123");   // Parsing

// Double Wrapper
Double doubleObj = Double.valueOf(45.67);
double val = Double.parseDouble("123.45");
boolean isNaN = Double.isNaN(doubleObj);
```

Wrapper Class Uses:

- **Collections**: Store primitives in collections
- **Null values**: Can store null unlike primitives
- **Utility methods**: Parsing, conversion methods

Mnemonic

"CNUG: Collections Null Utility Generics"

Question 4(c) [7 marks]

Write a program in Java to develop Banking Application in which user deposits the amount Rs 25000 and then start withdrawing of Rs 20000, Rs 4000 and it throws exception "Not Sufficient Fund" when user withdraws Rs. 2000 thereafter.

Solution

Listing 15. Banking Application with Exception

```
// Custom Exception class
class InsufficientFundException extends Exception {
    public InsufficientFundException(String message) {
        super(message);
    }
}

// Bank Account class
class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    public void deposit(double amount) {
        balance += amount;
        System.out.println("Deposited: Rs." + amount);
        System.out.println("Current Balance: Rs." + balance);
    }

    public void withdraw(double amount) throws InsufficientFundException {
        if (amount > balance) {
            throw new InsufficientFundException("Not Sufficient Fund");
        }
        balance -= amount;
        System.out.println("Withdrawn: Rs." + amount);
```

```
        System.out.println("Remaining Balance: Rs." + balance);
    }

    public double getBalance() { return balance; }
}

public class BankingApplication {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(0);

        try {
            account.deposit(25000);
            account.withdraw(20000);
            account.withdraw(4000);
            // This will throw exception
            account.withdraw(2000);

        } catch (InsufficientFundException e) {
            System.out.println("Exception: " + e.getMessage());
            System.out.println("Available Balance: Rs." + account.getBalance());
        }
    }
}
```

Exception Handling Components:

- **Custom Exception**: `InsufficientFundException` extends `Exception`
- **throw keyword**: Throws exception when balance insufficient
- **try-catch**: Handles the exception to prevent crash

Mnemonic

"CTTM: Custom Throw Try-catch Message"

Question 4(a OR) [3 marks]

Describe the complete lifecycle of a thread.

Solution

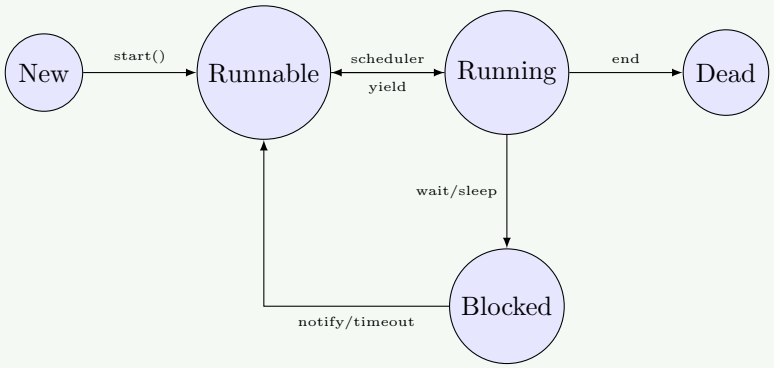

Figure 1. Thread Lifecycle

Thread States:

- **New**: Thread created but not started
- **Runnable**: Ready to run, waiting for CPU
- **Running**: Currently executing
- **Blocked**: Waiting for resource or condition

- **Dead**: Execution completed

Mnemonic

"NRRBD: New Runnable Running Blocked Dead"

Question 4(b OR) [4 marks]

List access specifiers and describe their purpose in JAVA.

Solution

Table 7. Access Specifiers

Specifier	Class	Package	Subclass	World
private	✓	×	×	×
default	✓	✓	×	×
protected	✓	✓	✓	×
public	✓	✓	✓	✓

- **Private**: Encapsulates data within class
- **Default**: Package-private access (no keyword)
- **Protected**: Package + Inheritance access
- **Public**: Universal access for APIs

Mnemonic

"PDPP: Private Default Protected Public"

Question 4(c OR) [7 marks]

Write a program that executes two threads. One thread displays "Thread1" every 1000 milliseconds, and the other displays "Thread2" every 2000 milliseconds. Create the threads by extending the Thread class.

Solution

Listing 16. Multithreading Example

```
class Thread1 extends Thread {
    public void run() {
        try {
            for (int i = 1; i <= 10; i++) {
                System.out.println("Thread1 - Count: " + i);
                Thread.sleep(1000); // 1000ms delay
            }
        } catch (InterruptedException e) {
            System.out.println("Thread1 interrupted");
        }
    }
}

class Thread2 extends Thread {
    public void run() {
```

```
        try {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Thread2 - Count: " + i);
                Thread.sleep(2000); // 2000ms delay
            }
        } catch (InterruptedException e) {
            System.out.println("Thread2 interrupted");
        }
    }
}

public class MultiThreadDemo {
    public static void main(String[] args) {
        Thread1 t1 = new Thread1();
        Thread2 t2 = new Thread2();

        System.out.println("Starting threads...");
        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            System.out.println("Main interrupted");
        }
        System.out.println("Done");
    }
}
```

Multithreading Concepts:

- **Thread class extension**: Both classes extend `Thread`
- **run()**: Contains execution logic
- **sleep()**: Pauses thread for specified milliseconds
- **start()**: Begins execution
- **join()**: Waits for thread completion

Mnemonic

"ERSSJ: Extend Run Sleep Start Join"

Question 5(a) [3 marks]

What is stream class? How are the stream classes classified?

Solution

Stream classes represent a flow of data from source to destination.

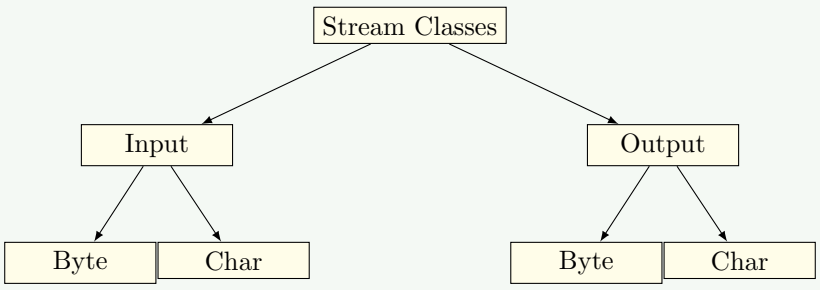

Figure 2. Stream Classification

Stream Types:

Table 8. Stream Types

Type	Purpose	Examples
Input	Read data	FileInputStream, Reader
Output	Write data	FileOutputStream, Writer
Byte	8-bit bytes	InputStream, OutputStream
Char	16-bit unicode	Reader, Writer

Mnemonic

"DDF: Direction Data-type Functionality"

Question 5(b) [4 marks]

Illustrate purpose of method overriding with example.

Solution

Method overriding allows a subclass to provide specific implementation of a method defined in its parent class, enabling runtime polymorphism.

Listing 17. Method Overriding Purpose

```
class Shape {
    public void draw() {
        System.out.println("Drawing generic shape");
    }
}

class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing circle");
    }
}

class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing rectangle");
    }
}

public class Demo {
    public static void main(String[] args) {
        Shape s;
        s = new Circle();
        s.draw(); // Calls Circle's draw

        s = new Rectangle();
        s.draw(); // Calls Rectangle's draw
    }
}
```

Benefits:

- **Runtime polymorphism**: Method selection at runtime
- **Specific implementation**: Customized behavior for subclasses
- **Code flexibility**: Common interface, different implementations

Mnemonic

"RSFD: Runtime Specific Flexibility Dynamic"

Question 5(c) [7 marks]

Write a program in Java to perform read and write operations on a Text file named Abc.txt.

Solution

Listing 18. File Read/Write Operations

```
import java.io.*;

public class FileOperations {
    public static void main(String[] args) {
        String fileName = "Abc.txt";

        // Write Operation
        try {
            FileWriter writer = new FileWriter(fileName);
            writer.write("Hello, this is Java file handling.\n");
            writer.write("End of file content.");
            writer.close();
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            System.out.println("Write error: " + e.getMessage());
        }

        // Read Operation
        try {
            FileReader reader = new FileReader(fileName);
            BufferedReader buffReader = new BufferedReader(reader);

            String line;
            System.out.println("Reading from file:");
            while ((line = buffReader.readLine()) != null) {
                System.out.println(line);
            }
            buffReader.close();
            reader.close();
        } catch (IOException e) {
            System.out.println("Read error: " + e.getMessage());
        }
    }
}
```

File Operation Steps:

- **Create stream**: `FileWriter` or `FileReader`
- **Perform operation**: `write()` or `readLine()`
- **Handle exceptions**: `try-catch` blocks for IOException
- **Close resources**: `close()` to prevent leaks

Mnemonic

"CPHC: Create Perform Handle Close"

Question 5(a OR) [3 marks]

Explain InputStream.

Solution

InputStream is an abstract class representing an input stream of bytes.

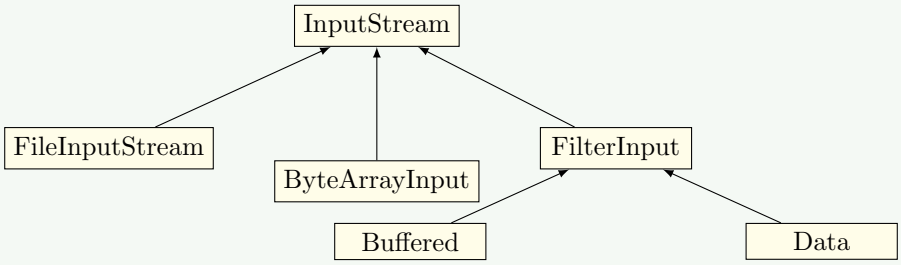

Figure 3. InputStream Hierarchy

Common Methods:

- `read()`: Reads single byte
- `read(byte[])`: Reads bytes into array
- `close()`: Closes the stream
- `available()`: Returns estimated available bytes

Mnemonic

"ABIR: Abstract Byte Input Resource"

Question 5(b OR) [4 marks]

Define package in JAVA. Write how package can be implemented in Java with proper syntax and one example.

Solution

A package is a namespace organizing related classes.

Steps to Implement:

- **Create directory**: Folder matching package name
- **Declare package**: `package name;` as first line
- **Compile**: Use `javac -d . File.java`
- **Run**: Use `java package.File`

Listing 19. Package Syntax Example

```
// File: utilities/MathOps.java
package utilities;

public class MathOps {
    public static int add(int a, int b) {
        return a + b;
    }
}
```

```
// File: Test.java
import utilities.MathOps;

public class Test {
    public static void main(String[] args) {
        System.out.println(MathOps.add(5, 10));
    }
}
```

Mnemonic

"NAOR: Namespace Access Organization Reusability"

Question 5(c OR) [7 marks]

Write a program in Java to demonstrate use of List. 1) Create ArrayList and add weekdays (in string form), 2) Create LinkedList and add months (in string form). Display both List.

Solution

Listing 20. List Demonstration

```
import java.util.*;

public class ListDemo {
    public static void main(String[] args) {
        // 1. ArrayList for Weekdays
        ArrayList<String> weekdays = new ArrayList<>();
        weekdays.add("Monday");
        weekdays.add("Tuesday");
        weekdays.add("Wednesday");

        System.out.println("Weekdays (ArrayList):");
        for(String day : weekdays) {
            System.out.println(day);
        }

        // 2. LinkedList for Months
        LinkedList<String> months = new LinkedList<>();
        months.add("January");
        months.add("February");
        months.add("March");

        System.out.println("\nMonths (LinkedList):");
        for(String month : months) {
            System.out.println(month);
        }

        // Operations
        System.out.println("\nFirst Month: " + months.getFirst());
        System.out.println("ArrayList Size: " + weekdays.size());
    }
}
```

List Features:

Table 9. ArrayList vs LinkedList

Feature	**ArrayList**	**LinkedList**
Structure	Dynamic Array	Doubly Linked List
Access	Fast O(1)	Slow O(n)
Insert/Delete	Slow (Shift needed)	Fast (Ptr change)

Mnemonic

"DODI: Dynamic Ordered Duplicate Index-based"