Scheme of Evaluation for OOPJ Mid-Semester Examination - March 2025

OOPJ Course Committee

March 4, 2025

Question 1: Answer All the Questions [1 Mark × 5]

a) Explain the significance of static variables and methods in Java. Discuss how static variables help in maintaining shared data among multiple objects.

Static variables and methods in Java belong to the class rather than any specific instance of the class. This has several significant implications:

- **Static Variables:** These are shared across all instances of a class. They are initialized only once, at the time of class loading, and maintain a single copy in memory. This is particularly useful for maintaining shared data among multiple objects, such as counters, configuration settings, or constants. For example, a static variable like `count` in a class can track the number of objects created, ensuring all instances access and modify the same value.
- **Static Methods:** These can be invoked without creating an instance of the class, making them useful for utility methods (e.g., `Math.max()`). They cannot access non-static (instance) variables or methods directly, as they operate on the class level.
- **Maintaining Shared Data:** Static variables are ideal for scenarios where data needs to be common across all objects, reducing memory usage and ensuring consistency. For instance, a `static int totalEmployees` in an `Employee` class can track the total number of employees across all instances, ensuring all objects reference the same value.

b) Can a class be both final and abstract? Justify your answer.

No, a class cannot be both `final` and `abstract` in Java. Here's the justification:

- **Final Class:** A `final` class cannot be subclassed or extended, meaning no other class can inherit from it. This restricts inheritance entirely.
- **Abstract Class:** An `abstract` class is designed to be subclassed, as it may contain abstract methods (methods without implementation) that must be implemented by subclasses. An abstract class cannot be instantiated directly and exists to provide a blueprint for other classes.

- **Contradiction:** If a class were both `final` and `abstract`, it would be impossible to fulfill the requirements of both keywords. A `final` class cannot be extended, but an `abstract` class requires extension to provide concrete implementations of its abstract methods. Hence, Java does not allow this combination.

c) Find and explain the output.

```
class Test {
    public static void main(String args[]) {
        float f = 1.2;
        boolean b = 1;
        System.out.println(f);
        System.out.println(b);
    }
}
```

Output Explanation:

- `float f = 1.2;` assigns the floating-point value 1.2 to variable `f`. When printed, it outputs `1.2`.
- `boolean b = 1;` attempts to assign an integer value (1) to a boolean variable. In Java, boolean values can only be `true` or `false`, not integers. This line will result in a compilation error because Java does not perform implicit conversion from `int` to `boolean`.
- Therefore, the program will not compile, and no output will be produced. If corrected (e.g., `boolean b = true;`), it would print `1.2` followed by `true`.

d) Find and explain the output.

```
class H {
    public static void main(String args[]) {
        System.out.println(fun());
    }

    int fun() {
        return 20;
    }
}
```

Output Explanation:

- The method `fun()` is defined as an instance method (non-static, returning `int`) with `return 20;`.
- However, `fun()` is called in `main()` using `System.out.println(fun());`, but `main()` is a static method and cannot directly call a non-static method without an instance of the class.
- This will result in a compilation error: "non-static method `fun()` cannot be referenced from a static context."

- If `fun()` were declared as `static int fun()`, the output would be 20.

e) Can the "main" method be overridden and overloaded in Java? Explain your answer.

- **Overriding the main method:** No, the `main` method cannot be overridden in Java. Overriding applies to instance methods in subclasses, but `main` is a static method. Static methods belong to the class, not instances, and are not subject to polymorphism or overriding. A subclass cannot override the `main` method of its superclass because static methods are resolved at compile-time, not runtime.

- **Overloading the main method:** Yes, the `main` method can be overloaded. Overloading means creating multiple methods with the same name but different parameter lists in the same class. For example, you can have:

```
public static void main(String[] args) { ... }
public static void main(String args) { ... } // Overloaded
```

However, the JVM specifically looks for `public static void main(String[] args)` as the entry point to run the program. Overloaded versions of `main` will not be executed as the program's entry point unless explicitly called.

Question 2: [3+2=5 Marks]

a) Write a Java program to create a class known as BankAccount with the member variables account number, account holder name, balance and member methods deposit(), withdraw(). Create a subclass called SavingAccount that overrides the withdraw() method to prevent withdrawals if the account balance falls below one thousand for saving account and two thousand for recurring account. Define the Driver class (a class that contains main method) to test it and also to display the current balance.

```
class BankAccount {
    int accountNumber;
    String accountHolderName;
    double balance;

    public BankAccount(int accountNumber, String
        accountHolderName, double balance) {
        this.accountNumber = accountNumber;
        this.accountHolderName = accountHolderName;
        this.balance = balance;
    }

    void deposit(double amount) {
```

```
13                  if (amount > 0) {
14                      balance += amount;
15                      System.out.println("Deposited: INR " +
                            amount + ". New balance: INR " + balance);
16                  } else {
17                      System.out.println("Invalid deposit
                            amount.");
18                  }
19              }
20
21              void withdraw(double amount) {
22                  if (amount > 0 && amount <= balance) {
23                      balance -= amount;
24                      System.out.println("Withdrawn: INR " +
                            amount + ". New balance: INR " + balance);
25                  } else {
26                      System.out.println("Invalid withdrawal
                            amount or insufficient balance.");
27                  }
28              }
29
30              double getBalance() {
31                  return balance;
32              }
33          }
34
35          class SavingAccount extends BankAccount {
36              public SavingAccount(int accountNumber, String
                    accountHolderName, double balance) {
37                  super(accountNumber, accountHolderName, balance);
38              }
39
40              @Override
41              void withdraw(double amount) {
42                  if (amount > 0) {
43                      if (this instanceof RecurringAccount) {
44                          if (balance - amount < 2000) {
45                              System.out.println("Cannot withdraw.
                                    Balance cannot fall below INR
                                    2000 for recurring account.");
46                              return;
47                          }
48                      } else {
49                          if (balance - amount < 1000) {
50                              System.out.println("Cannot withdraw.
                                    Balance cannot fall below INR
                                    1000 for saving account.");
51                              return;
52                          }
53                      }
54                      balance -= amount;
```

```
                    System.out.println("Withdrawn: INR " +
                        amount + ". New balance: INR " + balance);
                } else {
                    System.out.println("Invalid withdrawal
                        amount.");
                }
            }
        }

        class RecurringAccount extends SavingAccount {
            public RecurringAccount(int accountNumber, String
                accountHolderName, double balance) {
                super(accountNumber, accountHolderName, balance);
            }
        }

        class Driver {
            public static void main(String[] args) {
                SavingAccount saving = new SavingAccount(101,
                    "John Doe", 5000);
                RecurringAccount recurring = new
                    RecurringAccount(102, "Jane Doe", 3000);

                System.out.println("Saving Account:");
                saving.deposit(500);
                saving.withdraw(3500); // Should fail, balance
                    would fall below 1000
                saving.withdraw(1000); // Should succeed
                System.out.println("Current balance: INR " +
                    saving.getBalance());

                System.out.println("\nRecurring Account:");
                recurring.deposit(1000);
                recurring.withdraw(1500); // Should fail,
                    balance would fall below 2000
                recurring.withdraw(500); // Should succeed
                System.out.println("Current balance: INR " +
                    recurring.getBalance());
            }
        }
```

b) Write a class to create a copy of a previously existing object present for the same class.

```
1   class Student {
2       String name;
3       int age;
4
5       // Constructor
6       public Student(String name, int age) {
```

```
7                 this.name = name;
8                 this.age = age;
9             }
10
11            // Copy constructor
12            public Student(Student other) {
13                this.name = other.name;
14                this.age = other.age;
15            }
16
17            // Display method
18            void display() {
19                System.out.println("Name: " + name + ", Age: " +
                      age);
20            }
21
22            public static void main(String[] args) {
23                Student student1 = new Student("Alice", 20);
24                System.out.println("Original Student:");
25                student1.display();
26
27                // Creating a copy
28                Student student2 = new Student(student1);
29                System.out.println("Copied Student:");
30                student2.display();
31
32                // Modify the copy to show independence
33                student2.name = "Bob";
34                System.out.println("After modifying copy:");
35                System.out.println("Original: ");
36                student1.display();
37                System.out.println("Modified copy: ");
38                student2.display();
39            }
40        }
```

Question 3: [3+2=5 Marks]

a) Define two user defined exception class named InvalidNameException and InvalidDOBException. Create a class KIITEE2025 which holds data members name, dd, mm, yyyy and a method getApplicantDetails() to accept name and date of birth of KIITEE2025 applicants, and display the details. The getApplicantDetails() method should be able to handle above user defined exceptions. InvalidNameException is thrown, if length of name is less than 2 and InvalidDOBException is thrown, if yyyy is greater than 2010 or less than 1970. Write a complete Java program to implement the above scenario.

Note: Students who have used `e.getMessage()` instead of using overridden `toString()` in the catch blocks should also receive full credits, as both approaches are valid for displaying exception details.

```
1   import java.util.Scanner;
2
3   class InvalidNameException extends Exception {
4       String name;
5       InvalidNameException(String name) {
6           this.name = name;
7       }
8       public String toString() {
9           return "InvalidNameException, length of " + name
                + " is less than 2.";
10      }
11  }
12
13  class InvalidDOBException extends Exception {
14      int year;
15      InvalidDOBException(int year) {
16          this.year = year;
17      }
18      public String toString() {
19          return "InvalidDOBException, " + year + " is
                greater than 2010 or less than 1970.";
20      }
21  }
22
23  class KIITEE2025 {
24      static void getApplicantDetails() throws
            InvalidNameException, InvalidDOBException {
25          String name;
```

```
                int dd, mm, yyyy;
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter Name:");
                name = sc.nextLine();
                System.out.println("Enter DOB (DD MM YYYY):");
                dd = sc.nextInt();
                mm = sc.nextInt();
                yyyy = sc.nextInt();

                if (name.length() < 2) {
                    throw new InvalidNameException(name);
                }
                if (yyyy < 1970 || yyyy > 2010) {
                    throw new InvalidDOBException(yyyy);
                }

                System.out.println("Name-" + name + ", DOB-" +
                   dd + " " + mm + " " + yyyy);
            }

            public static void main(String args[]) {
                try {
                    getApplicantDetails();
                } catch (InvalidNameException e) {
                    System.out.println("Exception Occurred: " +
                       e.toString());
                } catch (InvalidDOBException e) {
                    System.out.println("Exception Occurred: " +
                       e.toString());
                }
            }
        }
```

b) Write a program in Java to create an interface Flyable with an abstract method fly() and Swimmable is another interface with an abstract method swim(). Bird class implements both Flyable and Swimmable interfaces, providing concrete implementations for fly() and swim() methods. In the Main class, an object of Bird is created, and both fly() and swim() methods are called to demonstrate the implementation of multiple interfaces.

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

        interface Swimmable {
            void swim();
        }
```

```
8
9           class Bird implements Flyable, Swimmable {
10              @Override
11              public void fly() {
12                  System.out.println("The bird is flying in the
                        sky.");
13              }
14
15              @Override
16              public void swim() {
17                  System.out.println("The bird is swimming in the
                        water.");
18              }
19          }
20
21          class Main {
22              public static void main(String[] args) {
23                  Bird bird = new Bird();
24                  bird.fly();
25                  bird.swim();
26              }
27          }
```

Question 4: [3+2=5 Marks]

a) Write a program that prompts the user to input two numbers, first and diff. The program then creates a one dimensional array of 10 elements and initializes them with an arithmetic sequence. The first number of the sequence is the first value and the next number is generated by adding the diff to the number preceding it. This formula is repeated for the rest of the sequence. E.g., if first = 11 and diff = 4, then the arithmetic sequence will be 11, 15, 19, 23, 27, 31 ... and so on.

```
1           import java.util.Scanner;
2
3           public class ArithmeticSequence {
4               public static void main(String[] args) {
5                   Scanner scanner = new Scanner(System.in);
6                   System.out.print("Enter the first number: ");
7                   int first = scanner.nextInt();
8                   System.out.print("Enter the difference: ");
9                   int diff = scanner.nextInt();
10
11                  int[] sequence = new int[10];
12                  for (int i = 0; i < 10; i++) {
13                      sequence[i] = first + (i * diff);
```

```
                }

                System.out.println("Arithmetic Sequence:");
                for (int num : sequence) {
                    System.out.print(num + " ");
                }
                System.out.println();
                scanner.close();
            }
        }
```

b) Create a package Department having a class Student. The students class is having the data members - name, semester, branch, roll-no, and cgpa. The class is also having a constructor and a show() method which is used to display the various attributes of student object. Finally design an application class Demo where the above package is imported and functionality of Student class is tested.

```
        // File: Department/Student.java
        package Department;

        public class Student {
            private String name;
            private int semester;
            private String branch;
            private int rollNo;
            private double cgpa;

            public Student(String name, int semester, String
               branch, int rollNo, double cgpa) {
                this.name = name;
                this.semester = semester;
                this.branch = branch;
                this.rollNo = rollNo;
                this.cgpa = cgpa;
            }

            public void show() {
                System.out.println("Student Details:");
                System.out.println("Name: " + name);
                System.out.println("Semester: " + semester);
                System.out.println("Branch: " + branch);
                System.out.println("Roll No: " + rollNo);
                System.out.println("CGPA: " + cgpa);
            }
        }

```

```java
// File: Demo.java
import Department.Student;

public class Demo {
    public static void main(String[] args) {
        Student student = new Student("Alice Smith", 4,
            "Computer Science", 12345, 8.5);
        student.show();
    }
}
```

Question 5: [3+2=5 Marks]

a) Is dynamic method dispatch necessary in Java? Write a program in Java to create a class Bank having ROI data member and find_ROI() member function. Derive three classes HDFC, ICICI, BOI with find_ROI() function and find the rate of interest of different Banks using dynamic method dispatch concept.

```java
class Bank {
    protected double ROI;

    public Bank(double roi) {
        this.ROI = roi;
    }

    public double find_ROI() {
        return ROI;
    }
}

class HDFC extends Bank {
    public HDFC(double roi) {
        super(roi);
    }

    @Override
    public double find_ROI() {
        return ROI + 0.5; // HDFC offers 0.5% higher
            interest
    }
}

class ICICI extends Bank {
    public ICICI(double roi) {
        super(roi);
    }

```

```
29              @Override
30              public double find_ROI() {
31                  return ROI + 0.3; // ICICI offers 0.3% higher
                        interest
32              }
33          }
34
35          class BOI extends Bank {
36              public BOI(double roi) {
37                  super(roi);
38              }
39
40              @Override
41              public double find_ROI() {
42                  return ROI + 0.2; // BOI offers 0.2% higher
                        interest
43              }
44          }
45
46          public class Main {
47              public static void main(String[] args) {
48                  Bank bank;
49
50                  bank = new HDFC(5.0);
51                  System.out.println("HDFC Rate of Interest: " +
                        bank.find_ROI() + "%");
52
53                  bank = new ICICI(5.0);
54                  System.out.println("ICICI Rate of Interest: " +
                        bank.find_ROI() + "%");
55
56                  bank = new BOI(5.0);
57                  System.out.println("BOI Rate of Interest: " +
                        bank.find_ROI() + "%");
58              }
59          }
```

Is dynamic method dispatch necessary in Java? Dynamic method dispatch is not strictly necessary but is a powerful feature of Java's object-oriented programming. It enables runtime polymorphism, allowing the appropriate method to be called based on the object's actual type, not the reference type. This is essential for flexibility, extensibility, and maintaining clean, reusable code, especially in inheritance hierarchies and frameworks.

b) What is abstraction? How encapsulation, polymorphism and inheritance facilitates abstraction? Explain with proper examples.

- **Abstraction:** Abstraction in Java is the process of hiding complex implementation details and showing only the essential features of an object. It helps in reducing

complexity and focusing on what an object does rather than how it does it. It is achieved using abstract classes and interfaces.

- **Encapsulation facilitates abstraction:** Encapsulation involves bundling data and methods that operate on that data within a single unit (class) and restricting direct access to some of an object's components. This hides the internal details and exposes only what is necessary, promoting abstraction. For example:

```
class Car {
    private int speed; // Private data
    public void setSpeed(int s) { // Public
       method to set speed
        if (s > 0) speed = s;
    }
    public int getSpeed() { // Public method to
       get speed
        return speed;
    }
}
```

Here, the internal `speed` variable is hidden, and users interact via methods, abstracting the implementation.

- **Polymorphism facilitates abstraction:** Polymorphism allows methods to be used in a way that depends on the object invoking them, abstracting the specific type. It enables writing code that works with a superclass or interface, ignoring the specific subclass implementation. For example:

```
abstract class Animal {
    abstract void sound();
}
class Dog extends Animal {
    void sound() { System.out.println("Bark"); }
}
class Cat extends Animal {
    void sound() { System.out.println("Meow"); }
}
public class Test {
    public static void makeSound(Animal a) {
        a.sound(); // Abstracted behavior
    }
    public static void main(String[] args) {
        makeSound(new Dog()); // Outputs "Bark"
        makeSound(new Cat()); // Outputs "Meow"
    }
}
```

The `makeSound` method abstracts the specific animal type, focusing on the `sound()` behavior.

- **Inheritance facilitates abstraction:** Inheritance allows a subclass to inherit properties and methods from a superclass, enabling abstraction by providing a

common interface or base behavior that can be extended or overridden. For example:

```
abstract class Shape {
    abstract double area();
}
class Circle extends Shape {
    private double radius;
    public Circle(double radius) { this.radius =
        radius; }
    double area() { return Math.PI * radius *
        radius; }
}
class Rectangle extends Shape {
    private double length, width;
    public Rectangle(double length, double
        width) {
        this.length = length;
        this.width = width;
    }
    double area() { return length * width; }
}
```

The `Shape` class abstracts the concept of area calculation, and subclasses provide specific implementations, hiding the details from the user.