INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, SRI CITY

TERM-II EXAMINATION – MONSOON 2024

Object Oriented Programming: Key

CSE:UG2 /PC **Date: 16-10-2024**

Duration: 90 Mins (03:30-05:00 PM) **Max. Marks: 20**

Instructions: **Roll No:________________**

1. All questions are compulsory.
2. Write the answers legibly.
3. Attach the question paper with the answer sheet

Answer all the questions
PART-A

1.	Fill in the blanks. a. Final methods cannot be overridden and a call to one can be resolved at compile time. This is referred to as early binding. b. System.gc() method can be used to request the JVM to run garbage collector in Java application.	**[2 Marks]**

2	What will be the output/possible error of the following code snippets.
	a. `class Parent {`
	`    void show() {`
	`    System.out.println("Parent");`
	`    }}`
	` class Child extends Parent {`
	`    void show() {`
	`    System.out.println("Child");`
	` }}`
	` public class Test {`
	`     public static void main(String[] args) {`
	`     Parent obj = new Child();`
	`     obj.show();`
	`}}`
		[1 Mark]

	Child	

b.

```
interface TestInterface {
        default void show() {
            System.out.println("Static Method Executed");
        } }
    class InterfaceDemo implements TestInterface {
        void show() {
            System.out.println("Class Method Executed");
        }
        public static void main(String args[]) {
            InterfaceDemo d = new InterfaceDemo();
            d.show();
        } }
```

[1 Mark]

Compilation error

```
void show() {
     ^
 attempting to assign weaker access privileges; was public
```

c.

```
class A {
    int i=100;
    void display() {
        System.out.println(i);
    } }
  class B extends A{
    int j;
    void display(){
        System.out.println(j);
            super.display();
    }}
  class inheritance_demo {
    public static void main(String args[]){
        B obj = new B();
        obj.i=1;
        obj.j=2;
        obj.display();
    } }
```

[1 Mark]

2
1

3. In an interface, which of the following is NOT allowed? **[1 Mark]**

A) Static methods
B) Instance variables
C) Default methods
D) Private methods

4.	Which statement about constructors in an abstract class is true? A) An abstract class cannot have a constructor. **B) Constructors of abstract classes can be called from subclasses.** C) Constructors of abstract classes cannot initialize instance variables. D) Abstract classes can only have private constructors.	**[1 Mark]**
5.	Select the correct statement/s. I. We use extends keyword for implementing the methods of abstract classes, II. We have to use the implements keyword for inheriting all the members of parent classes.	**[1 Mark]**
	Both are false	

PART-B

6.	Design a payment system that supports multiple payment methods, including credit cards and digital wallets. Create an abstract class named Payment with a protected field amount, an abstract method processPayment(), and a concrete method refund() for processing refunds. Define an interface PaymentMethod with methods for validating payment methods and retrieving formatted payment details, including a constant variable for the payment type. Implement two concrete classes: CreditCardPayment, which must validate that the card number is 16 digits long and starts with a valid prefix, and DigitalWalletPayment, which should ensure the wallet ID starts with "WALLET-" and is alphanumeric. Your design should demonstrate key object-oriented principles, including inheritance and encapsulation.	**[4 marks]** **APV**

```java
// PaymentSystem.java

// Abstract Class     1 Mark
abstract class Payment {
    protected double amount;

    public Payment(double amount) {
        this.amount = amount;
    }

    public abstract void processPayment();

    public void refund() {
        System.out.println("Refund processed for amount: " + amount);
    }
}

// Interface     1 Marks
interface PaymentMethod {
    String PAYMENT_TYPE = "Generic Payment";

    boolean validate();
    String getFormattedPaymentDetails();
}

// Concrete Class for Credit Card Payment 1 Mark
class CreditCardPayment extends Payment implements PaymentMethod {
    private String cardNumber;

    public CreditCardPayment(double amount, String cardNumber) {
        super(amount);
        this.cardNumber = cardNumber;
    }

    @Override
    public void processPayment() {
        if (validate()) {
            System.out.println("Processing credit card payment of: " + amount);
        } else {
            System.out.println("Invalid credit card details.");
        }
    }

    @Override
    public boolean validate() {
        return cardNumber.length() == 16 && (cardNumber.startsWith("4") ||
cardNumber.startsWith("5") || cardNumber.startsWith("6"));
    }

    @Override
    public String getFormattedPaymentDetails() {
        return "Credit Card Payment: Amount - " + amount + ", Card Number - " +
cardNumber.replaceAll("(?<=\\d{4})\\d(?=\\d{4})", "*");
    }
}

// Concrete Class for Digital Wallet Payment 1 Mark
class DigitalWalletPayment extends Payment implements PaymentMethod {
    private String walletId;

    public DigitalWalletPayment(double amount, String walletId) {
        super(amount);
        this.walletId = walletId;
    }

    @Override
    public void processPayment() {
        if (validate()) {
            System.out.println("Processing digital wallet payment of: " + amount);
        } else {
            System.out.println("Invalid digital wallet ID.");
        }
    }

    @Override
    public boolean validate() {
        return walletId.startsWith("WALLET-") &&
walletId.substring(7).matches("[a-zA-Z0-9]+");
    }

    @Override
    public String getFormattedPaymentDetails() {
        return "Digital Wallet Payment: Amount - " + amount + ", Wallet ID - " +
walletId;
    }
}

// Main Class to Demonstrate the Payment System // Not necessary
public class PaymentSystem {
    public static void main(String[] args) {
        Payment creditCardPayment = new CreditCardPayment(150.75,
"5123456789012346");
```

	```
	        creditCardPayment.processPayment();
	        System.out.println(creditCardPayment.getFormattedPaymentDetails());
	        creditCardPayment.refund();

	        Payment digitalWalletPayment = new DigitalWalletPayment(89.99,
	"WALLET-12345");
	        digitalWalletPayment.processPayment();
	        System.out.println(digitalWalletPayment.getFormattedPaymentDetails());
	        digitalWalletPayment.refund();
	    }
	}
	```

7.	Discuss what is a checked exception and unchecked exception, and describe the occurrence of exceptions, any two examples for each, for checked and unchecked exceptions.
	
	Ans:
	Checked Exceptions are the exceptions that are checked at compile time. Checked exceptions represent invalid conditions in areas outside the immediate control of the program (like memory, network, file system, etc.).
	
	Ex1: FileNotFoundException
	This exception occurs when we try to access that file which is not available in the system.
	```
	import java.io.*;
	class FileNotFoundException {
	    public static void main(String[] args)
	    {

	        // Reading file from path in local directory
	        FileReader file = new FileReader("C:\\test\\a.txt");

	        // Creating object as one of ways of taking input
	        BufferedReader fileInput = new BufferedReader(file);

	        // Printing first 3 lines of file "C:\test\a.txt"
	        for (int counter = 0; counter < 3; counter++)
	            System.out.println(fileInput.readLine());

	        // Closing file connections
	        // using close() method
	        fileInput.close();
	    }
	```
		[4 marks]

}

Ex2: ClassNotFoundException
This Exception occurs when methods like Class.forName() and LoadClass Method etc. are unable to find the given class name as a parameter.

```
import java.io.*;
class ClassNotFoundException {

    public static void main(String[] args)
    {
        // Calling the class gfg which is not present in the
        // current class temp instance of calling class
        Class temp = Class.forName("gfg");

        // It will throw ClassNotFoundException
    }
}
```

Unchecked exceptions are the exceptions that are not checked at compile time. These exceptions are usually caused by programming errors, such as attempting to access an index out of bounds in an array or attempting to divide by zero.

Ex1: Arithmetic Exception
The Arithmetic Exception occurs when it attempts to divide a number by zero.It compiles fine, but it throws ArithmeticException when run. The compiler allows it to compile because ArithmeticException is an unchecked exception.

```
class ExOfArithmaticException {
    public static void main(String args[])
    {
        int x = 0;
        int y = 10;
        int z = y / x;      //Arithmetic Exception
    }
}
```

Ex2: ArrayIndexOutOfBoundsException
This exception is thrown when you attempt to access an array index that is out of bounds.

```
public class ArrayIndexOutOfBoundException {

 public static void main(String[] args) {
String[] arr = {"Rohit","Shikar","Virat","Dhoni"};
//Declaring 4 elements in the String array

       for(int i=0;i<=arr.length;i++) {

//Here, no element is present at the iteration number arr.length, i.e 4
            System.out.println(arr[i]);
//So it will throw ArrayIndexOutOfBoundException at iteration 4
       }

   }

}
```

8.

a. Write two differences between static and non-static nested classes in JAVA with example.

[2+2 Marks] PS

Ans: (If any two differences are written with example then you can give two marks.)

(i) In the case of non-static nested classes or inner classes, without an outer class object existing, there cannot be an inner class object.
But in the case of a static nested class, without an outer class object existing, there may be a static nested class object.
(ii) Non-static nested class can access both static and non-static members of enclosing class while static class can only access the static members of the enclosing class directly.
(iii) Static nested class can declare both static and non-static members.
Non-static nested classes cannot declare static fields and static methods. It has to be declared in either static or top level types.

```
class A
{
   static int a=10;
   int  b=9;
   class B
   {
      // static int x; not allowed here
     void display1(){
            System.out.println("outer_a = " + a); //both variables accessible
            System.out.println("outer_b = "+ b);
     }
   }
   static class C
   {
      static int x; // allowed here
    void display2(){
            System.out.println("outer_a = " + a);
            //System.out.println("outer_b = "+ b); Not allowed here
      }
    }
}

class Test
{
    public static void main(String… str)
    {
        A a = new A();
        // Non-Static Inner Class
        // Requires enclosing instance
        A.B obj1 = a.new B();
        obj1.display1();
        // Static Inner Class
        // No need for reference of object to the outer class
        A.C obj2 = new A.C();
       obj2.display2();
    }
}
```

b. Explain the concept of Composition in JAVA with an example.

Ans. (Explanation 1 mark and example 1 mark)
Has-a represents composition. In a has-a relationship, an object contains as member references to other objects. Such relationships create classes by composition of existing classes. For example, given the classes Employee, BirthDate and TelephoneNumber, it's improper to say that an Employee is a BirthDate or that an Employee is a TelephoneNumber. However, an Employee has a BirthDate, and an Employee has a TelephoneNumber.

```
public class Date
 {
            private int month;    // 1-12
            private int day;      // 1-31 based on month
            private int year;    // any year
             public Date( int theMonth, int theDay, int theYear )
            {
                        month = theMonth;
                        year = theYear;     // could validate year
                        day =  theDay ;
                        System.out.printf("Date object constructor for date %s\n",
this);}
            // return a String of the form month/day/year
             public String toString()
            {
                        return String.format( "%d/%d/%d", month, day, year ); } // end
method toString
}
public class Employee
{
        private String firstName;
        private String lastName;
        private Date birthDate;
        private Date hireDate;
// constructor to initialize name, birth date and hire date
        public Employee( String first, String last, Date dateOfBirth, Date
dateOfHire ){
                firstName = first;
                lastName = last;
                birthDate = dateOfBirth;
                hireDate = dateOfHire;
         }
         public String toString(){
        return String.format( "%s, %s Hired: %s Birthday: %s", lastName,
firstName, hireDate, birthDate );}
}
public class EmployeeTest {
public static void main( String[] args )
{
Date birth = new Date( 7, 24, 1949 );
Date hire = new Date( 3, 12, 1988 );
Employee employee = new Employee( "Bob", "Blue", birth, hire );
System.out.println( employee );
 } // end main
 } // end class EmployeeTes
```