Ans : 10 (Extend part) IT23051

Palindrome cheaking program

```
import java.util.Scanner;
public class Palindrome checker {
  static boolean isNumberPalindrome (int num) {
  int original = num, reverse = 0;
  while (num > 0) {
   int digit = num % 10;
   reverse = reverse * 10 + digit;
   num /= 10;
  } return original == reverse;
  }
  static boolean isStringPalindrome (String str) {
  int left = 0, right = str.length() - 1;
  while (left < right) {
  if (str.charAt (left) != str.charAt (right)) {
   return false;
  } left++;
   right--;
  } return true;
  }
```

```
public static void main (String[] args) {
Scanner scanner = new scanner (System.in);
  System.out.print ("Enter a number : ");
  int num = scanner.nextInt ();
System.out.println(num +" is "+ (isNumberpalindrome(num)
      ?"" : "not ")+"a palindrome .");
scanner.nextline ();
System .out.print ("Enter a string : ");
String str = scanner.nextLine();
System.out.println (str +"is" + (isstringpalindrome(str)?
    " " : "not") + "a palindrome.");
scanner.close();
}
}
```

Ans to Qn : 11

Abstraction: Abstraction is a concept of hiding the implementation details and showing only the necessary features of an object. It allows the users to work with high-level functionalities without worrying about the underlying complexity

Encapsulation: Encapsulation is a process of wrapping data (variables) and methods (functions) into a single unit (class) and restricting direct access of the data by using access modifiers like as private, protected and public

Difference between abstract class and interface

Abstract class:

Defination: A class that contains at least one varctual function

Method implementation: can have both abstract (pure varctual) and concrete methods.

IT23051

Access modifiers: can have private, public and protected members.

constructor: Can have a constructor.

Multiple Inheritance: Support single inheritance

usage: used when we want to share common code among multiple related classes.

Inheritence: Interface:

Defination: A completly abstract class that contains only abstract methods.

Method Implementation: Can only have abstract methods in Java, newer versions allow default methods)

Access modifier: All members are public by default

constructor: Can not have a constructor.

Multiple Inheritance: Support multiple inheritance.

usages: Used when we want to define a constract that multiple class must follow.

Answer : 12

```
class BaseClass {
    void printResult(String result) {
        System.out.println(result);
    }
}
class SumClasses extends BaseClass {
    void computeSumSeries() {
        double sum = 0.0;
        for (double i = 1.0; i >= 0.1; i -= 0.1) {
            sum += i;
        }
        printResult("Sum of Series: " + sum);
    }
}
class DivisorMultipleClass extends BaseClass {
    int gcd(int a, int b) {
        if (b == 0) return a;
        return gcd(b, a % b);
    }
    int lcm(int a, int b) {
        return (a * b) / gcd(a, b);
    }
```

```
    void computeGCDLCM(int a, int b){
    printResult("GCD of "+a+" and "+b+": "+gcd(a,b));
    printResult("LCM of "+a+" and "+b+": "+lcm(a,b));
}
}
class NumberConversionClass extends BaseClass {
    void pr(String massage){
    System.out.println("[Formatted Output]: "+message);
}
}
public class MainClass {
    public static void main(String[] args){
    SumClass sumobj = new SumClass();
    Sumobj.computeSumSeries();
    DivisorMultipleClass gcdlcmObj = new DivisorMultipleClass();
    gcdLcmobj.compute GCDLCM(24,36);
    NumberConversionClass conversionObj = new NumberConversionClass();
    conversionObj.convertNumber(45);
    CustomPrintClass printObj = new CustomPrintClass();
    PrintObj.pr("Inheritance in Java is powerful!");
}
}
```

Answer : 13

```
import java.util.Date;
class GeometricObjee {
  private string color;
  private boolean filled;
  private Date dateCreated;
  public GeometricObjcect() {
  this.color = "white";
  this.filled = false;
  this.dateCreated = new Date();
}
public GeometricObject (String color, boolean filled){
this.color = color;
this.filled = filled;
this.dateCreated = new Date();
}
public string getColor () {
    return color;
}
public void setColor (string color) {
    this.color = color;
}
public boolean isFilled () {
  return filled;
}
```

```
Public void setFilled (boolean filled) {
  this.filled = filled;
}
Public Date getDateCreated () {
   return dateCreated;
@Override
Public String toString(){
 return "Created on" + dateCreated + "\n Color:
          + color + " and filled: " + filled;
 }
}
class circle extends GeometricObject{
   private duble radius;
   public Circle (){
   this.redius = 1.0;
 }
 public Circle(double radius){
  this.radius = radius;
 }
 public circle (double radius, String color, boolean filled){
 super (color, filled);
 this.radius = radius;
 }pube double getRadius (){
 }    return radius;
```

```
public void setRadius (double radius) {
    this.radius = radius ;
}
public double getArea () {
    return Math.PI * radius * radius;
}
public double getPerimeter () {
    return 2* Math.PI * radius;
}
public double getDiameter() {
    return 2* radius;
}
public void printCircle(){
System.out.println("circle radius: "+radius);
}
}
class Rectangle extends GeometricObject {
    private double width;
    private double height;
    public Rectangle () {
    this.width = 1.0;
    this.height = 1.0;
}
```

```
public double getwidth () {
  return width;
}
public void setwidth (double width){
  this.width = width;
}public double getheight () {
  return height;
}public void setheight (double height){
   this.height = height;
}
public double getArea() {
  return width * height;
}public double getPerimeter() {
  return 2* (width + height);
}
}
public class TestGeometricObjects {
  public static void main (String[] args){
  Circle circle = new Circle (5.0, "Red", true);
  System.out.println ("Circle Details:");
  System.out.println ("Redius : "+ circle.getRadius()
```

```
System.out.print In ("Area : "+circle.get Area());
System.out.printIn ("Percimeter : "+ circle.get Perimeter());
System.out.printIn (circle.toString ());

System.out.printI();
Rectangle rectangle = new Rectangle (4.0, 7.0, "Blue", false);
System.out.printIn ("Rectangle Details:");
System.out.print In ("width : "+ rectangle.getwidth ());
System.out.printIn ("Height : "+ rectangle.getHeight ());
System.out.printIn ("Area : " + rectangle.get Area());
System.out.printIn ("perimeter : "+ rectangle.getPerimeter());
System.out.printIn (rectangle.toString());
}
}
```

Answer no:14

```
import java.math.BigInteger;
public class FactorialBigInteger {
    public static BigInteger factorial(int n){
        BigInteger result = BigInteger.ONE;
        for (int i=2; i<=n; i++) {
            result = result.multiply(BigInteger.valueOf(i));
        }
        result return result;
    }
public static void main(String[] args){
    int number = 50;
    System.out.println(number+"! ="+ factorial(number));
    }
}
```

IT23051

Answere No: 15

Feature	Abstract class	Interface
Purpose	Provides a partial implementation of class that other classes can extend	Provide a contract that class must adhere to, without defining implementation
method Types	Can have both abstract and non-abstract methods	Can have only abstract methods.
Fields (variables)	Can have instance variables.	Can have only public static final constants
constructors	Can have constructors	Can't have constructor
multiple inheritance	Support single inheritance	Supports multiple inheritance
Access modifier	Can have private, protected and public methods	All methods are public by default.

when to use an Abstract class vs an Interface

Scenario	use Abstract class	use Interface
Need to share common behavior	If some methods should have a default implementation share among multiple subclasses	If all methods are abstract.

IT23051

Scenario	Use Abstract class.	Use interface
Multiple inheritance required	Not possible (Java allows single inheritance)	Preffered, as a class can implement multiple ~~inheritance~~ interfaces
state or instance variable required	use abstract class, since it allows instance variables	Not possible, as interface only allow public static final constants.
Future proofing	Allows adding new methods without breaking existing subclasses	Before Java 8 adding new methods would break all implementing classes

A class can implement multiple interface in Java. This allows a class to inherit behavior from multiple sources, overcoming the limitation of single inheritance in abstract classes.

#Conflicting Method signatures in Interfaces

⇒ If two interfaces have methods with the same signature but different default implementations the class must override the conflicting method explicitly.

Answers to Qn : 16

polymorphism in Java : Polymorphism is one of the core principles of object oriented programming (OOP) in Java. It allows objects to be treated as instances of their parent class while retaining their specific behavior.

Dynamic Method Dispatch : Dynamic method Dispatch (also called runtime polymorphism) refers to Java's mechanism where the method that gets executed is determined at runtime based on the object's actual type rather than the reference type

Example of polymorphism Using Inheritance and method overriding :

```
class Animal {
  void makeSound (){
  System.out.println("Animal makes a sound");
  }
}
class Dog extends Animal {
```

```
    @Override
    void makeSound( ){
        System.out.println("Dog barks");
    }
}
class Cat extends Animal {
    @Override
    void make sound ( ) {
    System.out.println("cat meows");
}
}
public class PolymorphismExample {
    public static void main (String[] args) {
    Animal MyAnimal;
    myAnimal = new Dog();
    myAnimal.makeSound();
    myAnimal = new Cat();
    myAnimal.makeSound();
}
```

IT23051

Performance Impact of polymorphism

1. Methods called Are Resolved at Runtime : Instead of calling a method directly, Java uses the virtual method Table (VMT) to determine to correct method.
2. Cache Efficiency May Decrease : If different subclass objects frequently replace each other, CPU branch prediction may suffer
3. memory Overhead : Each object stores metadata, which can slightly increase memory usage

Trade-offs : Polymorphism vs specific method calls

Aspect	Using polymorphism	Using specific method calls
Flexibility	Highly flexible, allows easy code extension	Less flexible ; must modify code for new cases
Code maintainability	Easier to maintain and modify	Harder to modify requires explicit type checking
Performance	Slightly slower due to run time method resolution	Faster because method resolution happens at compile time
memory usage	Slightly higher due to VMT usage	Lower as method calls are direct
Extensibility	New classes can be added without modifying existing code	Requires modifying existing logic to accomodate new classes

IT23051

Ans to Qn : 17

Difference between ArrayList and LinkedList in Java

Feature	ArrayList	LinkedList
1. Underlying Structure	- Dynamic array (Resizable array)	- Doubly Linked List
2. Access Time (get index)	- O(1) Direct access via index	- O(n) Traverses nodes sequentially
3. Insertion at End (add(E, e))	- O(1) Amortized, when capacity allows	- O(1) Direct addition
4. Insertion at Beggning	- O(n)	- O(1)
5. Insertion at middle	- O(n)	- O(n)
6. Deletion at End	- O(1)	- O(n)
7. Detetion at Beggning	- O(n)	- O(1)
8. Detetion at middle	- O(n)	- O(n)
9. Memory Usage	- Less	- More
10. Itercation Performance	- Faster	- Slower

When to use ArrayList Vs LinkedList

Use ArrayList when:

1. You need fast random access
2. Insertions and deletions mostly happen at the end
3. Memory efficiency is a concern
4. Iteration performance is important due to cache loca

use Linked list when:

1. You need frequent insertions/deletions at the beginning or middle.
2. You don't nieed frequent random access
3. You have large datasets with frequent insertions/ deletions where shifting in Arraylist is costly.

Performance consideration of Leargе datasets:

1. Arraylist is better for read-heavy operations where you mostly access elements.
2. Linked list is better for frequent insertions and deletion especilly at the bigg begining or middle
3. Memory overhead in Linked list can be significant due to extra storage for pointers.

IT23051

Answer : 18

```
import java.util.Random;
Public class customRandomGenerator {
    private static final int[] predifiened Array
                        = {3, 7, 11, 17, 23, 31, 41, 53, 61, 73};
    public static int[] myRand (int n, int maxValue){
    int[] randomNumbers = new int[n];
    long currentTime = System.currentTimeMillis();
    for (int i=0; i<n; i++ ){
        int index = i % predefined Array.length;
        randomNumbers[i] = (int)((currentTime * predefined
                            Array[index]) % maxValue);
    }
    return randomNumbers;
}
public static int myRand (int maxValue){
    long currentTime = System.currentTimeMillis();
    int index = (int)(currentTime % predefinedArray.length
    return (int)((currentTime * predefinedArray[index]
                        % maxValue
}
```

```
public static double myRand() {
    long currentTime = System.currentTimeMillis();
    int index = (int) (currentTime % predefinedArray.length);
    return ((currentTime * predefinedArray[index]) % 10000 / 100);
}

public static void main(String[] args) {
    int n = 5;
    int maxValue = 100;

    int[] randomNumbers = myRand(n, maxValue);
    System.out.println("Generated Random Numbers:");
    for (int num : randomNumbers) {
        System.out.println(num);
    }
    System.out.println("\nSingle Random Number: " + myRand(maxValue));
    System.out.println("\nRandom Floating-Point Number: " + myRand());
}
}
```

IT23051

Answering Qn: 19

Multithreading in Java allows concurrent execution of two or more threads, enabling efficient utilization of CPU resources. Java provides built in support for multithreading through the Thread class and the Runable interface.

Thread class vs Runable Interface

1. Thread class:

⇒ The thread class is a concrete class that extends Java.lang.Thread.

⇒ To create a thread, you can extend the Thread class and override its run() method

2. Runnable ~~class~~ Interface:

⇒ The runnable interface is a functional interface that defines a single run() method.

⇒ To create a thread, you can implement the Runnable interface and pass an instance of the class to a thread object.

IT23051

Potential Issues with Multiple Threads

1. Race conditions: Occurs when multiple threads access shared resources concurrently, leading to inconsistent or unexpected results.
2. Deadlocks: Occurs when two or more threads are blocked forever, waiting for each other to release locks
3. Thread Inheritance: Occurs when multiple threads access to modify shared data simultaneously leading to inconsistent states
4. Memory consistency Errors: Occurs when different threads have inconsistent views of the same data.

Synchronized keyword.

The synchronized keyword in Java is used to control access to critical sections of code, ensuring that only one thread can execute the synchronized block or method at a time. This helps in achieving thread safely

IT23051

```
class Resource {
    Synchronized void method1 (Resource another){
    System.out.println("method 1");
    another.method2();
    }
    Synchronized void method2 () {
    System.out.println ("method 2");
    }
}
public class Deadlock {
public static void main (String [] args){
final Resource resource1 = new Resource();
final Resource resource2 = new Resource();
Thread t1 = new Thread (() -> resource1.method1 (resource
Thread t2 = new Thread (() -> resource2.method1 (resour
t1.start();
t2.start();
}
}
```

Avoiding Deadlock

1. Lock ordering : Always acquire locks in a consistent order.
2. Lock time out : Use tryLock() with a timeout to avoid waiting indefinitely
3. Avoid Nested Locks : Try to avoid locking multiple resources if possible.
4. Use Higher-level concurrency Utilities : Prefer using java.util.concurrent utilities like ReentrantLock, semaphore etc

IT23051

Answering Qn: 20

Exception handling in Java is a mechanism to handle runtime errors, ensuring that the normal flow of the application is maintained. Java provides a robust and object-oriented way to handle exceptions using try, catch, finally, throw and throws keywork.

checked vs. Unchecked Exceptions

1. checked Exceptions:
   - → checked at compile time
   - → must be either caught or declared using throw
   - → Example: IOException, SQLException.
2. Unchecked Exceptions
   - → Not checked at compile time
   - → occer at runtime
   - → Example: NullPointerException, AmithmeticException

Role of throw and throws

1. throw: The throw keyword is used to explicitly throw an exception within a method or block code.

throws: The throws keyword is used in method signature to declare that the method might throw one or more exceptions

Ensuring Exception Handling Does not Introduct Resource Lock

Resource leaks can occur if resources like file handles, database connections or network sockets are not properly closed after use, epecially when an exception is thrown. To prevent resource leaks, you can use try-with-resources statement, which ensures that each resource is closed at the end of the statement.