Lab-1

```
import java.io.file;
import java.io.PrintWriter;
import java.util.Scanner;

Public class SumOfNaturalNumber{
Public static void main(String[]args){
try{
file inputfile = new file("input.txt");
Scanner scanner = new Scanner(input.file);
Scanner.useDelimiter(",")
int maxNumber = Integer.MIN_VALUE;
while (Scanner.hasNextInt()){
int num = Scanner.nextInt();
if (num > maxNumber){
maxNumber = num;
}
}
Scanner.close();
int sum = (maxNumber * (maxNumber+1))/2;
PrintWriter writer = new PrintWriter("Output.txt");
writer.println(sum);
writer.close();
System.out.println("Result written to output.txt");
} catch (Exception e){
}
}
```

Lab-2 Differences Between Static and final fields and Method in Java.

Static

Definition: Belongs to the class rather than instances.

Memory Allocation: Stored in the class memory

Modification: Can be changed

Method: Can be called using the class name

fields: Shared across all object of the class.

final

Definition: Used to declare constant or prevent method overriding.

Memory Allocation: Stored ~~in the class spe~~ separately for each instance.

Modification: Cannot be modified after initialization

Method: Prevents method overriding when used with methods

fields: Cannot be reassigned once initialized.

(P.T.O)

Lab-03 Here is a Java Program to find all Armstrong numbers within a given range.

```
import Java.util.Scanner;
Public class factorionNumbers{
Private static final int[] FACTORIALS = new int[10]
static{
FACTORIALS [0] = 1;
for (int i=1; i<10; i++){
FACTORIALS [i] = i* FACTORIALS [i-1];
}
}
Private static boolean isFactorion (int num){
int sum =0 , temp =num;
while (temp>0) {
int digit =temp %10;
sum = + FACTORIALS[digit];
temp /=10;
}
return sum ==num;
}
public static void main (string I ]args){
Scanner scanner =new Scanner (system.in);
System.out.print ("Enter the lower bound of range:");
int lower = scanner.nextInt ();
System.out.Print ("Enter the next of the range");
int upper = scanner.nextInt ();
```

14. Difference Among Class, Local and Instance varibles.

Class variable:-

Definition: Shared across all instances of a class.
Scope: Class level (shared by all instances)
Lifetime: Exists as long as the class is in memory.
Access Method: Accessed using className.variable_name

Instance Variable:

Definition: Unique to each instance of a class.
Scope: Instance level (specific to an object)
Lifetime: Exists as long as the instance exists.
Access Method: Accessed using self.variable_name.

Local Varibale:

Definition: Defined inside a method or function.
Scope: Limited to the function where its declared.
Lifetime: Exists only during function execution
Access Method: Directly used within the function.

```
Class Example {
int num;
Example (int num) {
this.sum = num;
void display() {
```

```
System.out.println("Fraction numbers in the range:");
for (int i = lower; i <= upper; i++){
if (isFactorion(i)) {
System.out.print(i + " ");
}
}
scanner.close();
}
}
```

lab-5 A Java program that defines a method to calculate the sum of all elements in a integer array and demonstrates in the main method.

```
Public class ArraySumCalculator {
Public static int calculateSum(int[] arr){
int sum = 0;
for (int num : arr) {
sum += num;
}
return sum;
}
Public static void main (String[] args) {
int[] numbers = {10, 20, 30, 40, 50};
int sum = calculateSum(numbers);
System.out.println("The sum of array elements is:" + sum);
}
}
```

(ATP)

Lab-06 Acces Moditions in Java are keywords that define the visibilty or accessibility of classes, metho and variables, Java provides four access moditions:

1. (Public) - Accesible from anywhere
2. (Private) - Accesible only within the same class
3. (Protected) - Accesible within the same package and subclasses.
4. (Default) - Accessible only within the same Package.

Types of variable in Java:

Java has three types of vanichle;

1. Local vanidbles: Declared inside a method Construcion or block.
2. Instance vanibles: Declared inside a class but outside methods, each object has its own copy
3. Class vaniables: Declared using static shared among all instances of a class.

```
Class Example {
int instancevar = 10;
static int staticvar = 20;
void show() {
int localvar = 30;
System.out.println("Local variable : " + localvar);
System.out.println("Instance variable : " + instancevar);
System.out.println("static variable : " + staticvar);
}
}
Public class Main {
Public static void main(String[] args) {
Example obj = New Example();
obj.show();
System.out.println("static variable (Access via class) "
                   + Example.staticvar);
}
}
```

Lab-7 A Java program to find the smallest positive root of a quadratic equation using the quadratic formula:

```
import java.util.Scanner;
Public class smallestPositiveRoot {
Public static void main (String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.Print ("Enter Coefficients a, b, c and c:");
double a = Scanner.nextDouble ();
double b = Scanner.nextDouble ();
double c = scanner.nextDouble ();
double discriminant = b*b - 4*a*c;
if (discriminant < 0) {
System.out.Println ("No real roots;");
} else {
double sqrtDiscriminant = Math.sqrt (discriminant);
double root1 = (-b + sqrtDiscriminant) / (2*a);
double root2 = (-b + sqrtDiscriminant) / (2*a);
double smallestPositiveRoot = Double.MAX_VALUE;
if (root1 > 0) smallestPositiveRoot = root1;
if (root2 > 0 && root2 < smallestPositiveRoot) smallest
                                    positiveroot = root2;
```

(P.T.O)

```
if (smallestPositiveRoot == Double.MAX_VALUE) {
System.out.println("No positive roots.");
} else {
System.out.println("The smallest positive root is:
                    " + smallestPositiveRoot);
}
}
scanner.close();
}
```

Lab 2 This program takes a string input and checks each character to determine if it is a letter, digit, or white space.

```
import java.util.scanner;
Public class characterclassifier {
Public static void classifyCharacters(String input)
{
int letters = 0, digits = 0, whitespaces = 0;
for (char ch; input.toCharArray()){
if (character.isLetter(ch)){
letters++;
} else if (character.isDigit(ch)) {
digits++;
} else if (character.isWhitespace(ch)){
whitespaces++; } (pt.)
```

```
}
System.out.println("Letters:" + letters);
System.out.println("Digits:" + digits);
System.out.println("Whitespaces:" + whitespaces);
}
Public static void main (String[] args) {
Scanner scanner = new scanner (system.in);
System.out.Print ("Enter a string:");
String input = scanner.nextLine();
classify character (input);
Scanner.close ();
}
}
```

Again in Java, an array is passed to a function by specifying the array type as a parameter

```
Public class ArrayPassingExample {
Public static void PrintArray (int[] arr) {
System.out.Print ("Array elements:");
for (int num; arr) {
System.out.Print (num + " ");
}System.out.Print ();
}
```

```
Public static void main (String[] args) {
int[] numbers = {10, 20, 30, 40, 50};
PrintArray(numbers);
}
}
```

Lab-8 Method overriding occurs in Java when a subclass provides a specific implementation of a method that is already defined in its superclass. The overridden method in the subclass must have

Method overriding : Subclass a new implementation of a superclass method with the same signature.

Dynamic Method Dispatch: Overridden methods are called based on the actual object type at runtime.

using (super): Calls the superclass version of an overridden method or constructor.

Access Modifiers Rule; Cannot make the overridden method more restrictive than the superclass.

Checked Exceptions: Subclass cannot throw broader exceptions than the overridden method in the subclass.

Lab-10

final Methods: A methods declared final cannot be overridden.

Static Method: Cannot be overridden they are hidden instead.

Constructors: Not inherited but can call the superclass constructors using super();

Feature	Static Members	Non-static members
Definition	Declared using static keyword.	Declared without static keyword
Memory Allocation	Stored in the class memory	Stored in the heap memory
Access	Can be accessed using the class name	Requires an instance of the class
Association	Belongs to the class rather than an object	Belongs to the individual objects
Usage	Used for common utilities like constants and helper methods	Used for instance specific behavior
Example	Static int count;	int age;

Lab-11 Class Abstraction;

Abstraction is the process of hiding implementation details and showing only the necessary features of an object. It helps reduce complexity and increase reusability.

```
abstract class vehicle {
abstract void start();
void stop() {
System.out.println("Vehicle is stopping...");
}
}
class Car extends vehicle {
@override
void start() {
System.out.println("Car is starting with a key...");
}
}
Public class AbstractionExample {
Public static void main(String[] args) {
Vehicle mycar = new Car();
Mycar.start();
Mycar.stop();
}
}
```

(pt:)

Encapsulation:

Encapsulation is the process of wrapping data (variables) and methods into a single unit (class). It restricts direct access to class data by making variables private. and provides getter & setter methods for controlled access.

```
Class BankAccount {
Private double balance;
Public void deposit (double amount) {
if (amount > 0) {
balance += amount;
System.out.println("Deposited: $" + amount);
} else {
System.out.println("Invalid deposit amount!");
}
Public double getBalance() {
return balance;
}
Public class EncapsulationExample {
Public static void main (String[] args) {
BankAccount account = new BankAccount();
```

(n.t.)

```
account.deposit(1000);
System.out.println("Current Balance: " + account.
                    getBalance());
}
}
```

Lab-12 (i) BaseClass - Provides a method to print result.

```
Class BaseClass {
void PrintResult(String result) {
System.out.println(result);
}
}
```

(ii) SumClass - computes the sum of the series 1+ 0.9+0.8+0.7+.....+0.1

```
class sumclass extends BaseClass {
double computeSum() {
double sum = 0;
for (double i=1.0; i>=0; i-=0.1){
sum+=i;
}return sum;
}
}
```

(P.T.O)

(iii) DivisonMultipleClass - finds the GCD and LCM of two numbers:

```
class DivisonMultipleClass extends BaseClass {
~~double computeSum(){~~
~~double sum=0;~~
~~for(double i=1.0; i>=0; i-~~
int gcd(int a, int b){
while(b!=0){
int temp=b;
b=a%b;
a=temp;
}return a;
}int lcm(int a, int b){
return (a*b)/gcd(a,b);
}
}
```

(iv) NumberConversionClass - Converts numbers between different base.

```
class NumberConversionClass extends BaseClass {
string toBinary(int num){
```

(P.T.O)

return Integer.toBinaryString(num);
}
String toHexadecimal(int num){
return Integer.toHexString(num).toUpperCase();
}
String toOctal(int num){
return Integer.toOctalString(num);
}
}

(v) CustomPrintClass - Implements a pr() method for formatted printing

class CustomPrintClass extends Baseclass {
void pr(String message){
System.out.println("[output]:" + message);
}
}

(vi) Mainclass - creats objects and calles methods from all classes

Public class Mainclass {
Public static void main(String[] args){
Sumclass sumobj = new Sumclass();

DivisorMultiple class gcdLcmobj = new DivisorMultipleClass();

NumberConvonsionclass numConvobj = new numberConversionClass();

CustomPrintClass Printobj = new CustomPrintClass();

// Compute sum of series

double sum = sumobj.computeSum();

subobj.PrintResult("Sum of series:" + sum);

// Compute GCD and LCM

int a = 36, b = 48;

int gcd = gcdLcmobj.gcd(a, b);

int lcm = gcdLcmobj.lcm(a, b);

gcdLcmobj.PrintResult("GCD of " + a + " and " + b + " is: " + gcd);

gcdLcmobj.PrintResult("Lcm of " + a + " and " + b + " is: " + lcm);

// Number Conversions

int number = 29;

Printobj.Pr("Binary of " + number + " is: " + numConvobj.toBinary(number));

Printobj.Pr("Hexadecimal of " + number + " is: " + numConvobj.toHexadecimal(number));

```
Printobj.pr1("Octal of" + number + "is:" + numberobj
                .toOctal(number));
}
}
```

14. The significance of BigIntegers in Java is that it allows us to handle arbitrarily large numbers which cannot be stored in primitive data types like int or long. Factorials grow very fast, making BigIntegers essential for calculations involving large numbers.

```
import java.math.BigInteger;
import java.util.Scanner;
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));
}
return result;
}
Public static void main (String[] args) {
Scanner scanner = new scanner (system.in);
```

(P.T.O)

```
System.out.Print("Enter a number:");
int number = scanner.nextInt();
Scanner.close();
System.out.Println(number+"! is ");
System.out.Println(Factorial(number));
}
}
```

15 Comparison of Abstraction: Abstract classes vs Interfaces in Java.

Abstract class	Interface
1. A class that cannot be instantiated but can have both abstract and concrete method.	1. A blueprint of a class that contains only abstract methods and default method.
2. Can have both abstract and non-abstract method	2. Methods were abstract only before Java 8. but now interface can have default and static methods.
3. Can have instance variables with any access modifier	3. Can only have public static final constant.
4. Can have constructors to initialize fields.	4. Cannot have constructors
5. Methods can be public, protected or Private.	5. A class can implement multiple interfaces
	6. All methods are public by default.

Multiple Interfaces in Java

```
interface A {
void methodA();
}interface B{
void methodB();
}class C implements A, B{
Public void methodA(){
System.out.Println("Method A implemented");
}Public void methodB(){
System.out.Println("Method B implemented");
}
}Public class main{
Public static void main (String[] args){
C obj = new C();
obj.methodA();
obj.methodB();
}
}
```

16 Polymorphism in Java:

Polymorphism in Java is the ability of an object to take multiple forms. It allows a subclass to provide a specific implementation of a method that is already defined in its superclass.

Dynamic Method Dispatch:

Dynamic method dispatch is the mechanism by which a method call to an overridden method is resolved at runtime, rather than compile-time. When a superclass reference points to a subclass object, Java determines which method to execute based on that actual object, not the reference type.

Polymorphism (Dynamic Binding)	Direct Method (Static Binding)
(1) High - Allows adding new behaviours easily (2) Slightly slower due to method lookup (3) High - can use a common interface (4) More readable	1. Low - Tightly coupled method 2. Faster as method resolution happens at compile-time 3. Low - Duplicates code across different classes 4. Less readable if too many specific method calls exist.

18 Both ArrayList and LinkedList are implementations of the List interface in Java. but they have different underlying data structures, affecting their performance in different operations.

ArrayList (Dynamic Array)	LinkedList (Doubly linked list)
1. Resizable array 2. O(1) - Direct access using index. 3. O(1) (Amortized) fast, unless resizing needed 4. O(n) - Requires shifting elements. 5. Less overhead (only store data) 6. Faster due to contiguous memory allocation 7. High (better CPU caching).	1. Doubly linked list. 2. O(n) - Sequential traversal required. 3. O(1) - Directly adding a node. 4. O(1) at beginning O(n) at middle - only pointer updates required 5. More overhead (stores data + Pointers) 6. Slower due to pointer traversal. 7. Low (nodes stored in scattered memory)

1. ArrayList is Cache-friendly - Data is stored contiguously, improving CPU caching and performance.
2. LinkedList uses Extra Memory - Each node stores extra pointers (next and prev), increasing memory overhead.
3. Garbage Collection Overhead in LinkedList - More objects and references slow and down. GC

performance.

4. Random Access in ArrayList is faster - get (index) in O(1) vs O(n) in LinkedList.

5. Insertion/Deletion performance - If frequent updates are required in a very large list, linked list might be preferable.

19 Multithreading in Java allows concurrent execution of two more parts of a program for maximum utilization of CPU. Each part of such a program is called a thread. and Java provides built in support for multithreading through the thread class and Runnable interface.

Thread class	Runnable Interface
1. Extends thread class, so it cannot extend any other class.	1. Implements Runnable interface allowing multiple inheritance through interfaces
2. Must override the run() method	2. run() method is defined in Runnable and passed to thread.
3. Less flexible due to single inheritance limitation	3. More flexible since it separates thread execution from object.
4. Suitable when creating a thread with additional functionalities.	4. Suitable when sharing a common resource among multiple threads

(at 9)

Using Runnable Interface.

```
Class my Runnable implements Runnable {
Public void run() {
for ( int i = 0 ; i < 5 ; i++) {
System.out.println (Thread.CurrentThread().getName()
                    + " : " + i);
}
}
Public class RunnableExample {
Public static void main (String[] args) {
Thread t1 = new Thread (new MyRunnable());
t1.start();
}
}
```

20 Exception handling in java ~~allows~~ is a powerfull mechanism that allows you to handle runtime errors, ensuring the normal flow of the application is maintained even when exceptional situations occurs. The basic structure of exception handling in java involves: try, catch, finally, throw, and throws blocks.

1. try: The code that might throw an exception is placed inside the try block

2. Catch: This Block follows the try block and is used to catch handle exceptions You can have multiple catch blocks to handle different types of exceptions.

3. finally: The finally block is optional and executes whether or not an exception is thrown. It is typically used to clear up resource like closing file streams, database connections etc.

4. Throw: The throw keyword is used in method declarations to indicate that a method can throw one or more exceptions This informs the caller of the method that they need to handle the exceptions or propagate it further.

```
public void readFile() throws IOException {
FileReader file = new FileReader("File.txt");
BufferReader fileInput = new BufferedReader(file);
throw new IOException("File not found");
```

21 @ Conflict Resolution

```
interface InterfaceA {
default void myMethod() {
System.out.println("My Method from InterfaceA");
}
}
interface InterfaceB {
default void MyMethod() {
System.out.println("My Method from InterfaceB");
}
}
abstract class AbstractClass {
public void mymethod() {
System.out.println("My method from abstractclass");
}
class ConcreteClass extends AbstractClass implements
  interface A, InterfaceB {
@Override
public void myMethod() {
System.out.println("My method from ConcreteClass");
}
}
```

* Static binding happens when the method is resolved at compile time based on the reference type. This is typical for private, final, and static methods where polymorphism does not apply.

```
class parent {
public void display() {
System.out.println("Parent display");
}
class child extends parent{
Public void display() {
System.out.println("child display");
}public static void statatic method(){
System.out.println("child static method");
}
}public class main{
public static void main(String[]args){
Parent p=new child();
p.display();
p.static Method();
}
}
```

* Static Binding vs Dynamic Binding

* Static Binding: This occurs at compile time. The method call is resolved based on the type of the reference variable, not the object it points to. static methods, private methods, and final methods are resolved using static binding because their target method is known at compile time. Essentially the compiler knows which method to call when the code is compiled.

* Dynamic Binding: This occurs at runtime. The method call is resolved based on the actual object type, not the exact method that was called in subclass are resolved dynamically because the exact method to invoke is determined when the program is running. This is an essential part of runtime polymorphism. where the method that gets executed depends on the object actual class, even if it is referred to by a reference of the superclass.

(cont.)

Example of Static and Dynamic Binding Producing Different Behaviour.

```
Class Animal {
void sound() {
System.out.println("Animal makes a sound");
}
static void static sound() {
System.out.println("Static sound in Animal");
}
class Dog extends Animal {
@ override
void sound() {
System.out.println("Dog barks");
}
Public class main {
Public static void main (String[] args) {
Animal myDog = new Dog();
myDog.sound();
MyDog.Static Sound();
}
}
```

(2to)

Ex: Advantage of Using the Executor Service Framework Over Manually Manging Threads.

1. Thread Pool Managment: With Executors you can mange a pool of worker threads instead of creating and manging individual threads manually. This helps in avoiding issues like thread exhaustion

2. Task Scheduling and Execution: Executor Service handles the scheduling of tasks for you. It ensures that tasks are executed currently, with automatic management of thread of pooling, which reduces the complexity of manging threads manually.

3. Better Resource Utilization: By using thread pools, you can limit the number of concurrent threads, preventing overloading of the system. This also helps avoid the situation where each task creats a new thread and the system becomes overwhelmed.

4. Graceful Shutdown: With ExecutorService, you can gracefully shut down the thread pool using shutdown() or shutdownNow() Methods, allowing you

5. Error Handling and future Support: The executorservice framework has built in support for handling errors and exceptions. It also support returning a future object that allows you to retrive the result of a task, track its progress, and handle exceptions.

6. Improved Code Readability: Using the Executorservice framework results in clean and more maintainable code, as it abstracts away the complexities of manual thread creation and mangement.

Key Issues to consider During Exception Handling:

1. Exception Type: Decide whether to use checked exception or unchecked. A RuntimeException is often used for issues that are typically caused by bugs (while checked exceptions are used for recoverable conditions.

Program to calculate the Area of a circle with setRadius Method that Throws an exception if the Radius is Negative.

```
class circle {
Private double radius;
Public void setRadius (double radius) throws
IllegalArgumentException {
if (radius < 0) {
throw new IllegalArgumentException ("Radius can not negative");
}
this.radius = radius;
}
public double calculateArea() {
return Math.PI * radius * radius;
}
}
Public class circleTest {
Public static void main (String[] args) {
Circle circle = new circle ();
try {
circle.setRadius (5.0);
System.out.println ("Area of the circle with radius 5.0: " + circle.Area());
System.out.println ("Error" + e.getMessage());
}
}
```

(10) Creating a Thread by Implementing the Runnable Interface:

```
class MyRunnable implements Runnable {
@override
Public void run() {
for(int i = 0; i<3; i++){
System.out.println("Runnable Thread is running:"+i);
try {
Thread.Sleep(1000);
} Catch (Interrupted Exception e) {
e.PrintStackTrace();
}
}
}
Public class RunnableThreadExample {
Public static void main (String[] args) {
MyRunnable myRunnable = new MyRunnable();
Thread thread = new Thread(myRunnable);
thread.start();
for(int i = 0; i<=3; i++){
System.out.println("Main Thread is running:"+i);
try { Thread.sleep(500);
} Catch (InterruptedException e) {
e.printStackTrace();
}
```

```
class MyThread extends Thread {
@Override
public void run() {
for (int i=0; i<5; i++) {
System.out.println("Thread Extending Thread class
                    is running: " + i);
try {
Thread.sleep(500);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
public class ThreadClassExample {
public static void main(String[] args) {
MyThread myThread = new MyThread();
myThread.start();
for (int i=0; i<5; i++) {
System.out.println("Main Thread is running: " + i);
try {
Thread.sleep(500);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
```

Q4) The complete Java program based on the given UML2 diagram:—

```
interface Edible {
String howToEat();
}
abstract class Animal {
abstract String sound();
}
class Tiger extends Animal
@override
String sound() {
return "Roar";
}
class Chicken extends Animal implements Edible{
@override
String sound(){
return "cluck";
}
@override
Public string howToEat(){
return "could be fried";
}
abstract class fruit implements Edible{}
class Apple extends fruit {
```

(P.T.O)

```
@override
Public string howToEat(){
return "Make apple cider";
}
class orange extends Fruit{
@override
Public string howToEat(){
return "Make orange juice";
}
Public class Main{
Edible[] fruits = {new Apple(), new Orange()};
for (Edible fruit : fruits){
System.out.println(fruit.howToEat());
}
}
```

(P.T.O)

(xxx) Java program that reads a series of numbers from a file, finds the highest value, computes the sum of all numbers, and writes the sum to another file using the Scanner and PrintWriter classes.

```
import java.io.*;
import java.util.Scanner;
Public class FileNumberProcessor {
Public static void main (String [] args) {
String inputFileName = "number.txt";
String outputFileName = "result.txt";
int MaxNumber = Integer.MIN_Value
int sum = 0;
try ( Scanner scanner = new Scanner(new File(input
while (Scanner.hasNextInt()) {          fileName)) {
int number = scanner.nextInt();
sum+= number;
if (number > maxNumber) {
MaxNumber = number;
}
} catch (FileNotFoundException e) {
```

System.out.println("Error: Input file not found");
return;
}
try (PrintWriter writer = new PrintWriter(new File(outputFileName))) {
writer.println("Secret number: " + secretNumber);
writer.println("Highest number: " + maxNumber);
} catch (FileNotFoundException e) {
System.out.println("Error: Unable to write");
}
System.out.println("Processing complete check " + outputFileName);
}
}

(200) Java Program to print the current date and time in specified using java.time Package.
(LocalDatetime, DateTimeFormatter).

```
import java.time.LocalDatetime;
import java.time.format.DateTimeFormatter;
Public class DatetimeformatterExample {
Public static void main (String[] args) {
LocalDatetime now = LocalDatetime.now();
DateTimeformatter formatter = DateTimeformatter
                    ("dd-MM-YYYY HH:MM:SS");
String formattedDatetime = now.format(formatter);
System.out.println("Current date and time:"
                    formattedDatetime);
}
```

format Pattern	output
"YYYY/MM/dd HH:mm"	2025/02/03 15:30
"E, MMM dd YYYY"	Thu, feb 13 2025
"hh:mm a"	03:30 PM

(P.T.O)

(20) Java class named counterClass that contains static variable instanceCount to track the number of objects created. If the count exceeds 50, it resets back to zero.

```
Public class counterClass {
Private static int instanceCount = 0;
Public counterClass() {
instanceCount++;
if (instance > 50) {
  instance = 0;
}
Public static int getInstanceCount() {
return instanceCount;
}
Public static void main (String [] args) {
for (int i = 1; i <= 55; i++) {
new counterClass();
System.out.println ("Object" + i + " created. Instance
                    Count: " + counterClass.getC());
}
}
}
```

(Ans.)

(2660) Java method in named findExtreme that.
* Accepts a string ("smallest or "largest") as the first argument):
* Accepts a variable number of integers (varargs)
* Return either the smallest or largest numb
based on the input.

```
Public class ExtremeFinder {
Public static int findExtreme (String type, int... numbers)
{if (numbers.length == 0) {
throw new IllegalArgument ("At least one
    number must be provided.");}
}
int extreme = number[0];
if (type.equalsIgnoreCase ("Smallest")) {
for (int num : number) {
if (num < extreme) {
 extreme = num;
}
}
} else if (type.equalIgnoreCase ("Largest")) {
for }
return extreme;
}
```

(Pt.9)

```
Public static void main (String[] args) {
int x = findExtreme ("Smallest", 5, 2, 9, 1);
int y = findExtreme ("largest", 8, 7, 10, 4);
System.out.println ("Smallest : " + x);
System.out.println ("largest : " + y);
}
}
```

(2020)

```
Public class StringComparison {
Public static void main (String[] args) {
String S1 = "This is ICT 2107 Java";
String S2 = new String ("This is ICT 2107 Java");
String S3 = "This is ICT 2107 Java";
System.out.println (S1.equals(S2));
System.out.println (S1 == S2);
System.out.println (S1 == S3);
}
}
```

1. S1.equals(S2) → true. The equals() method in Java compares the content of the string. Since S1 and S2 contain the same text, the result is true. (OUTPUT)

2. $S_1 == S_2 \rightarrow$ false

↳ The == operator compares object references, not content. S_1 is stored in the string pool and S_2 is explicitly created using new String(), so it is a new object in the heap memory not in the string pool.

3. $S_1 == S_3 \rightarrow$ true. Both S_1 and S_3 are string literals. Java interns string literals, since S_1 and S_3 refer to the same object in the string pool. the result is true.

(Xbox) It looks like are referring to five methods (Method(); methodC(), methodD() . methodE()

1. Approach:

Define two interfaces (Alpha and Beta) Each interface can define at most two method, so we divide methodA(), methodB() methodC() and MethodD() between these two interfaces.

(OT.)

```
interface Alpha{
void methodA();
void methodB();
}
interface Beta{
void methodC();
void methodD();
}
abstract class AbstractBase implements Alpha{
Public void methodA(){
System.out.Println("Method A executed");
}Public void methodB(){
System.out.Println("Method B executed");
}abstract void methodE();
}
class finalclass extends AbstractBase implements
                                        Beta{
Public void methodC(){
System.out.Println D(){
System.out.Println("Method D executed");
}Public void methodE(){
System.void methodE(){
```

```
System.out.Println("MethodE executed");
}
Public static void main(String[] args){
final class obj = new finalclass();
obj.methodA();
obj.methodB();
obj.methodC();
obj.methodD();
obj.methodE();
}
}
```

(xxv) To resolve the conflict, explicitly override the show() method in 2 and specify which interface's show() method to call using x.super or y.super

```
show()
interface x{
default void show(){
System.out.Println("x.show");
}
}
```

```
interface x {
default void show() {
System.out.println("x:s show");
}
}Public class z implements x, y {
@override
Public void show() {
x.super.show();
}
public static void main(String[] args) {
z obj = new z();
obj.show();
}
}
```

Here is a complete java program implementing a singleton pattern for the class singleton Example, This ensure that only one instance of the class exists at a time using the lazy initialization with double checked locking approach which is (P.t.o) thread safe.

~~Class Singleton Example~~

```
Public class CommadLineCalculation {
Public static void main (String [] args) {
try {
if (args.length!=2) {
throw new IllegalArgumentException ("Please
                Provide exactly two integer argumet
int num1 = Integer.ParseInt (args[0]);
int num2 = Integer.ParseInt (args[1]);
int sum = num1 + num2;
int difference = num1 - num2;
int product = num1 * num2;
int quotient = (num2 != 0) ? (num1 / num2) : 0;
System.out.println ("Sum" + sum);
System.out.println ("Difference:" + difference);
System.out.println ("Product:" + Product);
if (num2 != 0) {
System.out.println ("Quotient:" + quotient);
```

(P.t.o)

```
} else {
System.out.println("Quotient : undefined (cannot
                                divided by zero)");
}
} catch (NumberFormatException e) {
System.out.println("Invalid input ! Please
                    enter two valid integers.");
} catch (IllegalArgumentException e) {
System.out.println(e.getmessage());
} catch (Exception e) {
System.out.println("An unexpected error got
                    message());
}
}
```

```
catch (IllegalArugmentException e) {
System.out.println(e.getmessage());
catch (Exception e) {
System.out.println("An unexpected error" + e.get
                                        message());
```

Name : Arpita Chandra Roy

ID : IT 23031

Information and Communication Technology

Courses : ICT 2101