IT23055

1]

```
import java.io.*;
import java.util.*;
public class NumProc {
  public static void main(String[] args) {
    String inFile = "input.txt";
    String outFile = "Output.txt";
try {
  Scanner sc = new Scanner(new File(inFile));
 sc.useDelimiter(",\\s*");
List<Integer> nums = new ArrayList<>();
while (sc.hasNext()){
 if(sc.hasNextInt()){
   nums.add(sc.nextInt());
 }
else {
   sc.next();
}
```

```
}
sc.close();
PrintWriter pw = new PrintWriter(new File(outFile));
for (int num: nums){
 int sum = sumNatNums(num);
 pw.println(num + "," + sum);
 }
 pw.close();
System.out.println("Processing complete.");
} catch (IOException e){
 System.err.println("Error reading or writting
                    files:" + e.getMessage());
}
}
private static int sumNatNums(int n){
 return n*(n+1)/2;
}
}
```

IT 23055

~~Date : 04.12.2024~~

2] Lab Test 3:

// Class 1:

```
public class Factorion{
  private int number;
  public Factorion(){ }
  public Factorion(int number){
     this.number = number;
  ~~public int get~~
  }
  public void setNumber(int number){
     this.number = number;
  }
  public int getNumber(){
    return this.number;
  }
private int factorial(int n){
      int fact = 1;
     for(int i=1; i<=n; i++){
      fact *= i;
     }
```

IT23055

```
        return fact;
    }
public boolean isFactorion(){
    int sum = 0;
    int temp = number;
    while (temp>0){
    int digit = temp % 10;
    sum += factorial(digit);
    temp /= 10;
    }
    return sum == number;
}
}
```

IT23055

Class 2:

```
import java.util.Scanner;

public class Factorion_Finder{
  public static void main (String[] args){
    Scanner sc = new Scanner (System.in);
    System.out.print ("Enter the start of the range: ");
    int start = sc.nextInt();
    System.out.print ("Enter the end of the range: ");
    int end = sc.nextInt();
    System.out.println("Factorion number in the range ["
                        +start+", "+end+ "] :");
    for(int i=start; i<= end ; i++) {
      Factorion factorion = new Factorion ();
      factorion.setNumber (i);
      if (factorion.isFactorion()) {
        System.out.println(i);
      }
    }
    sc.close();
  }
}
```

IT23055

4a] Difference ~~between~~ among class, local and intance variable:

class variable	Intance Variable	Local variable
1. Shared among all intances of the class.	2. Vnique to each instance	1. Exists only within a function or method.
2. Available across all instances	2. Limited to a specifie object	Only within the method where defined.
3. Inside the class, but outside method	3. Inside methods prefined with ~~self~~ This.	defined inside method without ~~self~~ This
4. Accessed by using the class name or instance.	4. Accessed by using only instance.	4. Accessed by using only methods.
5. changes to the class variable affect all instances.	5. changes to an instance variable affect only that specific instance	5. Exists temporarily and gets deleted after execution

IT23055

The this keyword in java is useful for:

1. Referring to instance variable when there is a naiming conflict.
2. Calling another constructor within the same class
3. This can be used to explicitly call another method of the same class.
4. If a method needs an object of the current class this keyword can be used.
5. Passing the current instance as an argument to another method.
6. An access modifier is a keyword used in object oriented programming to defined the scope and visibility of classes, methods, and other members.

IT23055

key differences of public, private and protected modifiers:

Public:

1. The most accessible modifiers.
2. members can be accessed from anywhere including different package and modules.

Private:

1. The most restrictive
2. members can only be accessed within same class.

Protected:

1. Provides some level of restriction.
2. Accessible within the same class, same package and subclass.
3. Not Accessible outside the class unless it is inheritted.

IT23055

The three main types of variables are local, Instance, static variable.

Local variables declared inside a method and only accessible within the method. Examples

```
publie class Example{
    publie static void main (String[] args){
        int Var = 20;
```

```
public class Example {
    int void display () {
        int Var = 10
        System.out.println ("Local variable : "+Var);
    }
}
```

Instance variables declared in a class but outside a method.

```
publie class Example{
    int InstanceVar = 20;
    void display () {
```

IT23055

```
        system.out.println("Instance Variable: "+Instance
public static void main(String[] args){
    Example obj = new Example();
    obj.display();
}
}
```

Static variable declared inside a class using static keyword.

```
public class example{
    public static int statVar = 30;
    public static void main(String[] args){
    System.out.println("Static Variable: "+statVar);
}
}
```

X123055

1]

```
Import java.util.Scanner;
public class RootFinder{
  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter coefficients a, b, and c: ");
    int a = sc.nextInt();
    int b = sc.nextInt();
    int c = sc.nextInt();
    double discriminant = b*b - 4*a*c;
    if (discriminant < 0){
      System.out.println("No real roots.");
    } else {
      double root1 = (-b + Math.sqrt(discriminant)) / (2*a);
      double root2 = (-b - Math.sqrt(discriminant)) / (2*a);
if (root1 > 0 && root2 > 0){
  System.out.println("The smallest positive root is: " +
                     Math.min(root1, root2));
```

IT23055

```
} else if (root1>0){
elese
  System.out.println("The smallest root is: "+root1);
}
else if (root2>0){
  System.out.println("The smallest root is:" + root2);}
else {
  System.out.println("No positive real roots.");
}
}
  sc.close();
}
}
```

IT23055

10] Differences between static and non static members

static members	Non-static members
1. Associated with the class rather than an instances	1. Associated with the intance of the class
2. Memory allocated when the class is loaded	2. Allocated separately for each object.
3. Accessed using the class name. (classname.member)	3. Accessed using an instance (objectName.member)
4. changes affect all instances of the class	4. changes affect only the specific object.

```
class Example {
  static int a = 0;
  int b = 0;
  static void staticmethod (){
    System.out.println (" This is a static method.");
}
```

IT23055

```
void nonstatic method(){
  System.out.println("This is a non-static
                      method.");

Example(){
    a++;
    b++;
  }
}

public class Test{
  public static void main(String[] args){
  Example obj = new Example();

System.out.println("Value : "+Example.a);

System.out.println("Value of non-static: "+obj.b);

Example.staticMethod();

obj.nonstatiemethod();

  }
}
```

I123055

```
public class Main {
  public static void main (String [] args) {
  String str = "Radar"; reversestr = " ";
  int strlength = str.length();
  for(int i = (strLength - 1); i >= 0; --i) {
    reversestr = reverseStr + str.charAt(i);
  }
  if (str.toLowerCase().equals (reversestr.toLowerCase())){
  System.out.println(Str+" is a Palindrome.");
  }
  else {
    system.out.println (str +" is not a palindrome String.")
  }
}
}
```

T 123055

```
public class main {
 public static void main(String [] args) {
 int num = 3553; reversedNum = 0, remainder;
 int orginalNum = num;
 while(num != 0) {
 remainder = num % 10;
 reversedNum = reverseNum * 10 + remainder;
 num /= 10;
 }
if (OriginalNum == reversedNum) {
 System.out.println (originalNum + "is palindrome.");
else {
 System.out.println (originalNum + " is not
                     palindrome.");
```

IT23055

11. Abstraction is the process of hiding the implementation details and showing only the essential feature of an object.

Encapsulation is the concept of restricting access to the internal state of an object and exposing only what is necessary.

Example of Abstraction and Encapsulation:

```
abstract class Animal {
   abstract void sound();
   void sleep(){
    System.out.println("Animal is sleeping.");
  }
}
Class dog extends Animal{
@Override
 void sound(){
```

IT23055

```
    System.out.println("Dog Barks");
    }
}

public class Main{
    public static void main(String[] args){
        Dog dog = new Dog();
        dog.sound();
        dog.sleep();
    }
}

class Employee{
    private String name;
    public String getName(){
        return name;
    }
public void setName(String name){
    this.name = name;
    }
}
```

IT23055

Differences between abstract class and interface:

Abstract	Interface
1. Can have both abstract and concentrate method.	1. Interface only have abstract method. can have default and static method.
2. Can have instance variables	2. Only public, static and final constants.
3. Can have constructors	3. cannot have constructors.
4. Supports public, private, protected default.	4. Methods are public by defualt
5. Supports single inheritance	5. Supports multiple inheritance

14. Significance of BigInteger in Java:

In java, the BigInteger class is used to handling arbitrarily large integers that can not be stored primitive data types like int or long. The maximum value of int is $2^{31}-1$ and for long it is $2^{63}-1$

however, factorial calculations grow exponentially and for numbers beyond 20, long is insufficient. BigInteger allows performing arithematic operations on large numbers without overflow.

Java program to return factorial:

```
import java.math.BigInteger;
import java.util.Scanner;
public class Factorial {
 public static BigInteger calFactorial(int n){
  BigInteger factorial = BigInteger.ONE;
for(int i=2; i<=n; i++){
 factorial = factorial.multiply(BigInteger.valueOf(i));
}
return factorial;
public static void main(String[] args){
Scanner scanner = new Scanner(System.in);
```

IT23055

```
System.out.println("Enter a number:");
int num = scanner.nextInt();
scanner.close();
System.out.println("Factorial of " + num + " is:");
System.out.println(calFactorial(num));
}
}
```

15. Abstraction achieved through abstract class:

```
abstract class Animal{
    abstract void sound();
    void sleep(){
        System.out.println("Dog barks.");
    }
}
public class
```

IT23055

15] Abstraction using Interface:

```
Interface Animal {
void sound();
}
Class dog implements Animal {
@Override
 public void sound(){
  System.out.println("Dog Barks");
 }
}
public class main {
 public static void main(String[] args){
  Dog dog = new dog();
  dog.sound();
 }
}
```

IT23055

We should ~~use~~ prefer an abstract class over an interface when we want to share common code and functionality among closely related classes that have an :relationship. Also when we need instance variables. Also when we need ~~abst~~ constructor.

yes, in java, a class can implement multiple interface.

In java, a class can implement multiple interface allowing it to inherit behaviors from multiple sources. This is a powerful feature that pomotes flexibility, modularity and reusability. * It ~~helps~~ is useful when dealing with conflicting method signatures. When a class implements multiple interface that contain methods with the same signature it must resolve the comflict explictly by providin

IT23055

its own implementation. If two interface declare the same method signature, the implementing class must override it to avoid ambiguity. If multiple interface have default methods with the same signature, the implementing class must either override the method or explicity call a specific interface's method. Unlike multiple inheritance in classes, Java inforces explicit conflict ~~resout~~ resolution, ensuring no ambiguity in method ~~resout~~ resolution.

IT23055

16] Polymorphism is a fundamental concept ~~of~~ in object oriented programming that allows objects of different classes to be treated as objects of a common superclass.

Dynamic method Dispatch (also known as run-time polymorphism is a mechanism in object oriented programming where the call to an overridden method. This is directly related to polymorphism because it allows a subclass to provide a specific implementation of a method that is already defined in its superclass. Poly-morphism enables an object to take multiple forms, dynamic method disptach is process that allows overridden methods to be invoked based on the actual object type at runtime.

IT23055

Polymorphism using method Overriding:

```
Class Animal {
    void sound() {
    System.out.println("Animals makes sound.");
    }
}
Class doDog extends Animal {
@Override
    void sound(){
System.out.println("Dog barks.");
    }
}
Public class Main{
    public static void main(string[] args){
    Animal animal = new Animal();
    animal.sound();
    Dog dog = new Dog();
    dog.sound();
}
```

IT23055

Polymorphism using method Overriding:

```
Class Animal {
  void sound() {
  System.out.println("Animals makes sound.");
  }
}
Class doDog extends Animal {
@Override
  void sound(){
system.out.println("Dog barks.");
  }
}
Publie class Main{
    public static void main(string[] args){
    Animal animal = new Animal();
    animal.sound();
    Dog dog = new Dog();
    dog.sound();
}
```

I123055

Trade-offs Polymorphism Vs Specific Method calls:

Using polymorphism	Direct Method calls
1. High code reusability	1. Low code reusability.
2. Flexibility high	2. Flexibility low
3. Performance slower	3. Performance faster.
4. Readability and Maintainablity better.	4. Worse.
5. compile time optimization Limited (can't inline easily	5. ~~compile~~ optimized (Inlining and better coaching.

Polymorphism using inheritance:

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

112 3055

```
class Dog extends Animal{
    void bark(){
    System.out.println("Dog is barking!");
}
}.

public class main{
    public static void main(String[] args){
    Animal animal = new Animal();
      animal.sound();
    Dog dog = new Dog();
     dog.sound();
     dog.bark();
```

T12305B

18. Difference between Array list and Linked List ?

Feature	ArrayList	LinkedList
Underlying data structure	Dynamic Array	Doubly linked List.
Access time	O(1) (Direct indexing)	O(n) (Sequential traversal)
Insertion at End	Amortized O(1) (May require resizing)	O(1) (Direct pointer update)
Insertion at begining	O(n).	O(1)
Performance of iteration	Fast	Slow
Best for	Frequent look-ups. Fewer insertion, or deletions	Frequent insertion and deletion at arbitrary positions.

IT23055

Prefer Array list when

1. We need fast random access.
2. Performs more searches and reads than insertion and deletion.
3. Memory usage is a concern as Array list has lower overhead.

Prefer linked list when

1. Frequent insertion and deletion occurs at the beginning or middle of the list.
2. have large datasets with unpredictable sizes and want to avoid resizing overhead.

An underlying data structure refers to the fundamental organization or format used to store and manage data within a specific system.

IT23055

#18 ~~Genarate~~ Generate random numbers.

```
import java.util.Arrays;
public class customRandomGenaratorr {
    private static final int[] predefineArray = {3,5,7,11,13,
                                                 17,19};
    private static final int maxVal = 100;
    public static int[] my_Rand (int n){
        int[] randomNumbers = new int[n];
        long currentTime = System.currentTimeMillis();
        for (int i = 0; i < n; i++) {
            int index = i % predefinedArray.length;
            randomNumbers[i] = (int) ((currentTime + predefenArra
                                       [index]) % maxVal);
        }
        return randomNumbers;
    public static void main(String[] args){
        int n = 5;
        int[] randomValues = myRand(n);
        System.out.println("Grenarated Randon Numbers : "+Arrays.
                           toString(randomValues));
    }
}
```

IT23055

(19) Multithreading in Java is a feature that allows concurrent execution of two or more parts of a program to maximize CPU utilization.

Difference between thread class and Runnable Interface:

Feture	Thread class	Interface
Inheritance	Extends Thread class (Single inheritance restriction	Implements Runnable interface.
Execution	Defines its own run() method and starts a new thread using start()	Implements run() and must be executed within a Thread instance
Flexibility	Less flexible as subclassing restricts further inheritance	More flexible as it promotes separation of concern

IT23055

Potential Issues in Multithreading

When dealing with multiple threads, several issues can arise:

1. Race Conditions: When multiple threads access shared resources simultaneously without proper synchronization, leading to inconsistent results

2. Deadlocks: When two or more threads are waiting ~~inden~~indefinitely for each others locks.

3. Starvation: When a thread is perpetually denied access to resources due to high-priority threads always getting execution.

The synchronized keyword in java ensures that only one thread can execute a block of code at a time, preventing race conditions. When a thread enters a synchronized method or block it locks the object.

IT23055

Example of Deadlock:

```
Class DeadlockExample{
  private final object lock1 = new object();
  private final object lock2 = new object();
Void method1(){
  synchronized (lock1){
  System.out.println("Thread 1: Holding lock1...");
  try{Thread.sleep(100); } catch(InterruptedException e){}
  synchronized (lock2){
  System.out.println("Thread 1: Holding lock1 and lock2.");
  }
}
}
void method2(){
  Synchronized (lock2){
   System.out.println("Thread2: Holding lock2.");
try{Thread.sleep(100);} catch(Interrupted
                              Exception e){}
synchronized (lock1){
```

IT23055

```
        System.out.println("Thread 2: Holding lock2 and lock1...");
    }
}
}
public static void main(String[] args){
    DeadlockExample obj = new DeadlockExample();
    Thread t1 = new Thread(obj::method1);
    Thread t2 = new Thread(obj::method2);
    t1.start();
    t2=start;
    }
}
```

How to avoid deadlocks:

1. Lock ordering: Always acquire locks in the same order across different threads.

2. Try-Lock Mechanism - Use tryLock() from java.util.concurrent.lock.Lock to avoid indefinite waiting.

IT23055

20] Exception handling in java is a mechanism that allows developers to handle runtime errors to maintain the normal flow of the program.

Difference between checked and unchecked Exceptions:

checked Exceptions	Unchecked Exceptions:
1. These are exceptions that the compiler enforces handling at compile time.	1. These are exceptions that the compiler does require the programmer to handle explicitly.
2. They must be either caught using a try catch block or declared using the throws keyword	2. Optional. Can be handle or required.
3. Exception but not Run time Exception.	3. Runtime exception.
4. checked at compile Time	4. checke at runtime.

IT23055

Creating and Throwing Custom Exceptions:

```
class InvalidAgeException {
 public InvalidAgeException(String message){
      super message);
    }
}
public void valideAge(int age) throws InvalidAge-
 throws InvaidAgeException {
  if(age<18){
   throw new InvalidAgeException("Age must be 18
                                  or above");
  }
}
```

throw vs throws

throw: Used inside a method to explicitly throw an exception

throws: Declare exceptions that a method might throw, allowing the caller to handle them.

I123055

21] Yes, an abstract class can implement an interface.

Method Resolution: Conflicting Implementations:
When both an abstract class and an interface provide a method implementation in the abstract class and a default method in the interface.

1, Class wins over interface → If a method exists in both an abstract class and an interface, the method from the abstract

2. Concrete class can override → If a class extends the abstract class and implements the interface, It can override the method from the to provide its own version.

22] Difference between HashMap, Tree Map, Linked HashMap?

Feature	HashMap	Tree Map	LinkedHashMap
Ordering	No ordering	Sorted in natural order.	Insersion order
Internal Data Structure	Hash table	Red black tree	Doubly Linked list + Hash table
Allow keys (null)	Yes	No	Yes
Thread-Safety	No thread-Safe	Not thread safe	Not thread Safe
Best Use case	Fast lookups and insersions	Sorted keys range queries	Maintain insersion order

Time complexity

~~insertion~~	HashMap	Tree Map	LinkedHashMap
insersion	O(1)	O(logn)	O(1)
Lookup	O(1)	O(logn)	O(1)
Deletion	O(1)	O(logn)	O(1)

IT23055

Key difference in ordering:

- Hashmap: Does not guarantee any order
- Tree Map: Maintains element in sorted order
- LinkedHashMap: Maintains insertion order.

When to use Each

. Fast key based lookups with no ordering require → Hashmap

- Use treemap when sorted keys are required.
- Use LinkedHashMap when maintaining insertion order.

IT23055

23] Static binding occurs at compile time, and dynamic binding occurs at runtime. This is closely related to polymorphism.

Static Binding:

1. The method to be called is determined at compile time.
2. Typically applies to private, static and final methods.
3. Used in method overloading.

Dynamic binding:

1. The method to be called is determined at runtime.
2. Happens with overridden methods in inheritance
3. Enables runtime polymorphism.

4

IT23055

Example:

```
class parent{
  void show(){
    System.out.println("Parent's show method
static void staticshow(){
    System.out.println("Parent's staticshow meth
  }
}
class child extends Parent{
  @override
   void show(){
      System.out.println("child's show method.");
   }
  static void staticshow(){
      system.out.println("child's static show meth
   }
}
```

IT23055

```
public class BindingExample {
    public static void main(String[] args){
    parent obj1 = new parent();
    parent obj2= new ~~parent~~ child();

obj1.show();
obj2.show();
obj1.staticshow();
obj2.staticshow();
    }
}
```

Performance and method resolution in ~~Inhe~~ Inheritance:

static ~~bu~~ binding: Improves performance since the method is resolved at compile time.

Dynamic binding enables runtime polymorphism which enhances flexibility but incurse a slight performance.

2] Advantages of using executor service over manually managing Threads:

1. Thread pooling: Reduces overhead by reducing threads.
2. Resource mangement: Limits active threads to prevent exhaustion.
3. Task scheduling: Supports delayed and periodic execution.
4. Graceful shutdown: Provides shutdown() and shutdownNow() for proper termination.
5. Concurrency Abstraction: Simplifies thread management, letting developers, focus on tasks.

IT23055

Difference between submit() and execute()

Feature	a:submit()	execute()
Return Type	Future<?> (for runnable.	void
Exception handling	Throws exception ~~directly~~ inside future	captures exceptions ~~inside future~~ directly
Task type	~~Only~~ accepts runnable and callable	Accepts only runnable
Result Handling	A Allows retrieving ~~task~~ results with Future.get()	Not return value

Benefits to use callable over Runnable:

1. Callable<V> allows task to return results, whereas runnable does not.

2. Callable can throw checked exceptions, making

error handling.

3. When submitted using submit(), callable returns a Future<V> that can be used to retrive the result or check the task status.

26] In java there are ~~to~~ two primary ways to create a thread:

1. By implementing the Runnable Interface
2. By Extending the thread class.

Example:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is Running.");
    }
```

IT23055

```
public static void main (String [] args){
    MyRunnable myrunnable = new MyRunnable();
    Thread thread = new Thread (myRunnable);
    thread.start();
}
}
```

Creating a thread .Extending the thread class :

```
class MyThread extends Thread {
    public void run(){
    System.out.println("Thread is Running.");
public static void main (String [] args){
    MyThread mythread = new MyThread ();
    mythread.start();
}
}
```

IT23055

27] Here The given UML diagram represents an interface and two main class hierarchies: Animal and Fruit.

Here's the Java implementation:

```
interface Edible {
  String howToEat();
}
abstract class Animal {
  abstract String sound();

  class Tiger extends Animal {
  @Override
String sound(){
  return "Cluck cluck";

@Override
publie string howtoEat(){
return "Could be fried.";
```

IT23055

```
    }
}
abstract class Fruit impliments Edible{}
class Apple extends Fruit {
 @override
 publica String howToEat(){
  return "Make an Apple Pie.";
class orange extends Fruit{
@override
public string howToEat(){
  return "Make orange juice.";
  }
}
public class TestEdible {
 public static void main(String[] args){
  Animal tiger = new tiger();
```

IT23055

```
System.out.println (" Tiger Sound: " + tiger.sound());

Animal chicken = new Chicken();
System.out.println("chicken Sound: +chicken.sound());
System.out.println("chicken How to eat: " +((Chicken)
chicken).howToEat());

Fruit Apple = new Apple();
System.out.println ("Apple how to Eat:" + apple.howTo.
                                          Eat());

Fruit orange = new orange();
System.out.println ("Orange How to eat" + Orange.
                                howToEat());

}

}
```

IT23055

28| Java's Garbage collection mechanism is an automatic memory management system that helps reclaim memory by removing objects that are no longer accessible.

Java uses reachability analysis to determine if an objects is still in use:

1. objects referenced from active thread stacks static fields, and JNI references are considered reachable
2. Objects connected to GC roots remain alive
3. Those not linked to GC roots are marked for collection.
4. The GC reclaims memory occupied by unreachable objects,

IT23055

Types of garbage collector:

1. Serial GC
2. Parallel GC
3. Concurrent Mark sweep
4. G1 (Garbage First) GC
5. ZGC
6. Shenandoah GC.

IT23055

30)

```
import java.~~util~~Util.*;
public class ArrayDission {
    ~~public class Array~~
    public static void main (String [] args) {
        Scanner sc = new Scanner (System.in);
        int n = sc.nextInt();
if (n>=20) {
```

```
System.out.println("n must be greater than 20");
return;
}
int [] arr1 = new int[n];
System.out.println("Enter "+n+" ");
for(int i=0; i<n; i++){
arr[i] = se.nextInt();
}
int m = (int) Math.ceil (n/10.0);
int[]arr2 = new int[m];
System.out.println("Enter:"+m);
for (int i=0; i<m; i++){
arr2[i] = se.nextInt();
if (arr2[i]==0){
System.out.println("0 zero is not allowed.");
i--;
}}
```

IT23055

```
System.out.println("Quotient and remainder:");
for(int i=0; i<m; i++){
int quo =(int) Math.ceil((double) arr1[i]/arr2[i]);
int rem = arr1[i] % arr2[i];
System.out.println("For "+arr1[i]+"/"+arr2[i]
        +" : Quotient="+quo+", remainder
        ="+rem);
}
sc.close();
}
}
```

31)

IT23055

```
System.out.println("Quotient and remainder:");
for(int i=0; i<m; i++){
 int quo =(int) Math.ceil((double) arr1[i]/arr2[i]);
int rem = arr1[i] % arr2[i];
System.out.println("For "+arr1[i]+"/"+arr2[i]
                    +" : Quotient = "+quo+", Remainder
                    = "+rem);
}
sc.close();
}
}
```

31.

IT23055

31]

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class CurrentDateTime{
public static void main (String[] args){
  LocalDateTime dt = LocalDateTime.now()
DateTimeFormatter fmt = DateTimeFormatter.ofPattern
                        ("yyyy-MM-dd HH:mm:ss");

String dtstr = dt.format(fmt);
 System.out.println ("Current Date and Time: "
                     +dtStr);

}
}
```

IT23055

32]

```
public class Counter {
  private static int count = 0;

public Counter(){
    count++;
if (count>50){
    count = 0;
  }
}
public static int getCount(){
   return count;
}
public static void main(String[] args){
for (int i=0;i<55;i++){
  new Counter();
System.out.println("count: "+ counter.getCount()
}
}
}
```

IT23055

33]

```
public class ExtremeFinder{
  public static int findExtreme (String[] type, int...
                                  numbers){

if(numbers.length==0) {

  throw new IllegalArgumentException ("At least
          One number must be provided
  }
int extr = numbers[0];
for (int num: numbers){
if (type.equals ("Smallest")){
   if (num < extr){
     extr=num;
   }
}else if (type.equals ("largest")){
  if (num > extr){
    extr=num;
  }
```

IT23055

```
} else {
    throw new IllegalArgumentException("Invalid
                type.");
}
}
return extr;
public static void main(String[] args){
 int x = findExtreme("Smallest", 5, 2, 9, 1);
 int y = findExtreme("Largest", 8, 3, 10, 4);
 System.out.println("Smallest : "+x);
 System.out.println("Largest: "+y);
}
}
```

IT23055

34)

~~s1.equals(s2)~~ →

Output:

true // content comparission: s1 and s2 have the same value.

False // Reference comparison: s1 and s2 are different Object.

True // s1 and s3 refer the same string Literal.

IT23055

40] public class CalcOps {

public static void main (String[] args){

try {

if (args.length<2){

throw new IllegalArgumentException ("Provide two integers as argument.");

}

int a = Integer.parseInt(args[0]);

int b = Integer.parseInt(args[1];

System.out.println("Sum: "+(a+b));

System.out.println("Diff: "+(a-b));

System.out.println("Prod: "+(a*b));

System.out.println("Quot: "+(b!=0?(a/b):

Undefined(div by zero)"));

}

IT23055

```
catch (NumberFormatException e) {
 System.out.println("Invalid input. Enter two numbers");
} catch (IllegalArgumentException e) {
 System.out.println(e.getMessage());

}
}
}
```