Sub :

Day

Time : Date : / /

1] import java.io.*;
import java.util.*;

public class NumberProcessor {
public static void main(String[] args) {
try {

File inputFile = new File("input.txt");
Scanner scanner = new Scanner(inputFile);
scanner.useDelimiter(",");

int maxNumber = Integer.MIN_VALUE;

while(scanner.hasNext()) {

int num = Integer.parseInt(scanner.next().trim());

if (num > maxNumber) {

maxNumber = num;

}
}
}

scanner.close();

IT-23022

Sub : Day Time : Date : / /

```
int sumOfNaturalNumbers = (maxNumber*(maxNumber+1))/2;

writer.close();
System.out.println("The result has written in output.txt file");
} catch (Exception e) {
    e.printStackTrace();
}
}
}
```

IT-23022

Sub: Day Time: Date: / /

21 Differences between static and final Fields and Methods in Java

Feature	Static	final
Definition	Belongs to the class rather than any specific object	Value/method can not be changed after initialization
usage	used when a value/method should be shared among all instances	used when a value/method should remain constant
Memory Allocation	Stored ~~when a value~~ in Method Area (Class Loader Memory) and shared across all instances	Stored in ~~hip~~ Heap memory or Method Area
Modification	Can be modified unless also declared final	Can not be modified after initialization
Access	Accessed via class name (Classname.staticfield)	Final variables can be accessed normaly final methods are used normaly

Sub: Day Time: Date: / /

Modification Inheritance	Static method can be inherited but can not be overriden	Final methods can not be overriden, final fields can not be re assigned

Accessing static fields/methods via Object:

We can access a static field or method through an object, but it is not recomended. In fact, doing so would ~~hav~~ give a compiler warning, as it is best practices to access static members using the class name.

3] import java.util.Scanner;

public class FactorionNumbers {

static int[] factorials = new int[10];

static {

for(int i = 0; i ~~<0~~ < 10; i++) {

factorials[i] = factorial(i);

```
    }
}
static int factorial(int n) {
int fact = 1;
for(int i = 1; i <= n; i++) {
    fact = fact * i;
    }
    return fact;
}
static boolean isFactorion(int number) {
int originalNumber = number;
int sumOfFactorials = 0;

while (number > 0) {
int digit = number % 10;
sumOfFactorials = sumOfFactorials * factorials[digit];
    number = number / 10;
}
```

Sub: Day Time: Date: / /

```
return sumOfFactorials == originalNumber;
}

public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the lower bound: ");
int lowerBound = scanner.nextInt();
System.out.print("Enter the upper bound: ");
int upperBound = scanner.nextInt();
System.out.println("Factorion numbers between"
        + lowerBound + "and" + "Upper bound");

for (int i = lowerBound; i <= upperBound; i++){
  if (isFactorion(i)) {
    System.out.println(i); }
  }
scanner.close();
}
}
```

IT-23022

Sub :

Day

Time : Date : / /

4] Differences between the class variable, instance variable, Local variable in bellow

Feature	class variable	instance variable	local variable
Declare with	Static keyword	No keyword (just declared)	Inside methods / blocks
Belongs to	The class	Each objects (instance)	The method / block
Scope	Class-wide	Object specific	method / blocks specific
Lifetime	As long as the class is loaded	As long as the object exists	As long as method is executed
Accessed via	Class name or instance	Instance reference.	inside the method

Sub :

Day

Time : Date : / /

5]

```
public class ArraySum{
public static int calculateSum(int[] arr){
int sum = 0;
for(int num : arr){
    sum = 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 the array
                    element is : "+sum);
}
}
```

IT-23024

Sub:

Day

Time: Date: / /

6)

Access Modifiers: Acess modifiers in Java are keywords that set the visibility and accessi accessibility of classes, methods, constructor and variables. They determine the scope of aceess for different classes and object

*Comparison of public, private and protected modifiers

Public	private	pro
① A public modifier or member can be accessed from anywhere	① A private member can be accessed from its class	② A protected member is accesible within same package an by subclass
② we use it when we want the member to be aceess by any other class	② It is used when we want to restrict access from other class.	② Used when we need to want to allow aceess to the member within the same package

LT-23022

Sub : Day Time : Date : / /

② public member can be accesible different package (subclass	③ private member can not be accessible different package	③ protected member can be accesible from different package
④ Public member are accesible from same package	④ private member are not accesible from same package	⑤ protected member are accessible from same package

```
import java.util.Scanner;
public class QudraticRoot {
Scanner scanner = new Scanner(System
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in)
System.out.print("Enter coefficients a, b, 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 root1 = (-b + Math.sqrt(discriminant))
    /(2*a);

double root2=(-b -Math.sqrt(discriminant))
    /(2*a)

doubleSmallestPositiveRoot = Double.MAX_VALUE;

if(root1>0) smallestPositiveRoot =Math.min(smallestPositive
                                    Root, root1);

if(root2>0) smallestPositiveRoot = Math.min(smallestPositiveRoot,root2);

if(smallestPositiveRoot == Double.MAX_VALUE) {
System.out.println("No positive real roots.");
}
```

```
else
{
System.out.println("The smallest positive root is:" +
                    smallestPositiveRoot);
}
}
scanner.close();
```

8] import java.util.Scanner;

```
public class CharacterTypeChecker {
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter a character: ");
char ch = scanner.next().charAt(0);
if(Character.isLetter(ch)) {
System.out.println("The character is a letter.");
}
else if(character.isWhitespace(ch)) {
   System.out.println("The character is a
                       whitespace.");
```

IT-23022

Sub: Day Time: Date: / /

```
}
else if (Character.isDigit(ch)) {
System.out.println("The character is a Digit");
  }
  else
   {
 System.out.println("The character is a
                    special channacte.");

    }
scanner.close();
}
}
```

IT-23002

Sub:

Day

Time: Date: / /

How to pass an Array to a function in Java:

```
public class ArrayPassingExample {
public static int sumArray(int[] numbers) {
 int sum = 0;
 for (int sum : numbers) {
    sum += num;
    }
 return sum;
}
public static void main(String[] args) {
int [] arr = {1, 2, 3, 4, 5}
 int sum = sumArray(arr);
 System.out.println("Sum of Array elements:"
                    + sum);
 }
}
```

IT-23022

Sub: Day Time: Date: / /

Q) Method overriding occurs when a subclass redefines a method from its superclass. It allows polymorphism, where a subclass provides a specific implementation of a method already defined in the parent class.

* Use of ~~superkey~~ super key:

The super keyword refers to the parent class and is used to

① call superclass methods that are overridden

② call superclass constructors.

Example: calling a superclass method

```
class parent {
  void display() {
    System.out.println("parent class method.");
  }
}
```

IT-23.02.2

Sub :

Day

Time : Date : / /

```
class child extends parent {
  void display() {
  super.display();
system.out.println("child class method.");
}
}
public class SuperMethodExample {
public static void main(String[] args) {
 child obj = new Child();
 obj.display();
}
}
```

IT- 23022

Sub :

Day

Time : Date : / /

potential issues when overriding methods:

① Access Restrictions:

- You ~~can n~~ cannot reduce the visibility of an overriden method. For example, if the super-class method is public, the overriden method can not be private.

② Exception:

③ Calling super() in constructors

Example:

```
class parent {

Parent (String name) {
System.out.println("parent constructor : " +name);
} }

class child extends parent {

child (string name) {
```

```
super(name);
System.out.println("Child constructor: " + name);
}
}
public class superConstructorExample {
public static void main(String[] args) {
Child obj = new Child("Java");
}
}
```

10] Difference between static and non-static member in Java

Feature	Static members	Non-static members
Belongs to	Class itself	individual object
Memory ussage	Allocated once in memory	Each object gets its own copy

Invocation	Acceessed using className.method()	Acceessed using object .method
Modification	Change affect all instanee	Changes affeeted only the specifre objeet
Example: ~~static int count~~	static int count ;	int instanceVar;

Example program demonstrating static and non static members

```
Class Example{
static int staticVar =10;
int instanceVar=20;
Static void staticMethod() {
System.out.println("static method : staticVar
                    =" + staticvar);
}
void instanceMethod(){
```

```
System.out.println("Instance method : instanceVar=
                    " + instanceVar);
    }
}
public class StaticVsNonStaticDemo {
public static void main(String[] args) {
Example.staticMethod();
Example obj=new example();
obj.instanceMethod();
    }
}
```

Palindrom checkan prognam:

```
import java.util.Scanner;
public class PalindromeChecker {
public static boolean isPalindrome(String str) {
```

IT-23022

Sub:

Day

Time: Date: / /

```
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 sc = new Scanner (System.in);
System.out.print("Enter a number or
                  string:");
String input = sc.next();
```

IT-23022

Sub :

Day

Time : Date : / /

```
if (isPalindrome(input)){
System.out.println("It is a palindrome.");
}
else
{
System.out.println("It is not a palindrom");
}
seannen.close();
}
}
```

IT-23022

Sub: Day Time: Date: / /

11] Class Abstraction:

(i) Abstraction is the concept of hiding implementation details and showing only essential features.

(ii) Implemented using abstract classes and interfaces.

example:

```
abstract class Vehicle {
abstract void start();
} class Car extends Vehicle{
void start() {
System.out.println("Car starts with a key.");
}}
public class AbstractionExample{
public static void main(String[] args){
```

```
Vehicle myCar = new Car();
myCar.start();
}}
}
```

~~Encapsulat~~

Encapsulation in Java: Encapsulation means wrapping data (variables) and methods into a single unit (class). Data is hidden using private access modifiers and accessed via getters and setters.

example:

```
Class Person{
private String name;
public void setName(String name){
this.name = name;
}
public String getName(){}{
```

IT-23022

Sub: Day Time: Date: / /

```
    return name;
  }
}
public class String getName() {}

public class EncapsulationExample {
  public static void main(String[] args) {
    Person p = new Person();
    p.setName("John");
    System.out.println("Person's name: "
        + p.getName());
  }
}
```

IT-23022

Sub :

Day

Time : Date : / /

Diffenences between abstract class and Interface :

Feature	Abstract class	Interface
methods	Can have both abstract and concrete methods	Only abstract methods (java 8+ allows default methods)
Constructor	Can have constructor	Can not have constructor
Vanriables	Can have instance variables	Only public static final variables
Inheritence	Allows single inheritence	Allows multiple inheritence
usage	Used when objects share common behairion but some methods need implementation	Used for defining a construct that multiple classes implem

IT-23022

Sub:

Day

Time: Date: / /

12)

```
class BaseClass {
void printResult(String message) {
    System.out.println(message);
}
}
class SumClass extends BaseClass{
double computeSum() {
double sum = 0;
for(double i = 1.0; i >= 0.1 ; i -= 0.1) {
 sum += i ;
 }
return sum;
}
}
class DivisorMultipleClass extends BaseClass{
```

IT-23022

Sub:

Day

Time: Date: / /

```
int gcd(int a, int b) {
return b == 0 ? a : gcd(b, a % b);
}
int lcm(int a, int b) {
  return (a * b) / gcd(a, b);
}
}
class NumberConversionClass extends BaseClass {
void converter(int num) {
System.out.println("Binary: " + Integer.toBinary
                    String(num));
System.out.println("Octal: " + Integer.toOctal
                    String(num));
System.out.println("Hexadecimal: " +
                    Integer.toHexString(num));
} }
```

IT-23022

Sub: Day Time: Date: / /

```
Class CustomPrintClass extends Baseclass {
void pr(String message) {
System.Out.println("***" + message + "***");
}
}
public class Mainclass {
public static void main(String[] args) {
SumClass sumObj = new SumClass();
DivisorMultipleClass gcdLcmObj = new NumberConversionClass;
NumberconversionClass convObj = new DivisorMultipleClass();
CustomPrintClass printObj = new CustomPrintClass();
printObj.pr("sum of series: " + sumObj.ComputSum());
printObj.pr("GCD of 24 and 36:" + gcdLcmObj.gcd(24,36));
convObj.convent(25);
}
}
```

IT-23022

Sub :

Day

Time : Date : / /

14] Significance of Big integer :

① BigInteger is used for handeling very large number beyond the limit of long (max value $2^{63} - 1$)

② It supports arithmatic operations, bitwise operation and modular arithmatic

③ Commonly used for cryptography, combinatorial calculations and large factorial computations

* Java program to compute Factorial using Big Integer :

```
import Java.math.BigInteger;
import Java.util.Scanner;
public static BigInteger factorial(int num){
    BigInteger fact = BigInteger.ONE
```

IT-23022

Sub:

Day

Time: Date: / /

```
for(int i=2; i<=num; i++){
fact = fact.multiply(BigInteger.valueOf(i));
}
return fact;
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Enter a number: ");
int num = scanner.nextInt();
BigInteger result = factorial(num);
System.out.println(num + "! is \n" + result);
scanner.close();
}
}
```

IT- 23022

Sub :

Day

Time : Date : / /

2] Comparison of Abstraction using Abstract classes and Interfaces in Java.

Feature	Abstract class	Interface
purpose	Used for partial abstraction	Used for full abstractio
Method Implementation	Can have abstract methods and concrete methods	Can have abstract methods, default methods
Fields (Instance variable)	Can have instance variable (with any access modifier	Can only have public static final constants
Constructor	Can have constructor	Can not have constructor
Access Modifiers for method	Methods can have any access modifier	Abstract methods are always public
Inheritance	Supports single inheritence	Supports multiple inheritan
Use case	When a base class needs to provide common behavior along with abstract methods	When multiple unrelated classes need to share a common contract but not implement

IT-2302

Sub:

Day

Time: Date: / /

* When to prefer an abstract class over an interface:

① When I need to provide common Functionality

ⓘ If We want to define some shared code for subclasses use an abstract class.

example: A vehicle class where startEngine() has a concrete implementation, but move() is abstract.

② When we need fields

③ When we expect future Enhancements.

④ When using a single Base class makes sense.

Can a class implement multiple Interfaces in Java.

Yes, a class can implement multiple interfaces in Java. This allows flexibility and multiple inheritance of type definitions.

Example:

IT-23022

Sub: Day Time: Date: / /

~~inheritance~~

```
interface A {
 void methodA();
            }

interface B {
 void methodB();
            }

class C implements A, B {
 public void methodA() {
 System.out.println("method A from Interface A");
                        }

 public void methodB() {
  System.out.println("method B from Interface B")

  }
 }
```

Here, class C implements both A and B, inheriting their method contracts.

IT-23022

Sub :

Day

Time : Date : / /

* Implication of using Multiple Interfaces:

① Conflicting method signatures

(i) If two interfaces have methods with the same signature but different purposes, implementing both in a single class may cause confusion

(ii) The class must provide a single implementation potentially leading to ambiguous behavior,

② Default methods in Interfaces (Java 8+)

(i) If two interfaces provide default methods with the same signature, the implementing class must override the method to resolve the conflict

16) polimorphism: Polymorphism is one of the four key pillars of Object-Oriented Programing It allows objects of different classes to be treated as objects of a commo supertype. In Java, polymorphism primarily manifests in two forms.

17-23022

Sub:

Day

Time: Date: / /

1. Compile-time polymorphism (method overloading)
2. Runtime polymorphism (method overriding)

Polymorphism and Dynamic Method Dispatch:

Dynamic method Dispatch (also called runtime polymorphism) is the mechanism by which a call to an overridden method is resolved at runtime rathan than compile-time. It occurs when a superclass reference variable refers to a subclass object, and the overridden method in the subclass in invoked

Key concept: The method that gets executed depends on the actual object type, not the references type

Example of polymorphism using Inheritence and method overridding:

IT-23022

Sub: Day Time: Date: / /

```
class Animal {
    void makesound() {
    System.out.println("Animal makes a sound");
    }
    }
class Dog extends Animal{
 void makeSound() {
System.out.println("Dog barks");
}
}
class Cat extends Animal {
void makeSound() {
 System.out.println("cat meows");
}
}
public class polymorphismExample{
public static void main(String[] args){
Animal myAnimal;
```

Sub :

Day

Time : Date : / /

```
myAnimal = new Dog();
myAnimal.makeSound();

myAnimal = new Cat();
myAnimal.makeSound();
}
}
```

Here, even though the references type is Animal Java calls the overridden method in the actual object type (Dog or Cat). This is dynamic method dispatch

* Impact of Polymorphism on Performance:

① Method Lookup at Runtime

- Method calls in polymorphism involve dynamic method resolution, which adds a slight performance cost compare to direct method calls

② Cache performance

③ Memory usage

IT-23022

Sub: _____ Day | | | | | | | |

Time: Date: / /

* Trades-off Between Polymorphism and specific method calls:

Aspect	Using Polymorphism	Using Specific method calls
Flexibility	High	Low
Code Reusability	High	Low
Performance	Slightly slower	Faster
Maintainability	Easier to maintain	more difficult to update

[7] Difference between Arraylist and Linked List:

Arraylist: Uses a dynamic array

Linked list: Uses a doubly linked list

* Time complexity comparison;

IT-23022

Sub: _____ Day | | | | | | | |

Time: Date: / /

Operation	Array list	LinkedList
Access	O(1) - Fast, direct access	O(n) - Requires traversal
Insertion at End	O(1) Amortized	O(1)
Insertion at Middle	O(n) - Requires shifting	O(n) Require traversa
Deletion at End	O(1)	O(1)
Deletion at Middle	O(n) - Requires shifting	O(n) - Requires traversal

* When we use:

Array list:

(i) Fast random access is needed

(ii) Memory efficiency is important.

(iii) Frequent addition at the end

Linked list:

(i) Frequent insertion/delation in the middle or begining

(ii) List size changes frequently

IT-23022

Sub:

Day

Time: Date: / /

performance impact on Large Data sets:

ArrayList: Better cache performance, less memory overhead

LinkedList: Higher memory usage, O(n) traversal for access

181

```
import java.util.Random;
import java.time.Instant;
public class CustomRandomGenerator {
private static final int[] seedArray = {3, 7, 11,
        19, 23, 31, 37, 41, 43};
private static final int MODULO = 100;
public static void myRand(int n) {
Random random = new Random(Instant.now().
        toEpochMilli());
System.out.println("Generating" + n + "Random
        numbers:");
```

IT-23022

Sub :

Day

Time : Date : / /

```
for (int i=0; i<n; i++) {
int randIndex = random.nextInt(seedArray.length);
int randNumber = seedArray[randIndex] * (int) Instant.
                 now().toEpochMilli()) % MODULO

System.out.println(randNumber);

try {
    Thread.sleep(10);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}

public static int myRand() {
Random random = new Random(Instant.now().to
                EpochMilli()

public static int myRand(int min, int max) {
    if (min > max) {
throw new IllegalArgumentException("Min value
    must be < less than max value.");
```

IT-23022

Sub: Day Time: Date: / /

```
}
Random random = new Random (Instant.now().
toEpochMilli());
return random.nextInt((max-min) +1)+min
}
public static void main(String[] args){
myRand(5);
System.Out.println("\nSingle Random number:"+
                    myRand());

System.out.println("\n Random number between
                    100 and 500: "+ myRand(100, 500));

}
}
```

IT-23022

Sub:

Day

Time: Date: / /

[2] Multithreating: Multithreating is a process that allows multiple tasks to run simultaneously, when multiple threads can execute in parallel. In java, multithreading is implemented using the thread class and the runnable interface

* Difference between Thread class and Runnable Interface:

Feature	Thread class	Runnable Interface
Implementation	Requires extending the thread class	Requires implementing the runnable interface
multiple inheritence	Java does not support multiple class inheritance so if you extend thread you can not extend any other class	Since it is an interface you can still extend another class
Code resuability	Less resuable because extending thread means the class can not extend anything else	more resuable as it allows extending other classes as well

IT- 23022

Sub: Day Time: Date: / /

Common issue for multi-threading:

1. Race condition:
2. Deadlock
3. Starvation
4. Livelock

synchronized keyword for thread safety:

When multiple threads access the same resource, data inconsistency can occur. Using the synchronized keyword, we can ensure that only one thread can execute a critical section at a time

Use synchronised for thread safety example:

```
class Counter {
private int count = 0;
public synchronized void increment() {
```

```
count ++;
}
public int getCount() {
return count;
}
}
```

Example of deadlock:

```
class DeadlockExample {
static final object lock1 = new object();
static final object lock2 = new object();
public static void main(String[] args) {
Thread t1 = new Thread(() -> {
    synchronized (lock1) {
    System.out.println("Thread 1: Holding
    lock1....");
try { Thread.sleep(100); } catch
```

```
catch(interruptedException e) { }
synchronized lock(2) {
System.out.println("Thread 1: Acquired lock 2!");
}
} });

Thread t2 = new (Thread(() -> {
synchronized (lock 2) {
System.out.println("Thread 2 : Holding lock2..");
try { Thread.sleep(100); } catch(InterruptException e)
{ } synchronized (lock 1) {
System.out. println ("Thread 2 : Acquired lock 1");
}
} });

t1.start();
t2.start();
```

IT-23022

Sub: Day Time: Date: / /

solution to avoid deadlock:

1. Maintain a consistent lock order
2. Use try-lock mechanism
3. Use timeout to prevent infinitive waiting

2 Q

Exception handeling in Java: Exception handeling is a way to manage error in Java so that the program does not crash unexpectedly. it is done by:

1. Try - Catch Block
2. Finally Block
3. Throws keyword

example:

```
try { int result = 10/0; }
  catch (ArithmaticException e) {
System.out.println("can not devide by zero");
  } finally {
System.out.println("This code always run");
}
```

IT-23022

Sub: Day Time: Date: / /

Checked vs Unchecked Exceptions:

Type	Example	When does it Occur?	Needs handeling
Checked Exception	IOException	At compile time	Yes, must handle with try catch or throws
Unchecked Exception	NullPointer-Exception	At run-time	No, program may crash if not handled

Creating and Throwing custom exceptions:

We can create our own exception class by extending Exception or RuntimeException.

Example:

```
class MyException extends Exception {
public MyException(String message) {
  super(message);
  }
}
```

IT-23022

Sub: Day Time: Date: / /

```
public class customExceptionExample {
public static void main(String[] args) {
try{
throw new MyException("This is a custom exception!");
}
catch (MyException e) {
  System.out.println(e.getMessage());
}
}
}
```

throw vs throws

Keyword	Purpose	Example
throw	Used to throw an exception inside a method	throw new ArithmeticException ("Error!");
throws	Used in a method signature to indicate that the method may cause an exception	public void myMethod() throws IOException { }

IT-23022

Sub: Day Time: Date: / /

Preventing Resource Leaks:

If an exception occurs when working with resources (like files, database connection), they must be closed properly to avoid memory leaks.

Using Try-with Resources: (close)

```
import java.io.*;
public class TrywithResourcesExample {
public static void main(string[] args) {
try (FileReader file = new FileReader("file.txt"))
    { } catch (IOException e) {
System.out.println("File not found");
}
}
}
```

IT-23022

Sub:

Day

Time: Date: / /

22) Differences Between Hashmap, Treemap, and LinkedHashmap:

Feature	Hashmap	Treemap	Linkedhashmap
Ordering	No order	Sorted (Natural/ custom order)	Insertion order maintain
Data structure	Hash table	Red-Black Tree	Hash table + Doubly Linked list
Time complexity	O(1) avg, O(n) worst	O(log(n)) for all all operations	O(1) avg, O(n) worst
Null Keys	yes	No	yes
Best use case	Fast lookup, no order needed	Sorted data retrieval	Maintain insertion order

* Ordering Differences between Hashmap and Treemap:

(i) Hashmap → No specific order

(ii) Treemap → maintain sorted order based on natural ordering or a custom comparator

IT-23022

Sub:

Day

Time: Date: / /

A+

28] Advantage of using Executor service over manually managin Threads:

① Thread polling

② Better Resource management

③ Task Submission Flexibility

④ Graceful shutdown

⑤ Future object support

⑥ Automatic Exception Handling

* Difference between submit and execute:

method	Type	Exception Handling	Use case
execute (Runnable task)	void	Throws an exception immediatly if the task fails	Suitable for tasks that do not return results
Submit(callable/ Runnable task	Future< ?>	Captures exceptions inside Future.get()	Suitable for tasks requiring a return value or exception handling

IT-23022

Sub :

Day

Time : Date : / /

Benifits of using callable Instead of Runnable

1. Return values
2. checked Exception Handling
3. Future integration

Example: Using ExecutorService to manage a Thread pool

```
import java.util.concurrent.*;
public class ExecutoServiceExample {
public static void main(String[] args) {
ExecutorService executor = Executors.newFixedThreadpool(3);
executor.execute(() -> system.out.println("Task 1 executed by" + Thread.concunnent Thread().getName()));
```

IT-23022

Sub:

Day

Time: Date: / /

```
Future<Integer> future = executor.submit(() -> {
System.out.println("Task 2 executed by " +
Thread.currentThread().getName());
});
try {
int result = future.get();
System.out.println("Result from task 2: " + result);
} catch (InterruptedException | ExecutionException e)
{ e.printStackTrace();
}
executor.shutdown();
}
}
```

IT-23022

Sub :

Day

Time : Date : / /

25) Exception Handling considerations:

① Specificity of Exception Handling

② Avoiding Empty Catch Blocks

③ Re-throwing Exceptions

④ Custom Exceptions

⑤ Finally Blocks

⑥ Exception Chaining

⑦ Avoiding Unnecessary Exception Handling

program to calculate the area of a circle with setRadius Method that throws an Exception

```
Class InvalidRadiusException extends Exception {
public InvalidRadiusExeception (String message) {
  super(message);
}
}
```

IT-2302

Sub:

Day

Time: Date: / /

```
class circile{
private double radius;
public void setRadius(double radius) throws
InvalidRadiusException {
if (radius<0) {
throw new InvalidRadiusException("Radius can
not be negative"); }
this.radius= radius;
}
public double calculateArea() {
return Math.PI *radius * radius;
}
}
publie class CircleArealCalculaton {
public static void main(String[] args){
Cirele circle = new Circle();
```

IT-23022

Sub :

Day

Time : Date : / /

```
try {
circle.setRadius(-5);
System.out.println("Area of the circle:" +
        circle.calculateArea()); }
catch (InvalidRadiusException e) {
 System.out.println("Error: " + e.getMessage());
 }
 }
}
```