1. Write the code to add a JLabel called name to a panel called userPanel

userPanel.add(name);

1. How can you set the text in a JTextField? Assume the JTextField object is called field and the text to be displayed is Hello.

field.setText("Hello");

1. How can you set the text in a JLabel? Assume the JLabel object is called field (and has already been created) and the text to be displayed is Hello.

field.setText("Hello");

1. How do you create a button called ok with the label “Ok”. Do not write the modifier.

JButton ok=new JButton ("Ok");

1. How do you create a button called cancel with the label “Cancel”. Do not write the modifier.

JButton cancel = new JButton ("Cancel");

1. Write the for each loop to iterate through a LinkedList called agents? call the element Agent agent.

for (Agent agent : agents)

1. How do you iterate though a LinkedList of Object called objects?

 for (Object object : objects)

Divide by zero exception:

import java.io.\*;

class GFG {

    public static void main(String[] args)

    {

        int a = 5;

        int b = 0;

        try {

            System.out.println(a / b); // throw Exception

        }

        catch (ArithmeticException e) {

            // Exception handler

            System.out.println(

                "Divided by zero operation cannot possible");

        }

    }

}

**Java collection:**

The Java Collection Framework is a unified architecture for representing and manipulating collections of objects. It provides a set of interfaces and classes to handle data structures like lists, sets, maps, and queues, offering operations such as searching, sorting, insertion, manipulation, and deletion. The framework simplifies programming by providing data structures and algorithms that are easy to use and consistent.

**Main Components of the Java Collection Framework**

1. **Interfaces**: These are abstract data types that represent collections. Key interfaces include:
   * **Collection**: The root interface for most of the collections.
   * **List**: An ordered collection that allows duplicate elements.
   * **Set**: A collection that does not allow duplicate elements.
   * **Map**: An object that maps keys to values, with no duplicate keys allowed.
   * **Queue**: A collection used to hold multiple elements prior to processing.
2. **Classes**: Implementations of the interfaces, like:
   * **ArrayList, LinkedList**: Implement the List interface.
   * **HashSet, LinkedHashSet, TreeSet**: Implement the Set interface.
   * **HashMap, LinkedHashMap, TreeMap**: Implement the Map interface.
3. **Algorithms**: Methods to perform operations such as sorting and searching on collections, provided through the Collections class.
4. **Utility Classes**: Such as Collections and Arrays that provide static methods for collection operations.

**Differences Between List, Set, and Map**

1. **List**:
   * **Definition**: An ordered collection that can contain duplicate elements. Elements can be accessed by their position (index) in the list.
   * **Use Case**: Use a List when you need to maintain the order of elements and allow duplicates.
   * **Example**: Use a List for storing a sequence of tasks in a to-do list where the order of tasks is important.

List<String> tasks = new ArrayList<>();

tasks.add("Write report");

tasks.add("Review code");

tasks.add("Write report"); // Duplicates are allowed

**Set**:

* **Definition**: An unordered collection that does not allow duplicate elements.
* **Use Case**: Use a Set when you need to ensure that all elements are unique and the order does not matter.
* **Example**: Use a Set for storing unique user IDs in a system.

Set<String> userIds = new HashSet<>();

userIds.add("user123");

userIds.add("user456");

userIds.add("user123"); // Duplicate will not be added

**Map**:

* **Definition**: A collection that maps keys to values, where each key is unique. A Map does not extend the Collection interface.
* **Use Case**: Use a Map when you need to associate keys with values, such as storing user profiles where each user ID is associated with a user object.
* **Example**: Use a Map for storing a dictionary where words are mapped to their definitions.

Map<String, String> dictionary = new HashMap<>();

dictionary.put("apple", "A fruit");

dictionary.put("book", "A set of written pages");

dictionary.put("apple", "A tech company"); // The value for "apple" will be updated.

**Error handling**

The try-catch-finally block in Java is used for exception handling, allowing developers to manage and respond to runtime errors in a controlled manner. The purpose of this block is to handle exceptions gracefully without crashing the program, enabling the program to continue running or exit in a controlled way.

**Purpose of Each Component**

1. **try block**: The code that may cause an exception is placed inside the try block. If an exception occurs, the flow of control immediately jumps to the catch block (if defined).
2. **catch block**: This block is used to handle the exception. It catches exceptions thrown by the try block. You can have multiple catch blocks to handle different types of exceptions.
3. **finally block**: This block contains code that will always be executed after the try and catch blocks, regardless of whether an exception was thrown or caught. It is typically used for cleanup activities, like closing files or releasing resources.

**How It Works**

* When an exception occurs in the try block, control is passed to the matching catch block based on the exception type.
* If no exception occurs, the catch block is skipped, and control moves to the finally block (if present).
* The finally block always executes, even if there is a return statement or an exception occurs within the catch block.

**Example**

**public class ExceptionHandlingExample {**

**public static void main(String[] args) {**

**int[] numbers = {1, 2, 3};**

**try {**

**// Code that may throw an exception**

**System.out.println("Accessing element at index 3: " + numbers[3]);**

**} catch (ArrayIndexOutOfBoundsException e) {**

**// Handling the specific exception**

**System.out.println("Exception caught: " + e.getMessage());**

**} finally {**

**// Code that will always execute**

**System.out.println("Finally block executed.");**

**}**

**System.out.println("Program continues...");**

**}**

**}**

Output:

**Exception caught: Index 3 out of bounds for length 3**

**Finally block executed.**

**Program continues...**

**Arithmetic Exception:**

**void arrayAndArithmeticException() {**

**try {**

**// Creating an array (Java equivalent of a list)**

**int[] myArray = { 10, 20, 30, 40, 50 };**

**// Modifying the index to a valid one within the array bounds**

**int index = 5;**

**// Accessing an element in the array (this will cause ArrayIndexOutOfBoundsException)**

**int value = myArray[index];**

**System.out.println("Value at index " + index + ": " + value);**

**// Performing an arithmetic operation that causes ArithmeticException (division by zero)**

**int result = value / 0; // This will cause ArithmeticException**

**System.out.println("Division result: " + result);**

**}**

**catch (ArrayIndexOutOfBoundsException e) {**

**System.out.println("ArrayIndexOutOfBoundsException: The specified index is out of range.");**

**}**

**catch (ArithmeticException e) {**

**System.out.println("ArithmeticException: Division by zero is not allowed.");**

**}**

**}**

**invalid input or number overflow**

import java.util.Scanner;

public class MultiplyNumbers {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

boolean validInput = false;

while (!validInput) {

try {

System.out.print("Enter the first number: ");

long num1 = Long.parseLong(scanner.nextLine());

System.out.print("Enter the second number: ");

long num2 = Long.parseLong(scanner.nextLine());

// Perform multiplication

long result = num1 \* num2;

// Check for overflow

if ((num1 != 0 && result / num1 != num2) || (num2 != 0 && result / num2 != num1)) {

throw new ArithmeticException("Error: Number overflow occurred.");

}

System.out.println("Multiplication result: " + result);

validInput = true; // Exit loop after successful operation

} catch (NumberFormatException e) {

System.out.println("Error: Invalid input. Please enter valid numbers.");

} catch (ArithmeticException e) {

System.out.println(e.getMessage());

}

}

scanner.close();

}

}

### Iterator in Java Collection Framework

An **iterator** in the Java Collection Framework is an object that provides a way to access the elements of a collection sequentially without exposing the underlying structure. It is part of the java.util package and is a standard mechanism for traversing collections like lists, sets, and maps.

**Distinction Between Loop and Iterator**

1. **Loop (e.g., for-loop, while-loop)**:
   * A loop is a control structure that repeatedly executes a block of code until a specified condition is met.
   * It can be used with counters or indices, especially for arrays or collections that support indexing.
   * Loops are less flexible when it comes to removing elements from collections during traversal.
2. **Iterator**:
   * An iterator is a more flexible and safe way to traverse collections, especially when removing elements.
   * It provides a consistent interface across various types of collections.
   * Iterator handles the underlying complexity of collection traversal and modification.

**LinkedList<Agent> agents = new LinkedList<>();**

**// For each loop to iterate through the LinkedList**

**for(Agent agent : agents) {**

**// Access each agent element here**

**System.out.println(agent);**

**}**

**// Shape interface with an abstract method**

**interface Shape {**

**double area(); // Abstract method to be implemented by derived classes**

**}**

**// Circle class implementing Shape interface**

**class Circle implements Shape {**

**private double radius;**

**// Constructor**

**public Circle(double radius) {**

**this.radius = radius;**

**}**

**// Implementing the area() method for Circle**

**@Override**

**public double area() {**

**return Math.PI \* radius \* radius; // Area of a circle = πr^2**

**}**

**}**

**// Rectangle class implementing Shape interface**

**class Rectangle implements Shape {**

**private double width;**

**private double height;**

**// Constructor**

**public Rectangle(double width, double height) {**

**this.width = width;**

**this.height = height;**

**}**

**// Implementing the area() method for Rectangle**

**@Override**

**public double area() {**

**return width \* height; // Area of a rectangle = width \* height**

**}**

**}**

**// Main class to demonstrate the inheritance and implementation**

**public class Main {**

**public static void main(String[] args) {**

**Shape circle = new Circle(5); // Creating a Circle object**

**Shape rectangle = new Rectangle(4, 6); // Creating a Rectangle object**

**// Printing areas using polymorphism**

**System.out.println("Area of Circle: " + circle.area());**

**System.out.println("Area of Rectangle: " + rectangle.area());**

**}**

**}**

**// Abstract class Shape with an abstract method and a non-abstract method**

**abstract class Shape {**

**// Abstract method to be implemented by derived classes**

**abstract double area();**

**// Non-abstract method**

**public void displayShapeInfo() {**

**System.out.println("This is a shape.");**

**}**

**}**

**// Circle class extending the abstract Shape class**

**class Circle extends Shape {**

**private double radius;**

**// Constructor**

**public Circle(double radius) {**

**this.radius = radius;**

**}**

**// Implementing the abstract area() method for Circle**

**@Override**

**double area() {**

**return Math.PI \* radius \* radius; // Area of a circle = πr^2**

**}**

**}**

**// Rectangle class extending the abstract Shape class**

**class Rectangle extends Shape {**

**private double width;**

**private double height;**

**// Constructor**

**public Rectangle(double width, double height) {**

**this.width = width;**

**this.height = height;**

**}**

**// Implementing the abstract area() method for Rectangle**

**@Override**

**double area() {**

**return width \* height; // Area of a rectangle = width \* height**

**}**

**}**

**// Main class to demonstrate the abstract class inheritance**

**public class Main {**

**public static void main(String[] args) {**

**Shape circle = new Circle(5); // Creating a Circle object**

**Shape rectangle = new Rectangle(4, 6); // Creating a Rectangle object**

**// Displaying shape information using the non-abstract method from Shape**

**circle.displayShapeInfo();**

**System.out.println("Area of Circle: " + circle.area());**

**rectangle.displayShapeInfo();**

**System.out.println("Area of Rectangle: " + rectangle.area());**

**}**

**}**