Day 9

Employee ID: 201933938

Login ID: iamasif

Name: Shaik Asif

| Inheritance, Polymorphism, Encapsulation, Abstraction, Interfaces, Exception Handling, |
| --- |
| Collections Framework intro, Streams, File I/O, Multithreading overview |

**Exceptions:**

An exception orexceptionalevent is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore these exceptions are to be handled. An exception can occur for many different reasons, below given are some scenarios where exception occurs.

A user has entered invalid data.

A file that needs to be opened cannot be found.

A network connection has been lost in the middle of communications or the JVM has run out

of memory.

**Types Exceptions:**

**Checked exceptions:** A checked exception is an exception that occurs at the compile time,

these are also called as compile time exceptions. These exceptions cannot simply be ignored

at the time of compilation, the Programmer should take care of handle these exceptions.

**Unchecked exceptions:** An Unchecked exception is an exception that occurs at the time of

execution, these are also called as Runtime Exceptions, these include programming bugs,

such as logic errors or improper use of an API. runtime exceptions are ignored at the time of

compilation.

Doc 18 exception Handling in Docs tro Study - **completed**

Task 1:

What do you understand by exceptions?

An exception is like a problem or unexpected situation that happens when a program is trying to do something like dividing a number by zero, opening a missing file, or using something that's not there. If we don’t take care of it, the program can crash.

3 min 10.20 to 10.23

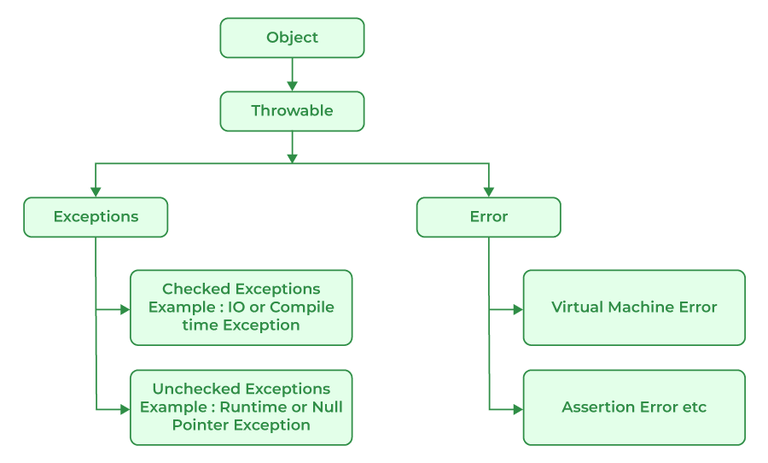
Task 2:

What are the categories of Exceptions do we have in Java? What are they?

**Two types are there**

1. Checked -while compiling
2. Unchecked - while running the code

5 min 10.24 to 10.2



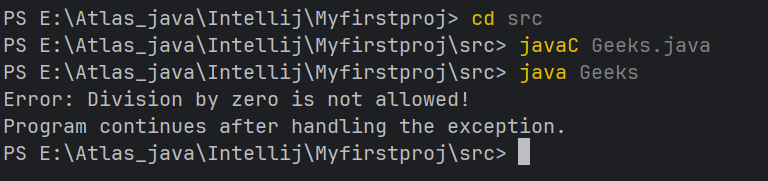
Plz refer the above image .. for more details.

Task 3:

Can you try the below code snippet and let me know which kind of exception is this

**ArithmeticException**

What is the output of the code..?



// Java program to demonstrates handling

// the exception using try-catch block

import java.io.\*;

class Geeks {

public static void main(String[] args)

{

int n = 10;

int m = 0;

try {

// Code that may throw an exception

int ans = n / m;

System.out.println("Answer: " + ans);

}

catch (ArithmeticException e) {

// Handling the exception

System.out.println(

"Error: Division by zero is not allowed!");

}

finally {

System.out.println(

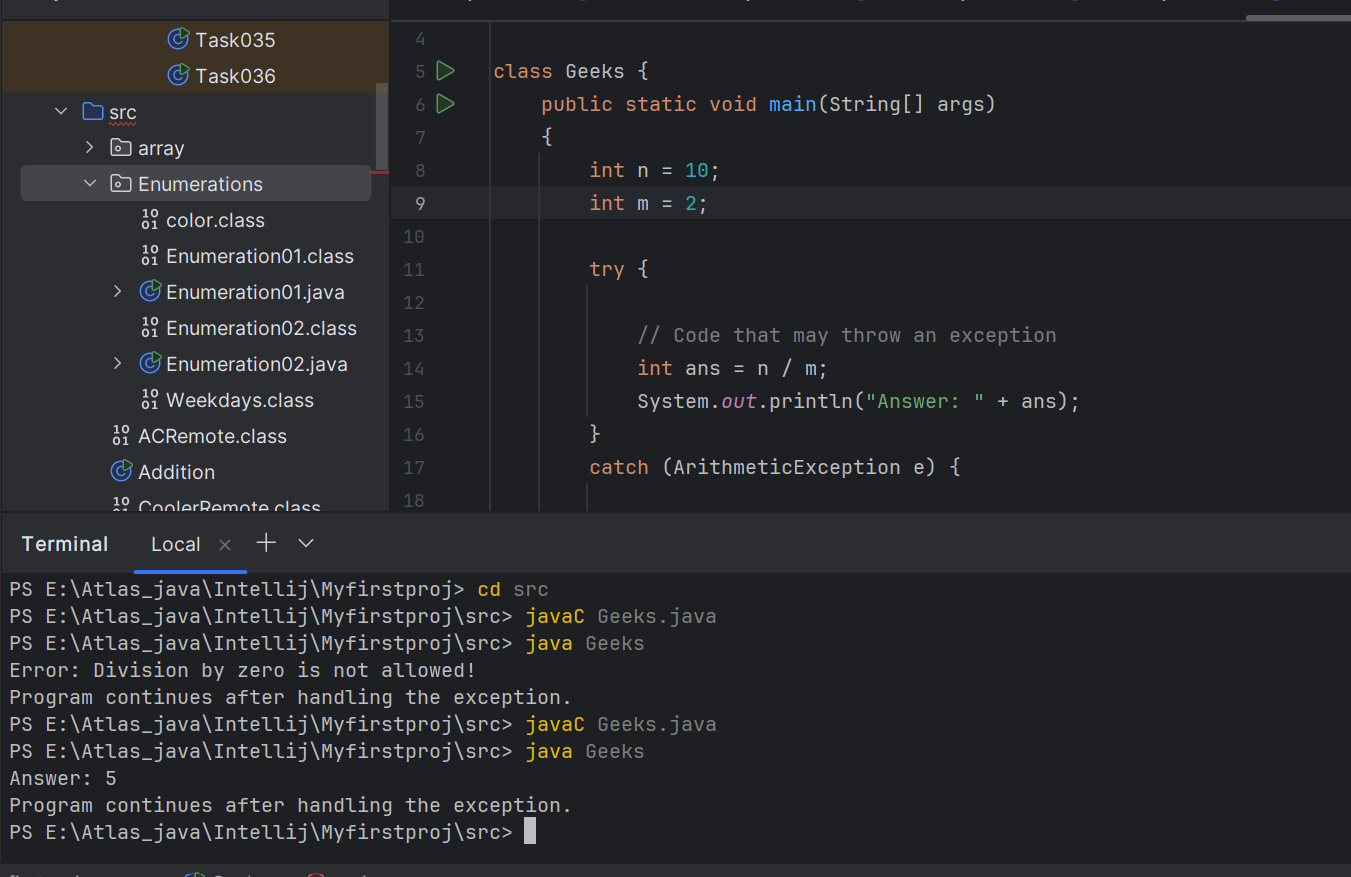
"Program continues after handling the exception.");

}

}

}

After adding m=2; below op:



Task 4:

List of checked and unchecked exceptions.

**Checked exceptions.**

| **Exception** |  | **Description** |
| --- | --- | --- |
| IOException |  | Input/output errors (e.g., reading files) |
| FileNotFoundException |  | File does not exist |
| SQLException |  | Database access error |
| ClassNotFoundException |  | Class not found at runtime |
| InterruptedException |  | Thread interrupted during sleep/wait |
| NoSuchMethodException |  | Requested method doesn't exist |
| ParseException |  | Error while parsing text (e.g., date formats) |
| InstantiationException |  | Trying to create an object of an abstract class or interface |
| IllegalAccessException |  | Accessing a class or field you don't have permission for |

**Unchecked exceptions:**

| **Exception** |  | **Description** |
| --- | --- | --- |
| ArithmeticException |  | E.g., divide by zero |
| NullPointerException |  | Accessing an object with null reference |
| ArrayIndexOutOfBoundsException |  | Array index out of range |
| StringIndexOutOfBoundsException |  | Invalid string index access |
| IllegalArgumentException |  | Passed invalid arguments to a method |
| NumberFormatException |  | Invalid conversion of string to number |
| ClassCastException |  | Invalid type casting |
| IllegalStateException |  | Method called at the wrong time/state |

10.45 to 10.49

Task 5:

Try with Multiple catch blocks …. Execute the below code snippet n display the out .. along with reason..

public class ExcepTest {

public static void main(String args[]) {

try {

int a[] = new int[2];

int b = 0;

int c = 1/b;

System.out.println("Access element three :" + a[3]);

}

catch (ArrayIndexOutOfBoundsException e) {

System.out.println("ArrayIndexOutOfBoundsException thrown :" + e);

}catch (Exception e) {

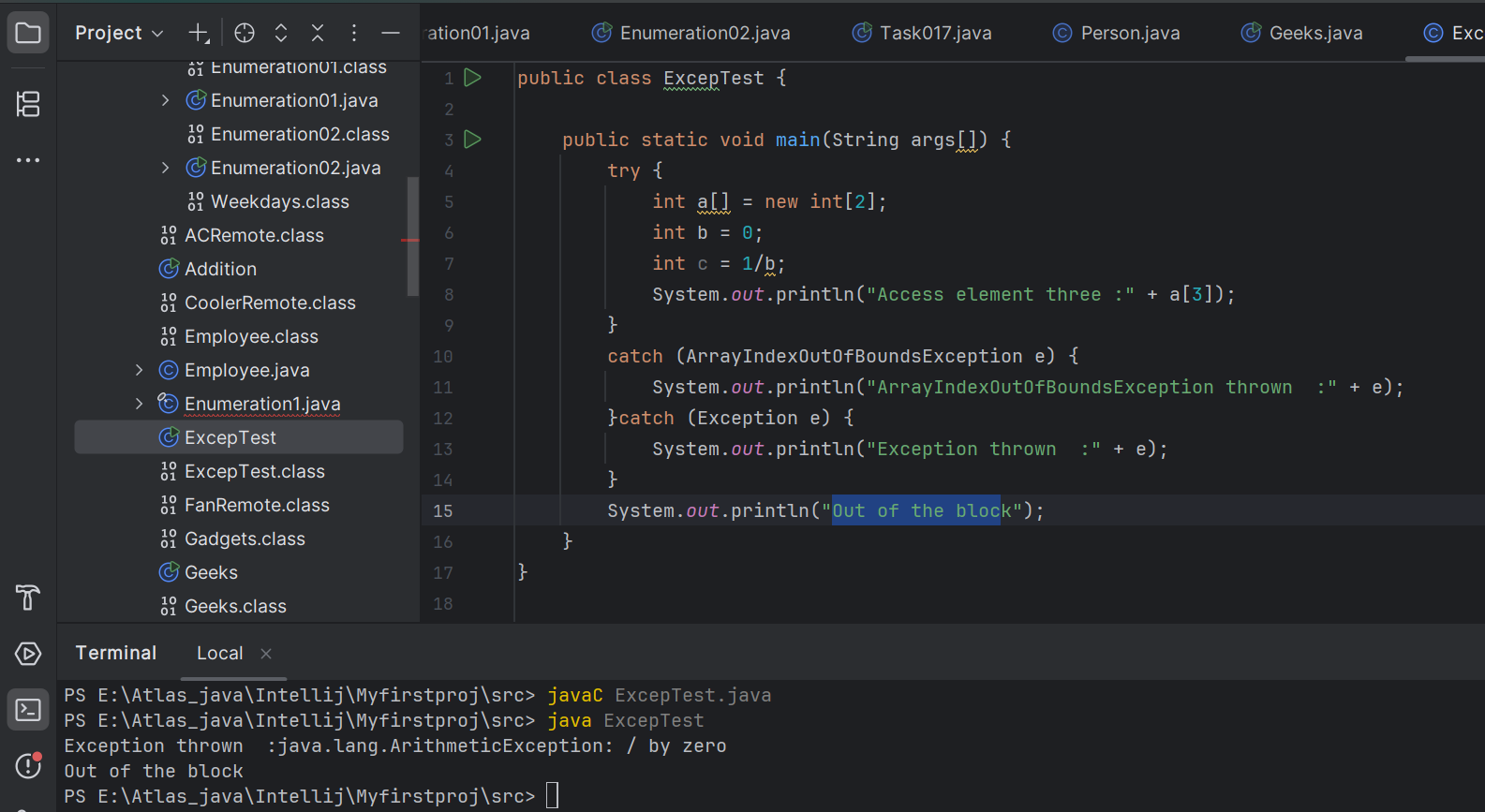
System.out.println("Exception thrown :" + e);

}

System.out.println("Out of the block");

}

}



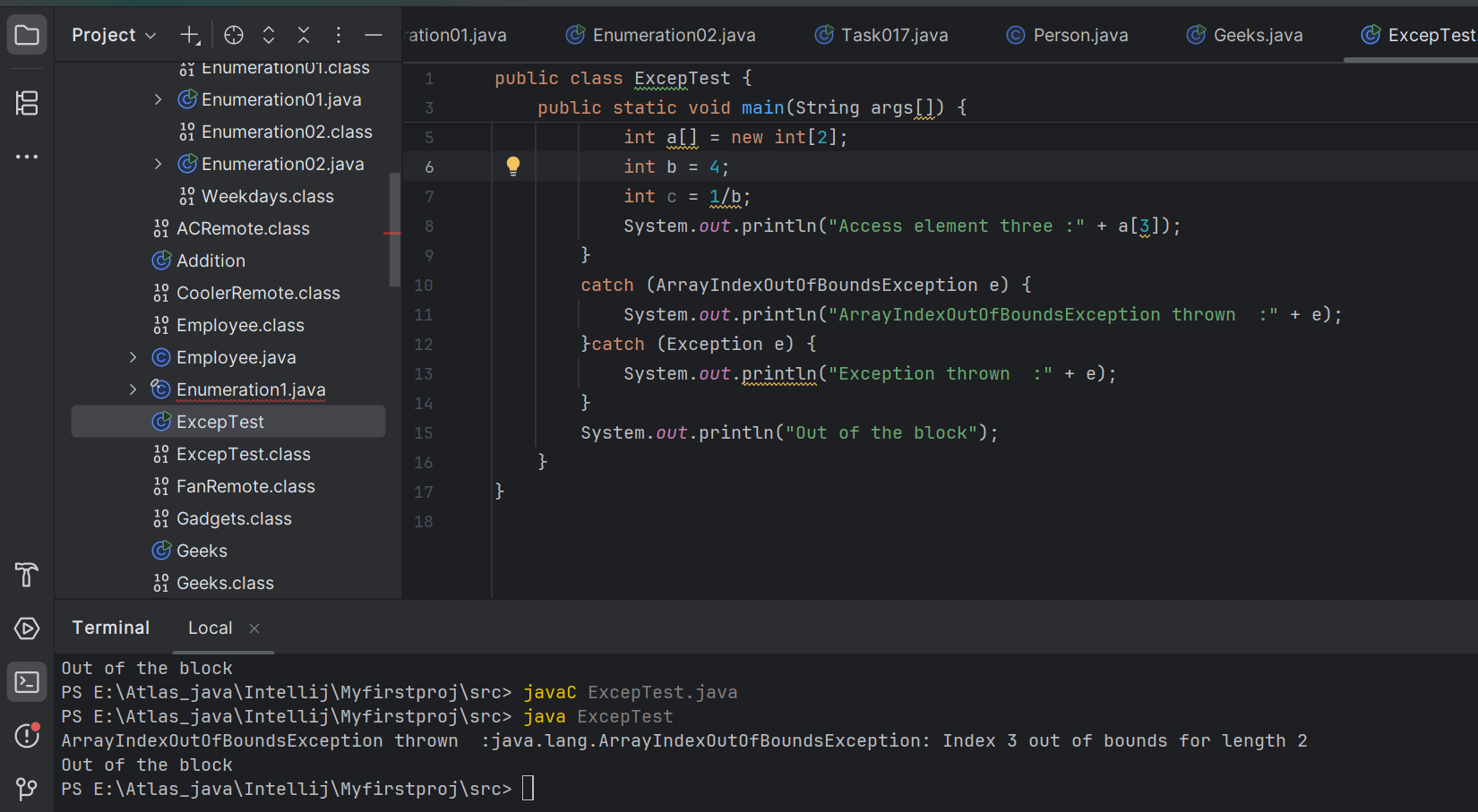
**Reason**

int c = 1 / b; causes an ArithmeticException (/ by zero)

As soon as the exception is thrown, the rest of the try block is skipped

The program looks for a matching catch block

**Here i changed the b value**



Task 6:

What is the output of the below code… give your reason for the output

public class ExcepTest {

public static void main(String args[]) {

try {

int a[] = new int[2];

int b = 0;

int c = 1/b;

System.out.println("Access element three :" + a[3]);

}

catch (ArithmeticException e) {

System.out.println("ArithmeticException thrown :" + e);

}

catch (ArrayIndexOutOfBoundsException e) {

System.out.println("ArrayIndexOutOfBoundsException thrown :" + e);

}catch (Exception e) {

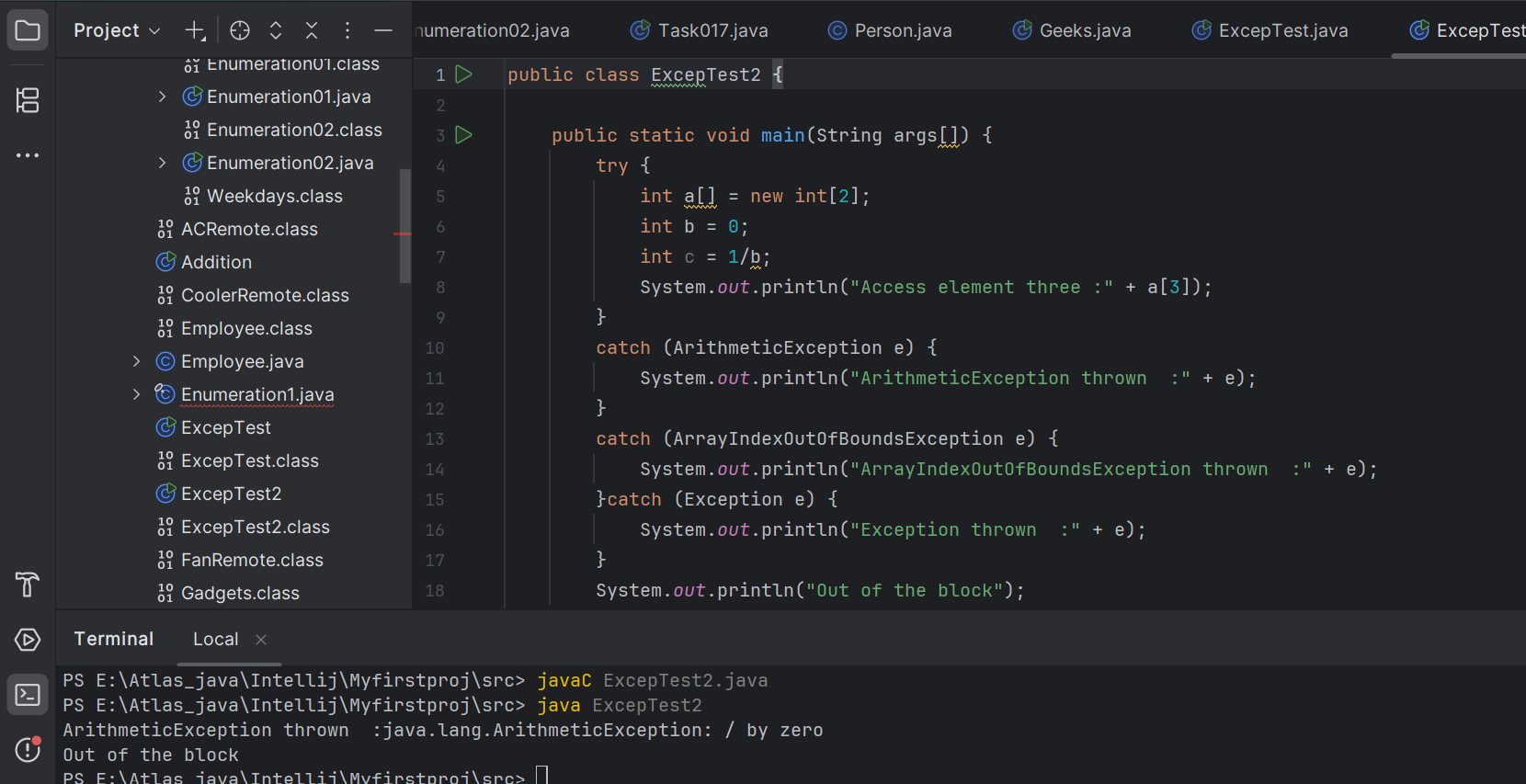
System.out.println("Exception thrown :" + e);

}

System.out.println("Out of the block");

}

}



**Reason:**

* The line int c = 1 / b; tries to divide 1 by 0

which causes an: ArithmeticException: / by zero

* This means the rest of the code inside the try block (including a[3]) is not executed.

10.56 to 11.00

Task 7:

In the below code we are having use multiple catch in a single statement: find the output and try to understand the code..

public class ExcepTest {

public static void main(String args[]) {

try {

int a[] = new int[2];

int b = 0;

int c = 1/b;

System.out.println("Access element three :" + a[3]);

}

catch (ArrayIndexOutOfBoundsException | ArithmeticException e) {

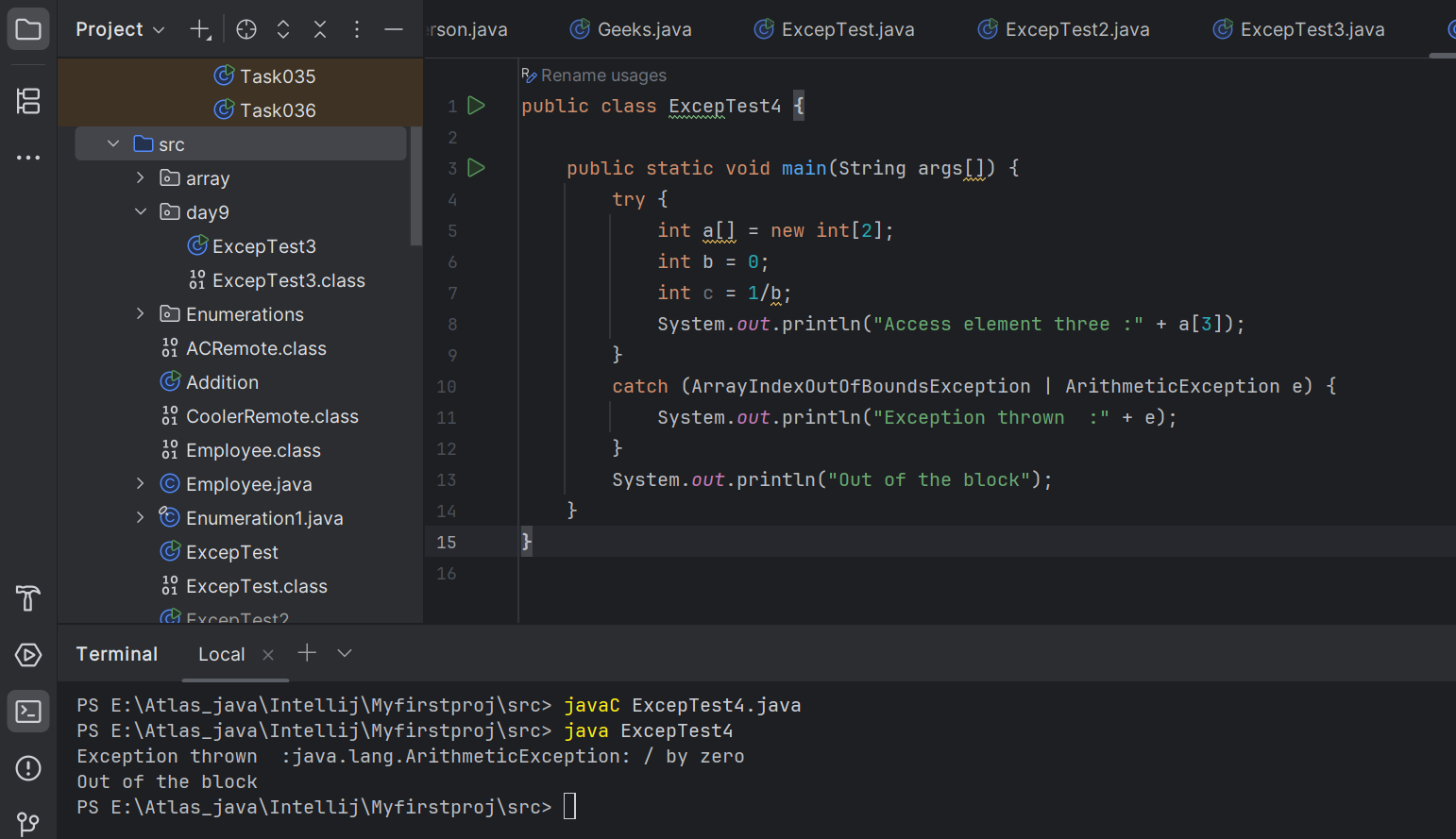
System.out.println("Exception thrown :" + e);

}

System.out.println("Out of the block");

}

}



**Reason:**

* int c = 1 / b; triggers an ArithmeticException because division by zero is illegal in Java.
* The program immediately jumps to the catch block and skips the line accessing a[3] (which would throw an ArrayIndexOutOfBoundsException if reached).

11.15 to 11.18

Nested try blocks

Task 008:

public class ExcepTest {

public static void main(String args[]) {

try {

int a[] = new int[2];

try {

int b = 0;

int c = 1/b;

}catch(Exception e) {

System.out.println("Exception thrown: " + e);

}

System.out.println("Access element three :" + a[3]);

}

catch (ArrayIndexOutOfBoundsException e) {

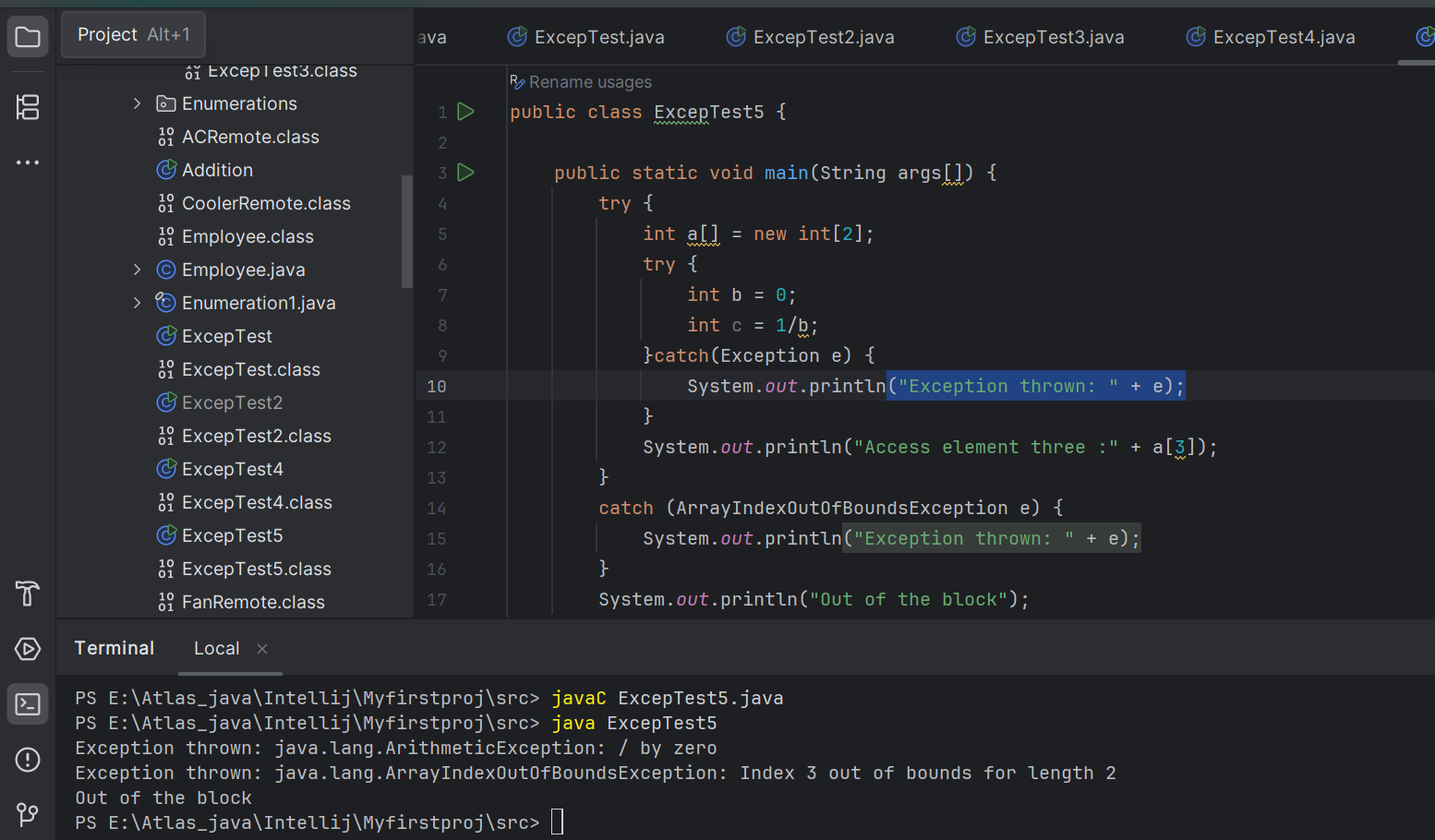
System.out.println("Exception thrown: " + e);

}

System.out.println("Out of the block");

}

}



Throw and Throws:

| **Feature** | **throw** | **throws** |
| --- | --- | --- |
| Purpose | Actually throws an exception | Declares an exception may be thrown |
| Used in | Method body | Method signature |
| Follows with | Exception object | Exception class name(s) |
| Can throw | Only one exception at a time | Can declare multiple exceptions |

Task 009

// Demonstrating how to throw an exception

class MyClass {

static void fun() throws IllegalAccessException

{

System.out.println("Inside fun(). ");

throw new IllegalAccessException("demo");

}

public static void main(String args[])

{

try {

fun();

//method2(); → arrayindex…

//Method3() —> file not found….

}

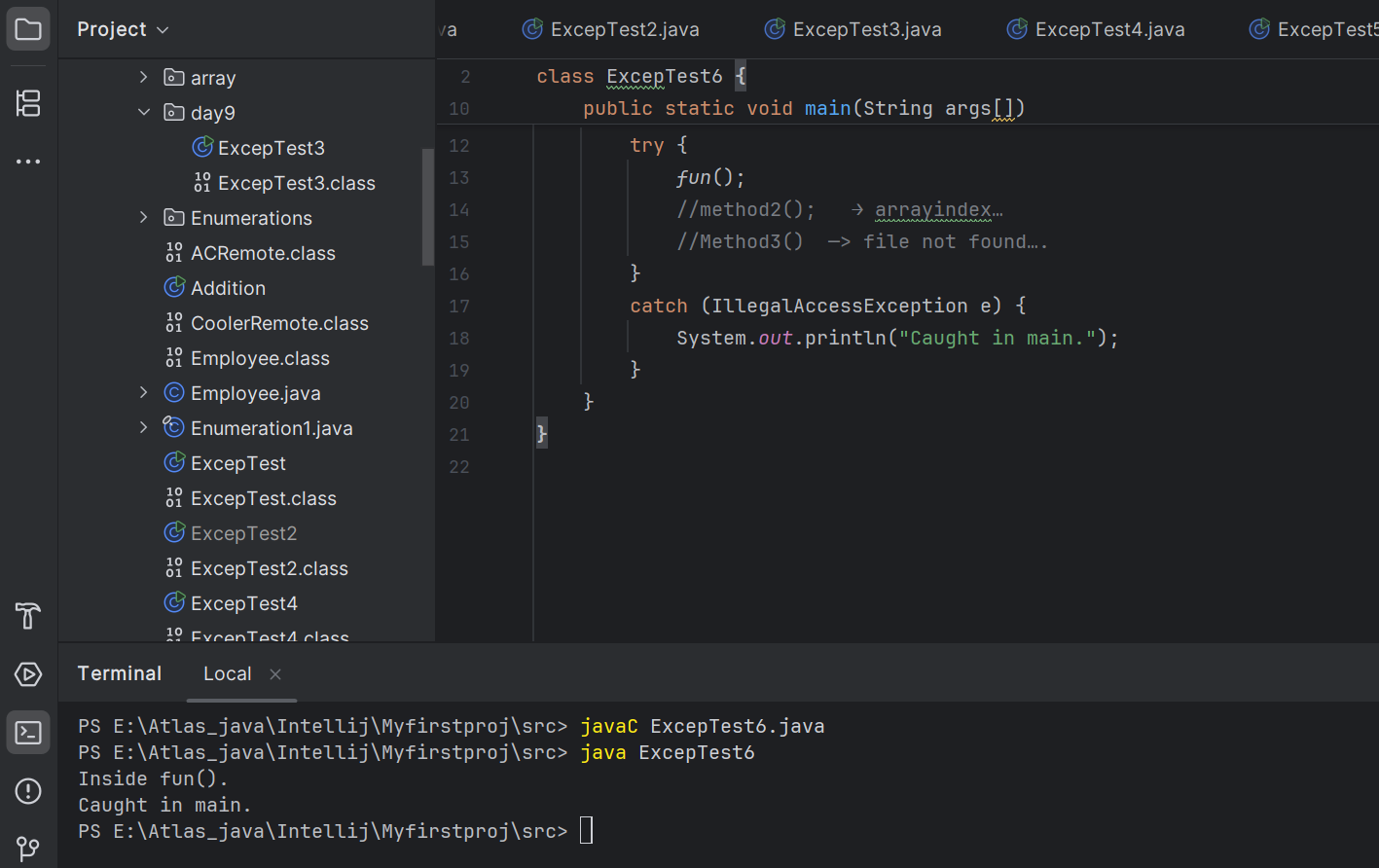
catch (IllegalAccessException e) {

System.out.println("Caught in main.");

}

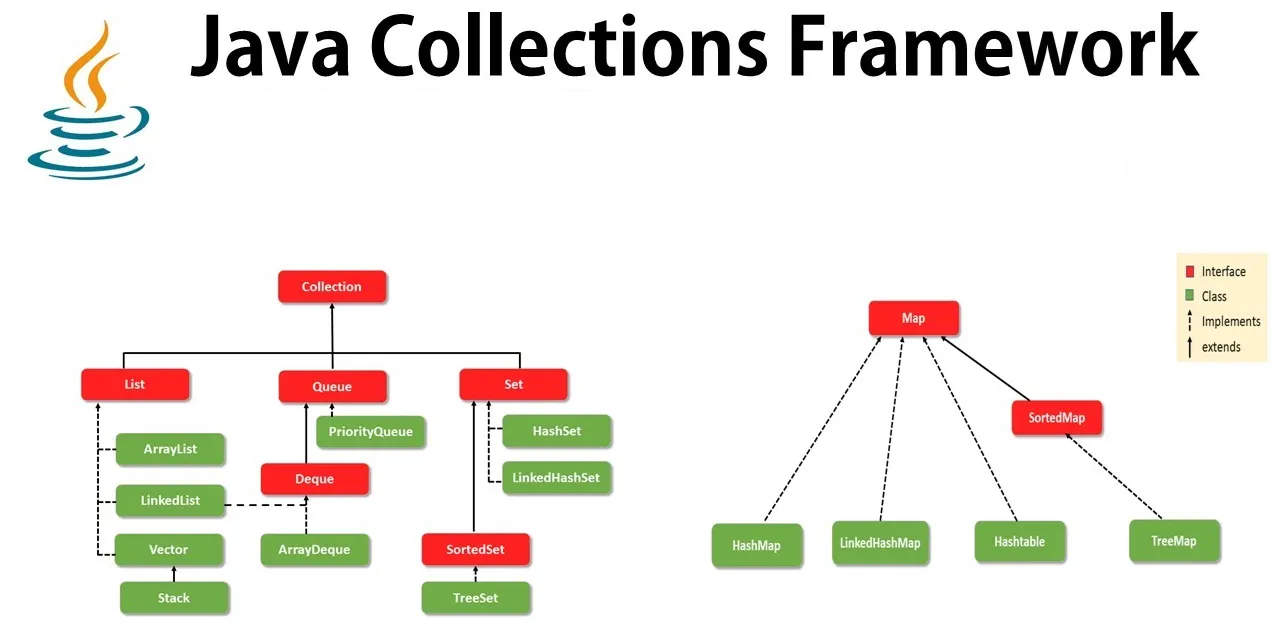
}

}



5 min 11.33 to 11.42

—--------------------------------------------------------------------------------------------------------------



A popular framework that provides an architecture to store and manipulate a group of objects. This framework has been introduced in JDK version 1.2 and contains all classes and interfaces.

There are 2 main root interfaces —

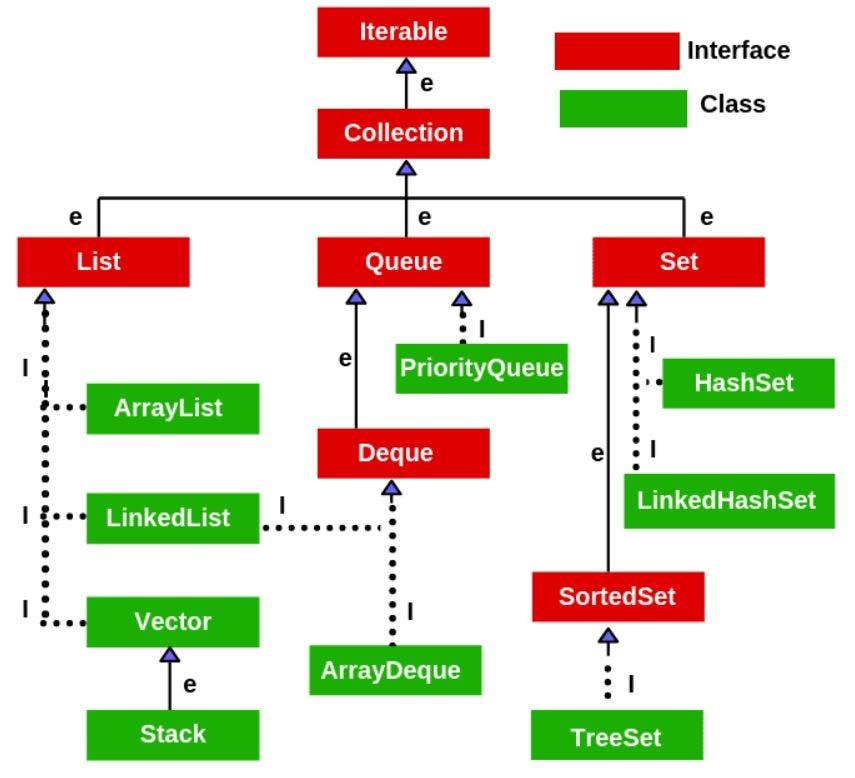
1. The Collection interface ( java.util.Collection )
2. The Map interface ( java.util.Map )

Before the introduction of the Collection framework, developers had to remember different methods, syntaxes, and various constructors present in different classes such as Arrays, Vectors, and Hashtables(classes that provide methods for grouping objects or collections before JDK 1.2). This was almost impossible and resulted in various problems. Developers had to write many different algorithms in order to do work with these early collections classes.

## **Advantages of Collection Framework**

* Reduce Programming effort.
* Increases the speed and quality of the program.
* Improve consistency.

# **Collection Framework Hierarchy**

****

# **Collection Interface**

* This is the root interface for all the collections in the API and it is placed at the top of the collection hierarchy.
* Provides basic operations for adding and removing elements.
* This interface extends the iterable interface, which has 3 methods. They are iterator, spliterator (splitting + iterator), and foreach(Consumer<? super T> action). However, we can only implement the Iterator method. The other 2 methods have default implementations. This iterator method returns an iterator object and that object can be used to iterate over the available elements inside a collection.
* List, queue, and Set interfaces extend the Collection Interface.

==========================================

Collections:

Need for Collections:

**Arrays in Java: A Limiting Approach**

Arrays are one of the most basic data structures in Java, allowing us to store a fixed number of elements of the same type. While arrays are useful in certain cases, they come with significant limitations:

1. **Fixed Size**: Once you declare an array, its size is fixed. You cannot modify the size of the array at runtime.
2. **Inflexibility**: Arrays don’t provide built-in methods for adding, removing, or searching elements. This means that common operations like inserting or deleting elements require manual management.
3. **Inefficient Memory Usage**: Arrays in Java are stored in **stack memory**, which can become inefficient for large amounts of dynamic data.
4. **No Built-in Methods**: Arrays lack built-in functions for tasks like searching, sorting, or modifying elements, requiring developers to manually code these functionalities.

int[] numbers = new int[5]; // Array of fixed size 5

numbers[0] = 10;

numbers[1] = 20;

// You cannot add more than 5 elements or remove any at runtime

numbers[5] = 60; // This will throw an "IndexOutOfBound" error

—---------------------------------------------------------------------------

Array Lists

—---------------------------------------------------------------------

**Java ArrayList** is a part of the **collections framework** and it is a class of java.util package. It provides us with dynamic-sized arrays in Java.

* The main **advantage of ArrayList** is that, unlike normal arrays, we don't need to mention the size when creating ArrayList. It automatically adjusts its capacity as elements are added or removed.
* It may be slower than standard arrays, but it is helpful when the size is not known in advance. Note that creating a large fixed-sized array would cause a waste of space.
* Since ArrayList is part of the collections framework, it has better interoperability with other collections. For example, conversion to a HashSet is straightforward.
* With generics, ArrayList<T> ensures type safety at compile-time.

import java.util.ArrayList;

class Main {

public static void main (String[] args) {

// Creating an ArrayList

ArrayList<Integer> a = new ArrayList<Integer>();

// Adding Element in ArrayList

a.add(1);

a.add(2);

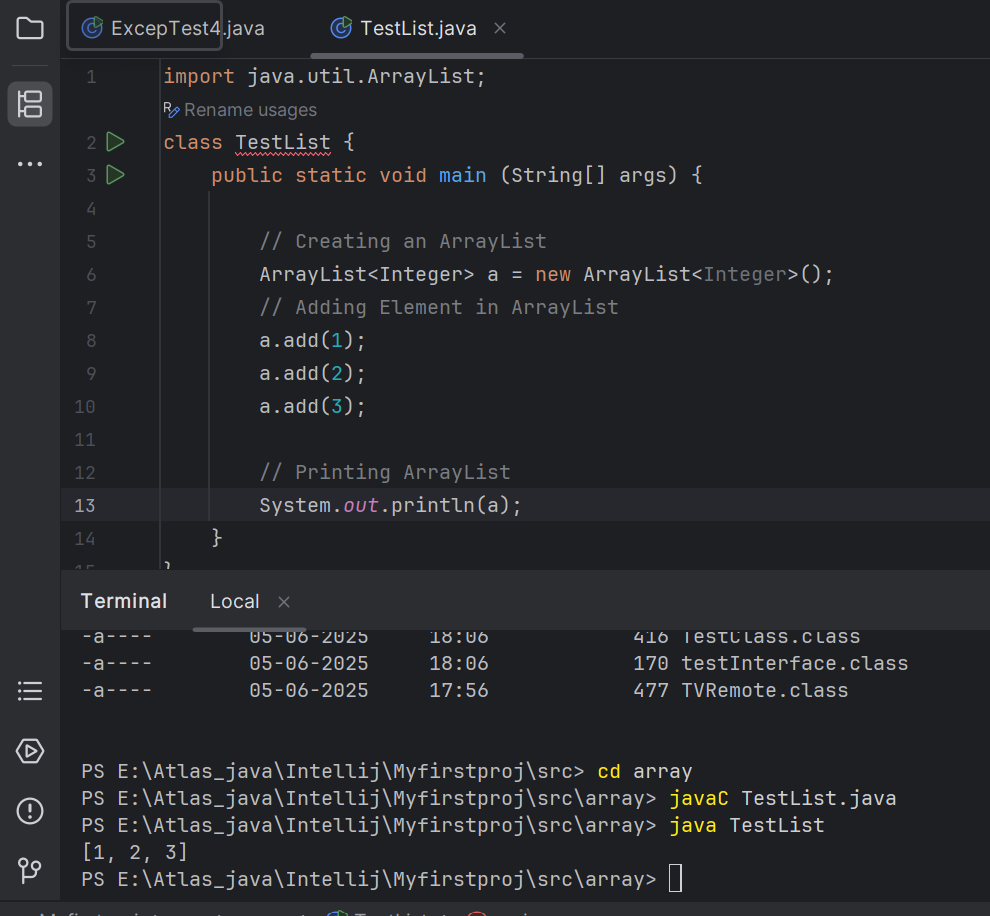
a.add(3);

// Printing ArrayList

System.out.println(a);

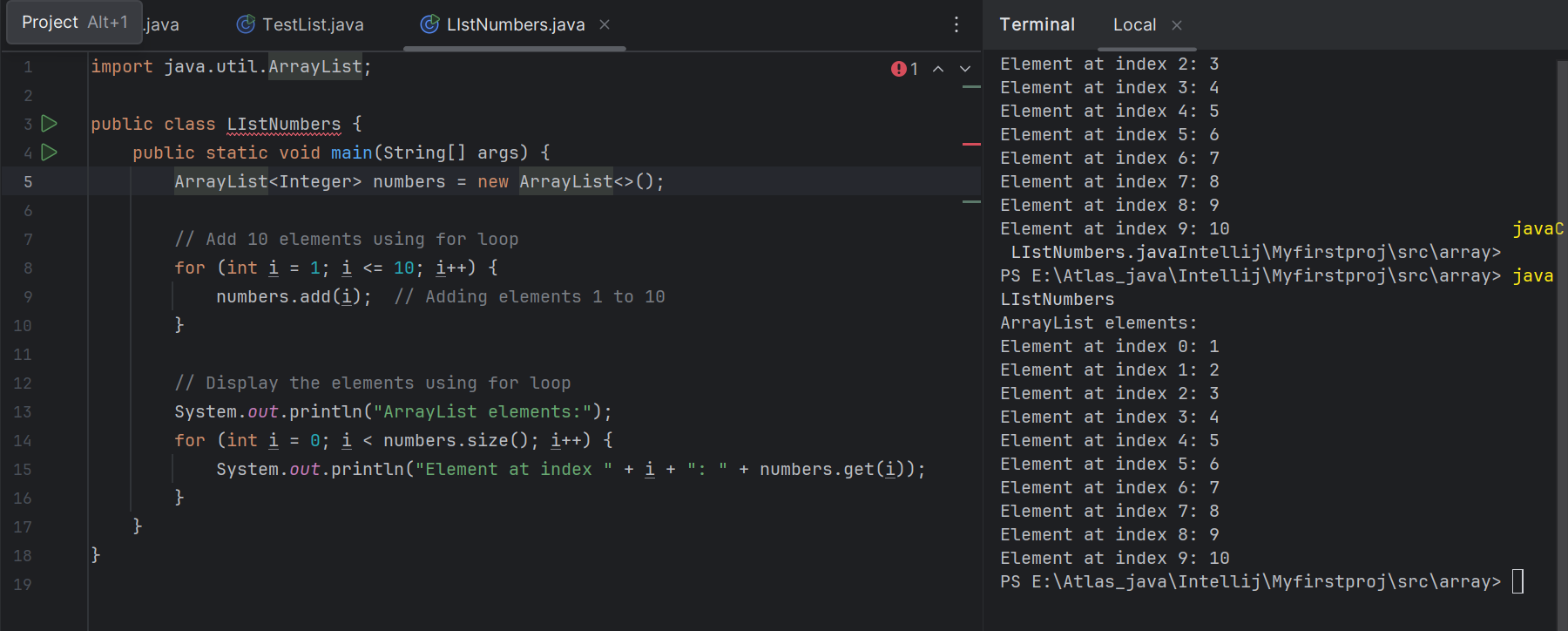
}

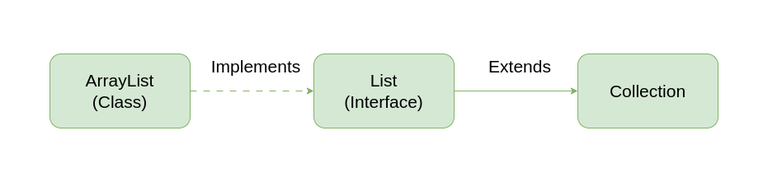
}



Task 011

Wap to create an array list to display 10 elements using for loop.





Note: You can also create a generic ArrayList

Important Features of ArrayList in Java

* ArrayList inherits AbstractList class and implements the List interface.
* ArrayList is initialized by size. However, the size is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
* Java ArrayList allows us to randomly access the list.
* ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.
* ArrayList in Java can be seen as a vector in C++.
* ArrayList is not Synchronized. Its equivalent synchronized class in Java is Vector.

Task 012

Find the output of the be code snippet..

// Addition, Deletion and Updation of Element

import java.util.\*;

class Main {

public static void main(String args[]){

ArrayList<String> al = new ArrayList<>();

al.add("Prasunamba");

al.add("Meher");

System.out.println("Orignal List : "+al);

al.add(1, "Hello");

System.out.println("After Adding element at index 1 : "+ al);

al.remove(0);

System.out.println("Element removed from index 0 : "+ al);

al.remove("Prasunamba");

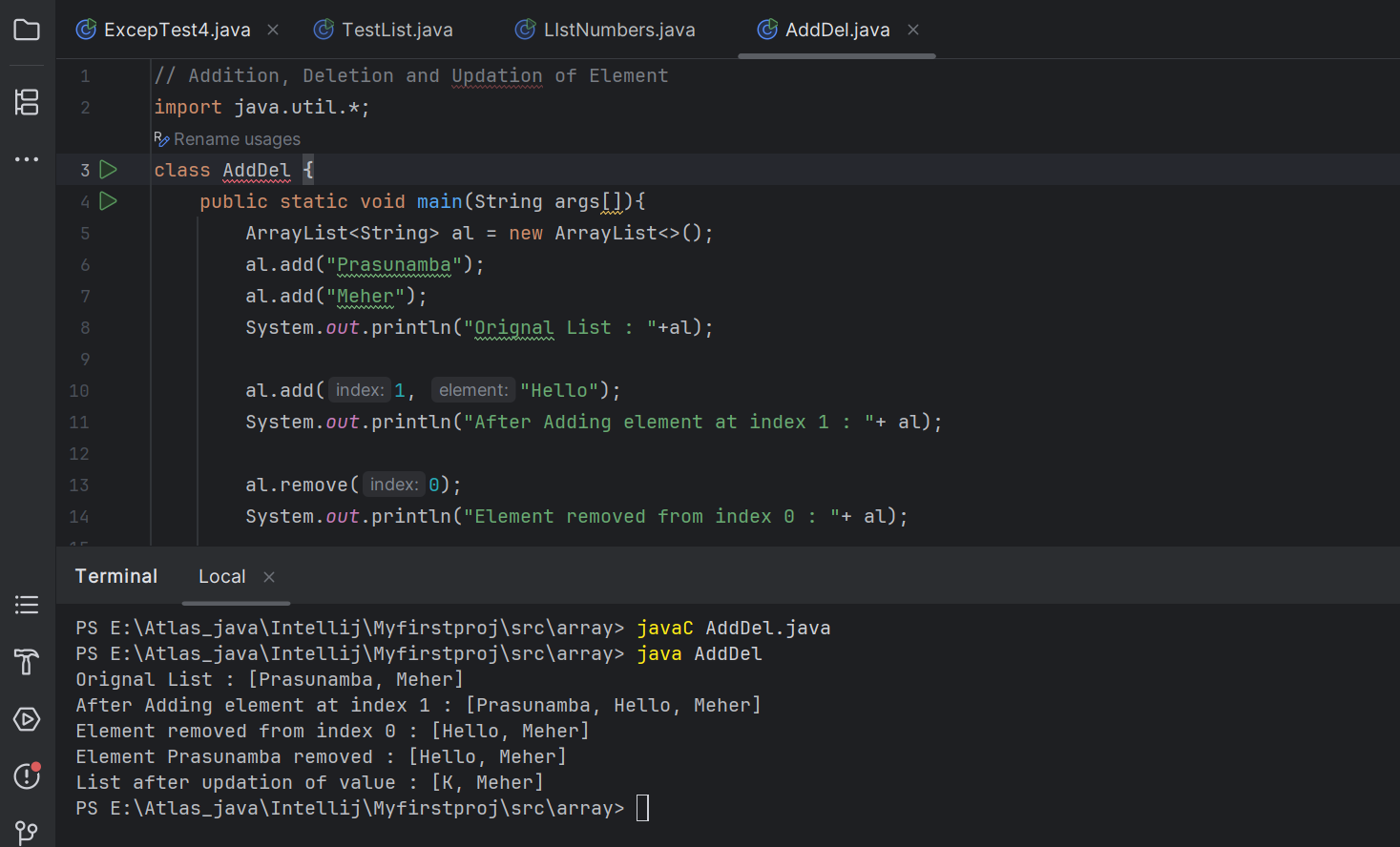
System.out.println("Element Prasunamba removed : "+ al);

al.set(0, "K");

System.out.println("List after updation of value : "+al);

}

}



==========================================

Things to Know:

**Java ArrayList Methods**

| **Method** | **Description** |
| --- | --- |
| add(int index, Object element) | This method is used to insert a specific element at a specific position index in a list. |
| add(Object o) | This method is used to append a specific element to the end of a list. |
| addAll(Collection C) | This method is used to append all the elements from a specific collection to the end of the mentioned list, in such an order that the values are returned by the specified collection’s iterator. |
| addAll(int index, Collection C) | Used to insert all of the elements starting at the specified position from a specific collection into the mentioned list. |
| clear() | This method is used to remove all the elements from any list. |
| clone() | This method is used to return a shallow copy of an ArrayList in Java. |
| contains(Object o) | Returns true if this list contains the specified element. |
| ensureCapacity(int minCapacity) | Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. |
| forEach(Consumer<? super E> action) | Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception. |
| get(int index) | Returns the element at the specified position in this list. |
| indexOf(Object O) | The index the first occurrence of a specific element is either returned or -1 in case the element is not in the list. |
| isEmpty() | Returns true if this list contains no elements. |
| lastIndexOf(Object O) | The index of the last occurrence of a specific element is either returned or -1 in case the element is not in the list. |
| listIterator() | Returns a list iterator over the elements in this list (in proper sequence). |
| listIterator(int index) | Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list. |
| remove(int index) | Removes the element at the specified position in this list. |
| remove(Object o) | Removes the first occurrence of the specified element from this list, if it is present. |
| removeAll(Collection c) | Removes from this list all of its elements that are contained in the specified collection. |
| removeIf(Predicate filter) | Removes all of the elements of this collection that satisfy the given predicate. |
| removeRange(int fromIndex, int toIndex) | Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. |
| retainAll(Collection<?> c) | Retains only the elements in this list that are contained in the specified collection. |
| set(int index, E element) | Replaces the element at the specified position in this list with the specified element. |
| size() | Returns the number of elements in this list. |
| spliterator?() | Creates a late-binding and fail-fast Spliterator over the elements in this list. |
| subList(int fromIndex, int toIndex) | Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. |
| toArray() | This method is used to return an array containing all of the elements in the list in the correct order. |
| toArray(Object[] O) | It is also used to return an array containing all of the elements in this list in the correct order same as the previous method. |
| trimToSize() | This method is used to trim the capacity of the instance of the ArrayList to the list's current size. |

**Some Key Points of ArrayList in Java**

1. ArrayList is Underlined data Structure Resizable Array or Growable Array.
2. ArrayList Duplicates Are Allowed.
3. Insertion Order is Preserved.
4. Heterogeneous objects are allowed.
5. Null insertion is possible.

**Complexity of Java ArrayList**

| **Operation** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| **Inserting Element in ArrayList** | O(1) | O(N) |
| **Removing Element from ArrayList** | O(N) | O(1) |
| **Traversing Elements in ArrayList** | O(N) | O(N) |
| **Replacing Elements in ArrayList** | O(1) | O(1) |

Below are the advantages and disadvantages of using ArrayList in Java:

**Advantages of Java ArrayList**

* **Dynamic size:** ArrayList can dynamically grow and shrink in size, making it easy to add or remove elements as needed.
* **Easy to use**: ArrayList is simple to use, making it a popular choice for many Java developers.
* **Fast access**: ArrayList provides fast access to elements, as it is implemented as an array under the hood.
* **Ordered collection**: ArrayList preserves the order of elements, allowing you to access elements in the order they were added.
* **Supports null values**: ArrayList can store null values, making it useful in cases where the absence of a value needs to be represented.

**Disadvantages of Java ArrayList**

* **Slower than arrays**: ArrayList is slower than arrays for certain operations, such as inserting elements in the middle of the list.
* **Increased memory usage**: ArrayList requires more memory than arrays, as it needs to maintain its dynamic size and handle resizing.
* **Not thread-safe:** ArrayList is not thread-safe, meaning that multiple threads may access and modify the list concurrently, leading to potential race conditions and data corruption.
* **Performance degradation**: ArrayList's performance may degrade as the number of elements in the list increases, especially for operations such as searching for elements or inserting elements in the middle of the list.

Task 13:

User defined Exception:

// A Class that represents user-defined exception

class Customer extends Exception {// predefined class Exception

public Customer(String m) { // constructor with parameters

super(m); // parent class constructor

}

}

// A Class that uses the above MyException

public class setText {

public static void main(String args[]) {

try {

// Throw an object of user-defined exception

throw new MyException("This is a custom exception");

}

catch (MyException ex) {

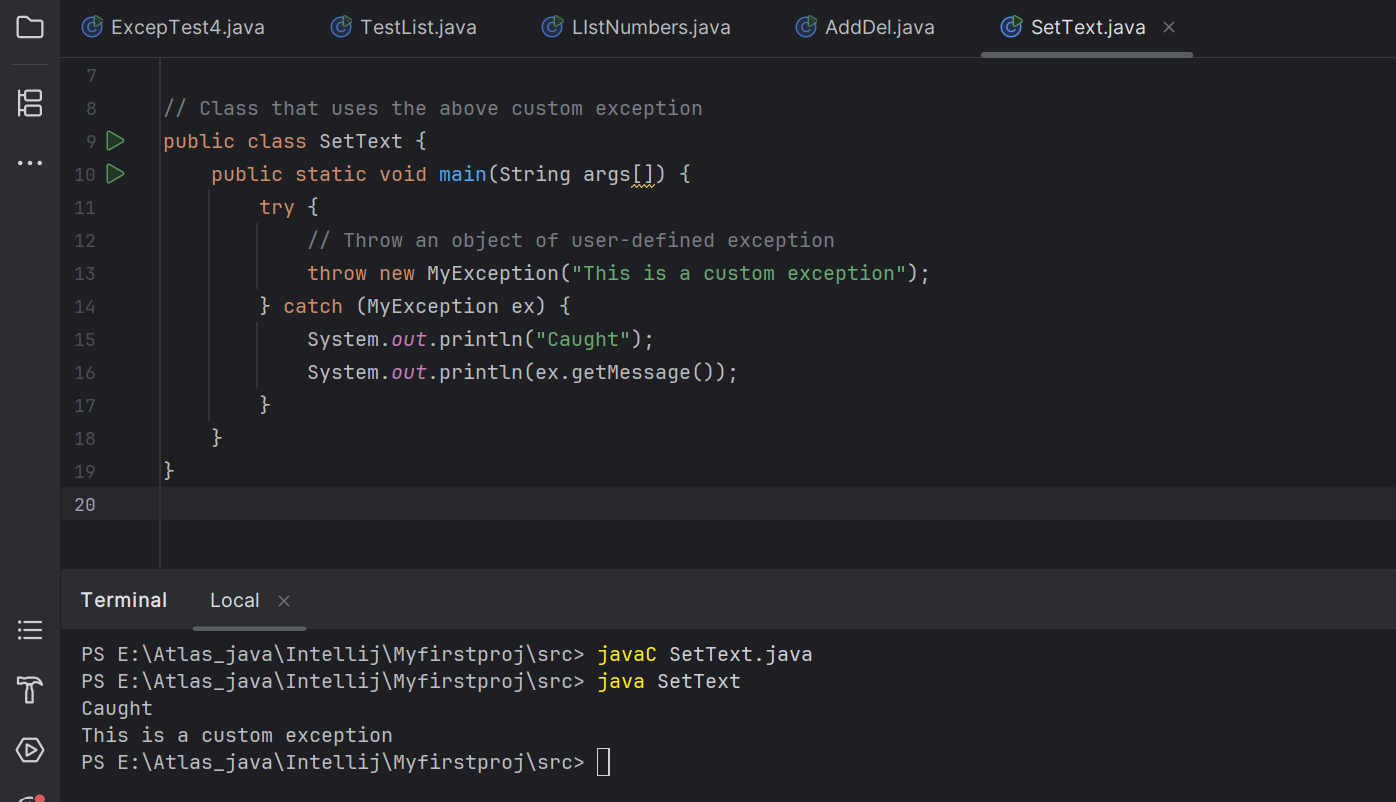
System.out.println("Caught");

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

}

}

}



What custom defined exceptions?

* To represent application-specific errors.
* To add clear, descriptive error messages for better debugging.
* To encapsulate business logic errors in a meaningful way.

Task 014

Inheritance

Classes customer/ person , employee, Manager … 2 variables in each class

Driver class – display all the variables… toString();

Hint : use getter and setters..

// Base class: Customer

class Customer {

private String name;

private int customerId;

// Setters

public void setName(String name) {

this.name = name;

}

public void setCustomerId(int id) {

this.customerId = id;

}

// Getters

public String getName() {

return name;

}

public int getCustomerId() {

return customerId;

}

// toString method

public String toString() {

return "Customer Name: " + name + ", Customer ID: " + customerId;

}

}

// Employee class inherits Customer

class Employee extends Customer {

private int employeeId;

private String department;

// Setters

public void setEmployeeId(int id) {

this.employeeId = id;

}

public void setDepartment(String dept) {

this.department = dept;

}

// Getters

public int getEmployeeId() {

return employeeId;

}

public String getDepartment() {

return department;

}

// toString method

public String toString() {

return super.toString() +

"\nEmployee ID: " + employeeId + ", Department: " + department;

}

}

// Manager class inherits Employee

class Manager extends Employee {

private String project;

private int teamSize;

// Setters

public void setProject(String project) {

this.project = project;

}

public void setTeamSize(int size) {

this.teamSize = size;

}

// Getters

public String getProject() {

return project;

}

public int getTeamSize() {

return teamSize;

}

// toString method

public String toString() {

return super.toString() +

"\nProject: " + project + ", Team Size: " + teamSize;

}

}

// Driver class with main method

public class InheritanceTest {

public static void main(String[] args) {

// Create a Manager object

Manager m = new Manager();

// Set values using setters

m.setName("Alice");

m.setCustomerId(1001);

m.setEmployeeId(2002);

m.setDepartment("Technology");

m.setProject("AI System");

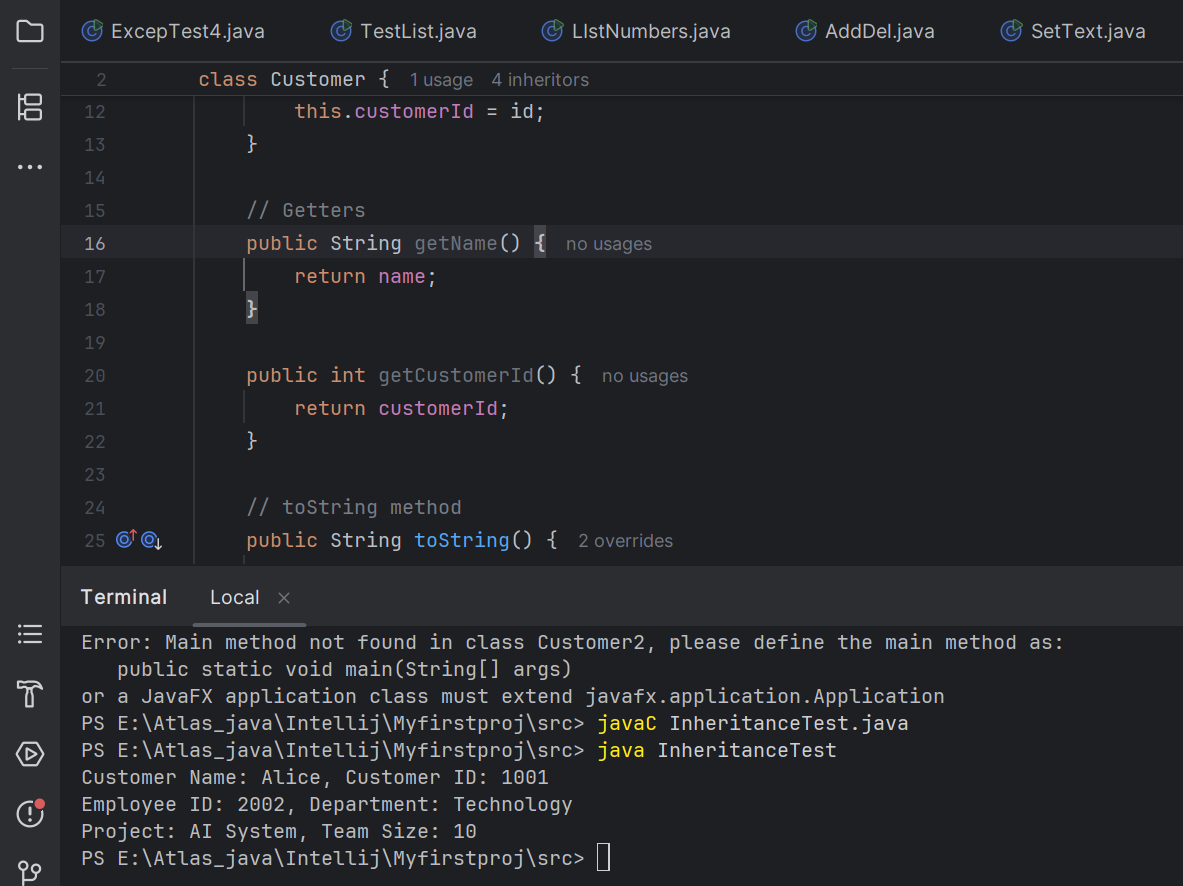
m.setTeamSize(10);

// Display all details

System.out.println(m.toString());

}

}



Inner classes Task 015:

class OuterClass {

int x = 10;

class InnerClass {

int y = 5;

}

}

public class Main1 {

public static void main(String[] args) {

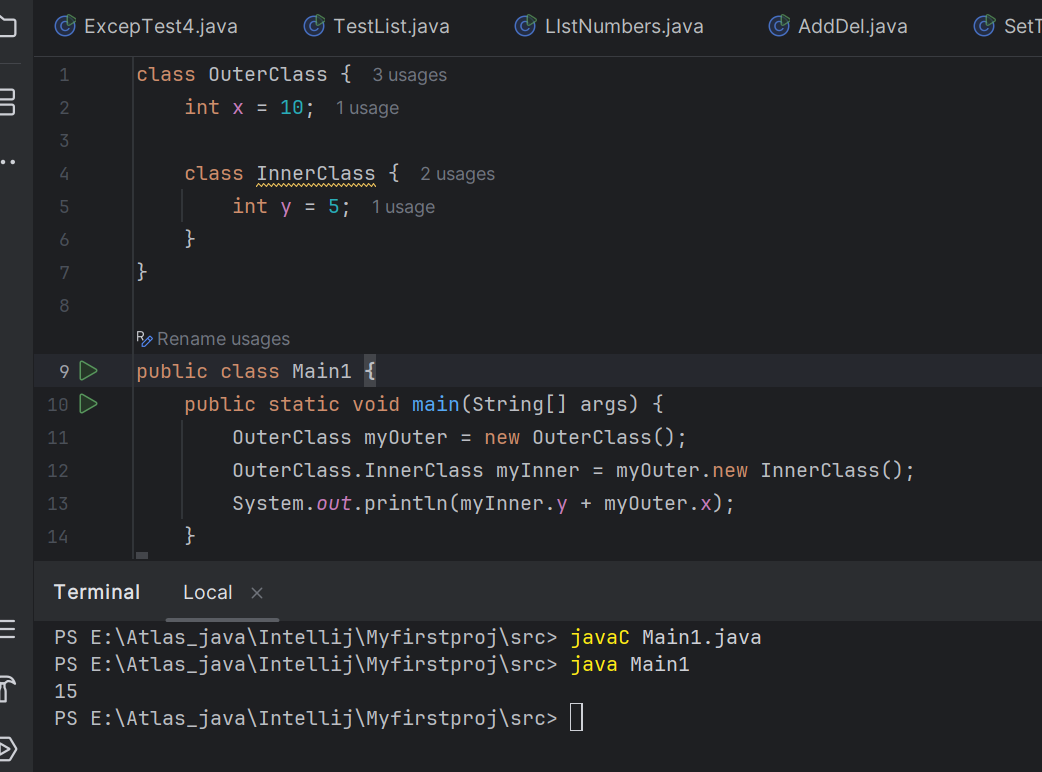
OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.y + myOuter.x);

}

}



Task 016

Use the above code and make the inner class as private and see the output..

Ex: private class InnerClass {

class OuterClass {

int x = 10;

private class InnerClass {

int y = 5;

int getSum() {

return x + y; // Accessing outer class variable directly

}

}

// Method to access the private inner class

public void displaySum() {

InnerClass myInner = new InnerClass();

System.out.println("Sum = " + myInner.getSum());

}

}

public class Main {

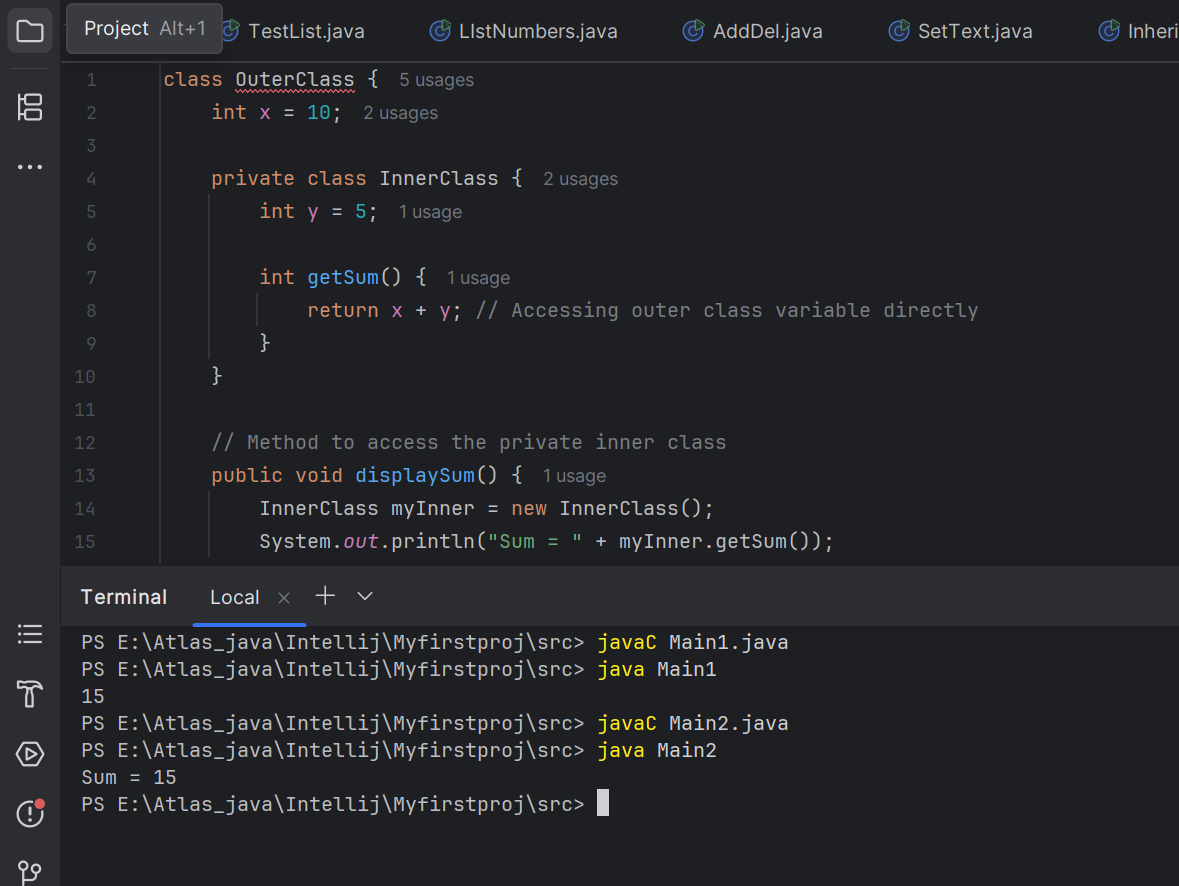
public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

myOuter.displaySum(); // This works!

}

}



8 min 17.20 to 17.28

Task 017

Use the above code Task 015 and make the inner class static … see the output and explain..

Ex: static class InnerClass {



class OuterClass{

Int x = 50;

Class InnerClass {

Public int innerMethod() {

Return x;

}

}

}

Public class DriverClass {

psvm(){

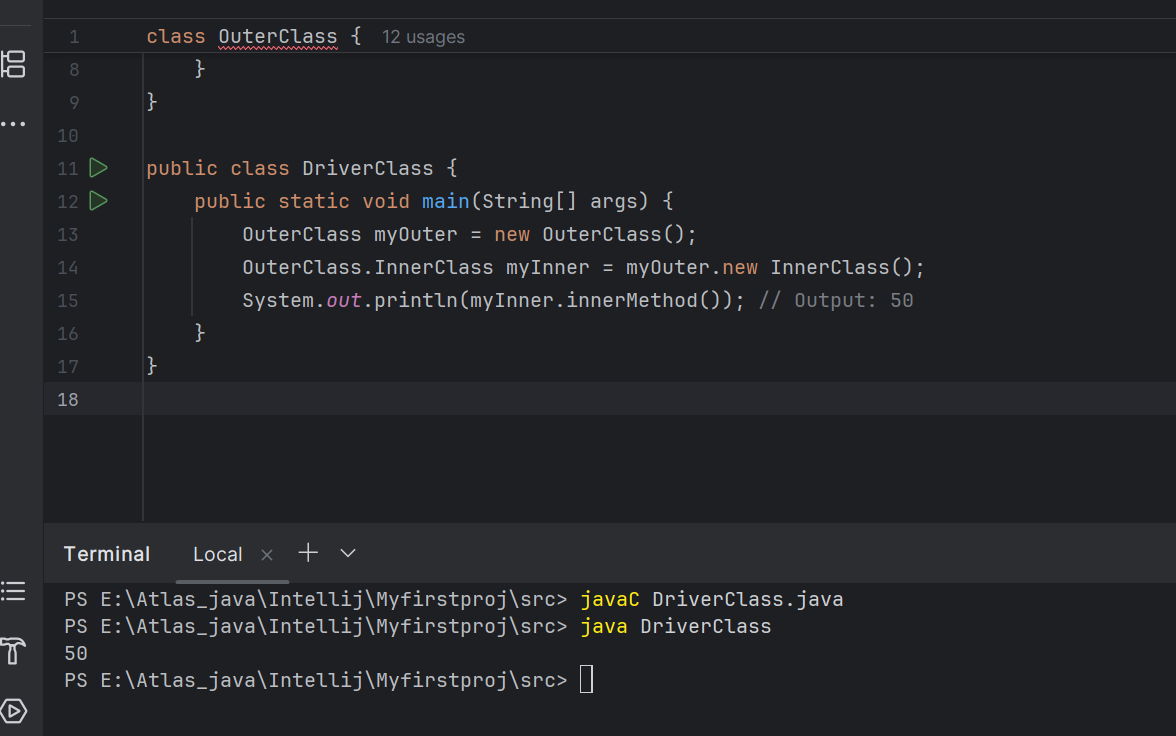
OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.innerMethod());

}

}



==========================================

# **List Interface**

* Responsible for declaring the behavior of collections which stores a sequence of elements. A list may contain duplicate elements and the order is retained in which we add elements. That means we can retrieve elements according to this order.
* There are 3 main subclasses that implement the list interface.

1. **ArrayList**
2. **Vector**
3. **LinkedList**

* The list defines some of its own methods as follows.

1. **Void add(int index, Object obj):** Insert Object into the specified index.
2. **boolean addAll(int index, Collection c):** Insert all elements of Collection c into the location of the specified index.
3. **Object get(int index):** Return object stored at the specified index.
4. **int indexOf(Object obj):** Return the index of the first instance of the object.
5. **int lastIndexOf(Object obj):** Return the index of the last instance of the object.
6. **ListIterator listIterator():** Returns an iterator.
7. **ListInterator list iterator(int index):** Returns an iterator that will invoke a list that begins at a specified index.
8. **Object remove(int index):** Remove an element at the specified index.
9. **Object set(int index, object obj):** Assign object to the location specified by the index.
10. **List subList(int start, int end):** Returns list of elements from specified start to end.

# **Set Interface**

Set is a special type of collection where we cannot store duplicate elements. This interface once contains the methods inherited from the Collection interface. This is an unordered collection of objects. Mathematical operations such as intersection, union, and difference are also supported in Set interface.

There are 2 main subclasses that implement the Set interface as it is.

1. HashSet
2. LinkedHashSet

**TreeSet**

This is implemented using the SortedSet interface. SortedSet is another special class that extends the Set interface. TreeSet uses a tree data structure to store the data.

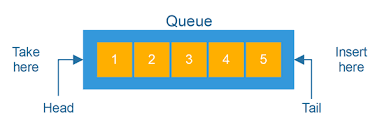
**SortedSet** — *Behaves like a simple set with the exception that it stores elements in a sorted format.*

Available Methods,

1. **add(element):** *All specific element to the set.*
2. **addAll(collection) :** *Append all elements from the provided collection to the set.*
3. **clear() :** *Remove all elements from the set.*
4. **contains(element) :** *Check for a specific element in the set.*
5. **containsAll(collection) :** *Check whether the set contains all elements present in the given set.*
6. **hashCode() :** *Get the hashcode value.*
7. **isEmpty() :** *Check whether the set is empty.*
8. **iterator() :** *Return the iterator of the set.*
9. **remove(element) :** *Remove given element from the set.*
10. **removeAll(collection) :** *Remove all elements from a collection*
11. **retainAll(collection) :** *Retain all elements from a set.*
12. **size() :** *Get the size of the set.*
13. **toArray() :** *Form an array using the elements in the set.*

# **Queue Interface**

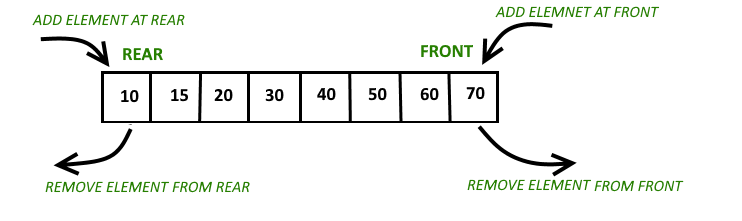
Holds elements that are going to be processed in a special order called FIFO(First In First Out). In queues, we can insert elements from one end of the list and take out elements from the other end.



Queue Interface is implemented by a single class known as the PriorityQueue. However, this interface is extended by another interface call Deque which implemented the ArrayDeque class.

# **Deque**

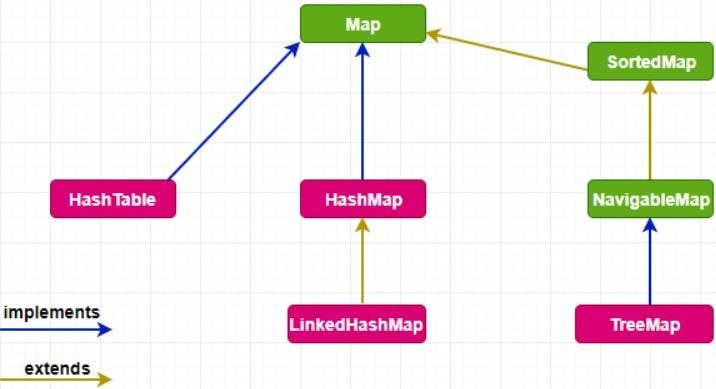
Dequeue stands for “double-ended queue”.In a double-ended queue, we can add or remove elements from either end of the queue. This increases the versatility of the queue. We can use the deque as a normal queue that follows FIFO. But most of the time deques are used to act as a stack that follows LIFO(Last In First Out). ArrayQue class implements the Deque interface.



# **Methods of Queue Interface**

* **add(element):** *Add elements to the end of the queue.*
* **element():** *View the head of the element. ( Throw NoSuchElementException when the queue is empty)*
* **offer(element):** *Insert an element into the queue. Returns false when the container is full.*
* **peek():** *View the head (top element) of the queue.*
* **poll():** *Removes and returns the head of the queue.*
* **remove():** *Removes and returns the head of the queue(Throw NoSuchElementException when the queue is empty).*

# **Map Interface**

****

Map interface behaves differently from the rest of the collection types. Map stores its elements as key/value pairs. Each object is mapped to a particular key. There cannot be any duplicate keys in Map. But we can have duplicate values.

**The Map is ideal for searching, updating, and deleting records based on a key.**

==========================================================

Interfaces : just like classes // blueprint.. / template

Java does not – multiple inheritance.. — in herfaces

Interface interfaceNmae{

method1();

method2();

}

Parent class → child class === > extends

Interface ti interface or class ====> implements..

Class ClassName extends PrentClass implements Interface1, interface2{

method1() {

}

}

Task 010

—-----------------------------------------------------------------

Custom exception —

Abstraction with classes done … with interfaces

Interfaces not done..