# **Day 9 - 13 June 2025**

**Document Name:**Day 9 - hmuvvala@ - Hari Gopal Muvvala

### **Task 001**

**Program:**

public class Task001 {

public static void main(String[] args) {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

}

}

**Output:**

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 3

**Explanation:**

Trying to access index 10 in an array of size 3 causes an ArrayIndexOutOfBoundsException. This is an unchecked runtime exception.

**Understanding:**

This task taught me what happens when we try to access an invalid index in an array. Java throws a runtime exception and stops program execution.

### **Task 002**

**Program:**

public class Task002 {

public static void main(String[] args) {

try {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

} catch (Exception ex) {

System.out.println("Something went wrong: " + ex.getMessage());

}

}

}

**Output:**

Something went wrong: Index 10 out of bounds for length 3

**Explanation:**

The try-catch block prevents the program from crashing. The exception is caught and a user-friendly message is displayed.

**Understanding:**

I learned how to use a try-catch block to catch exceptions and handle errors smoothly without crashing the program.

### **Task 003**

**Program:**

public class Task003 {

public static void main(String[] args) {

try {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

} catch (Exception e) {

System.out.println("Something went wrong.");

} finally {

System.out.println("I'm from finally block.");

}

}

}

**Output:**

Something went wrong.

I'm from finally block.

**Explanation:**

The finally block executes whether an exception is caught or not. It is used for cleanup actions like closing files, releasing resources, etc.

**Understanding:**

This task helped me understand that the finally block always runs, making it useful for final steps even if an error occurs.

### **Task 007**

**Program:**

public class Task007 {

public static void main(String[] args) {

throw new ArithmeticException("Access denied - cannot divide by zero");

}

}

**Output:**

Exception in thread "main" java.lang.ArithmeticException: Access denied - cannot divide by zero

**Explanation:**In this task, we are not waiting for Java to throw an error. We are doing it ourselves using the throw keyword. This is how we can deliberately stop the program and give a custom error message when we detect something wrong.

**My Understanding:**I realized that I can tell Java to "stop the program and show this message" when I detect something invalid — like illegal input or wrong data. It's like telling someone “Stop right there! You're not allowed in here.”

### **Task 008**

**Program:**

public class Task008 {

public static void main(String[] args) {

int age = 15;

if (age < 18) {

throw new ArithmeticException("Access denied - must be at least 18 years old.");

} else {

System.out.println("Access granted - You are old enough!");

}

}

}

**Output:**

Exception in thread "main" java.lang.ArithmeticException: Access denied - must be at least 18 years old.

**Explanation:**We're using an if-condition to check the user's age. If it's less than 18, we throw an exception manually. Otherwise, we show a success message. This simulates real-world checks like login validations or form inputs.

**My Understanding:**I understood that throwing an exception conditionally helps enforce rules. If someone doesn't meet my condition (like age), I can stop the program myself — instead of waiting for Java to find the error.

### **Task 009**

**Program:**

import java.util.ArrayList;

public class Task009 {

public static void main(String[] args) {

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

a.add(1);

a.add(2);

a.add(3);

System.out.println(a);

}

}

**Output:**

[1, 2, 3]

**Explanation:**This task introduces ArrayList, a flexible version of arrays. You can add elements without worrying about fixed size. The add() method inserts elements, and System.out.println() prints the entire list.

**My Understanding:**It’s like using a list that can grow automatically. I don’t need to say “this list will have 10 items.” I just keep adding, and it expands. It’s very useful compared to normal arrays that are fixed in size.

### **Task 010**

**Program:**

import java.util.ArrayList;

public class Task010 {

public static void main(String[] args) {

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

al.add("Hari");

al.add("Gopal");

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

al.add(1, "Muvvala");

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

al.remove(0);

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

al.remove("Hari");

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

al.set(0, "HGM");

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

System.out.println("Printing elements one by one:");

for (int i = 0; i < al.size(); i++) {

System.out.println(al.get(i));

}

}

}

**Output:**

Original List: [Hari, Gopal]

After Adding element at index 1: [Hari, Muvvala, Gopal]

Element removed from index 0: [Muvvala, Gopal]

Element Hari removed: [Muvvala, Gopal]

List after updation of value: [HGM, Gopal]

Printing elements one by one:

HGM

Gopal

**Explanation:**We used ArrayList to perform operations like add, insert at a position, remove by index or value, update using set(), and loop through using a for loop.

**My Understanding:**This task helped me get hands-on with common operations in ArrayList. It’s like managing a dynamic shopping list — I can add, insert, remove, or change any item whenever I want.

### **Task 011**

**Program:**

public class Task011 {

int x = 10;

class InnerClass {

int y = 5;

}

public static void main(String[] args) {

Task011 myOuter = new Task011();

Task011.InnerClass myInner = myOuter.new InnerClass();

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

}

}

**Output:**

15

**Explanation:**This task introduces inner classes. We created an object of the outer class, and using that, we created an object of the inner class. We then accessed both outer and inner class variables.

**My Understanding:**I learned how to use inner classes. It’s like having a small machine (inner class) inside a big machine (outer class), and we need to start the big one first before using the small one.

### **Task 012**

**Program:**

public class Task012 {

int x = 10;

private class InnerClass {

int y = 5;

}

public static void main(String[] args) {

Task012 myOuter = new Task012();

Task012.InnerClass myInner = myOuter.new InnerClass();

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

}

}

**Output:**

15

**Explanation:**The inner class is now private, but it’s still accessible from the main() method because it’s inside the same outer class. So it works without error.

**My Understanding:**I thought private means we can’t access it at all, but here I learned that within the same class, even private inner classes can be accessed. It's like having a private room in your own house — you still have the key!

### **Task 013**

**Program:**

public class Task013 {

int x = 10;

static class InnerClass {

int y = 5;

}

public static void main(String[] args) {

Task013.InnerClass myInner = new Task013.InnerClass();

Task013 myOuter = new Task013();

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

}

}

**Output:**

15

**Explanation:**Here, the inner class is declared as static, so we can create its object without needing an outer class object first. However, since x is a non-static member of the outer class, we still need to create an object of the outer class to access it.

**My Understanding:**A static inner class is more independent — it doesn’t need the outer object. But it can’t directly access non-static variables like x. It’s like a separate guest in the house who can’t freely access the private items of the owner without permission.

### **Task 014**

**Program:**

public class Task014 {

int x = 50;

class InnerClass {

public int innerMethod() {

return x;

}

}

public static void main(String[] args) {

Task014 myOuter = new Task014();

Task014.InnerClass myInner = myOuter.new InnerClass();

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

}

}

**Output:**

50

**Explanation:**We created a method inside the inner class that accesses a variable from the outer class. This works because inner classes have access to all members (even private) of the outer class.

**My Understanding:**This task helped me realize that inner classes can use outer class variables directly. It’s like an inner room having access to everything in the outer house.

### **Task 015**

**Program:**

public class Task015 {

int x = 10;

static class InnerClass {

static int y = 5;

}

public static void main(String[] args) {

Task015.InnerClass myInner = new Task015.InnerClass();

System.out.println(myInner.y);

}

}

**Output:**

5

**Explanation:**Both the inner class and its variable y are static. This means we don’t need to create an object of the outer class at all. We can directly use InnerClass.y.

**My Understanding:**I learned that when both the class and variable are static, they behave like global tools — no need to carry any object with you. It’s like accessing public utilities — no keys, no restrictions.

### **Task 016**

**Program Summary:**

Created class hierarchy using inheritance:

* Person → parent
* Employee → extends Person
* Manager → extends Employee

Used getters and toString() to display all variables from a Manager object.

**My Understanding:**

Learned how child classes inherit parent properties using extends and how to override toString() to print class data neatly.

### **Task 017**

**Topic:** Java 8 Features (Overview)

**Key Points:**

* Lambda Expressions
* Stream API
* forEach()
* Default/Static methods in Interfaces
* Functional Interfaces
* Method References
* New Date-Time API

**My Understanding:**

Java 8 simplified coding using lambdas and made collections more powerful with streams and forEach. Interfaces also became smarter.

### **Task 018**

**Program Summary:**

Used forEach() three ways:

1. Iterator
2. Anonymous Consumer class
3. Custom class implementing Consumer

**My Understanding:**

Learned how Consumer works with forEach and how Java 8 encourages writing cleaner loops.

### **Task 019**

**Topic:** DigitalOcean Java 8 Article

**Action:** Referred to [DigitalOcean Java 8 Tutorial](https://www.digitalocean.com/community/tutorials/java-8-features-with-examples)

**My Understanding:**

Understood Java 8 core features with examples, like Optional, Streams, and method references. Good resource for real usage.

### **Task 020**

**Topic:** Java Collection Interfaces

**Summary:**

* ArrayList: Ordered, duplicates allowed, index-based access
* LinkedList: Same as ArrayList but node-based
* HashSet: Unordered, no duplicates
* HashMap: Key-value pairs, keys unique

**My Understanding:**

These four collection types cover most real-world data handling in Java. List for order, Set for uniqueness, Map for key-value storage.

### **✅ Personal Summary – What I Learned Today**

* I started with exception handling: try, catch, finally, throw, and throws. I understood how exceptions stop program flow and how we can handle them.
* I practiced custom exceptions by creating a class that extends Exception and throwing it manually.
* I moved on to Java 8 features like lambda expressions, forEach(), Consumer, and method references.
* I explored different ways to loop through collections and process elements cleanly.
* I reviewed the basics and differences between ArrayList, LinkedList, HashSet, and HashMap.
* I built inheritance chains using Person, Employee, and Manager, and displayed all their properties using toString().
* I learned that inner classes can be non-static, static, or private, and how each behaves differently in terms of accessibility.

### **🔁 Simple Reminders for Myself:**

* Use try-catch to handle exceptions and avoid crashes
* finally always runs, useful for cleanup
* Use throw to stop execution manually when needed
* ArrayList is good for order and indexing
* HashSet ensures uniqueness
* HashMap is useful for key-value pairs
* Inner classes need outer class object (unless static)
* forEach is cleaner than regular loops when using Java 8