DAY 3

Introduction to Arrays:

What is an Array?

An array is a collection of elements that are:

- Of the same data type (like int, float, String, etc.)
- Stored in continuous memory locations
- Accessed using indexes

Why use Arrays?

Instead of creating separate variables for each value:

```
int a = 10;

int b = 20;

int c = 30;
```

We can use one array:

```
int[] numbers = {10, 20, 30};
```

Key Points:

- Array size is fixed (we decide it when we create the array).
- Indexing starts at 0.
- Elements are accessed by their index.

Example: numbers[0] gives 10.

Types of Arrays:

One-Dimensional Array (1D)

- A 1D array is a simple list of elements stored in a row.
- Example:

```
int[] arr = {1, 2, 3, 4};
System.out.println(arr[2]); // Output: 3
```

  Here:

  arr[0] → 1

  arr[1] → 2

  arr[2] → 3

  arr[3] → 4

Two-Dimensional Array (2D)

- A 2D array is like a table with rows and columns.
- Example:

```
int[][] matrix = {
  {1, 2},
  {3, 4}
};
System.out.println(matrix[0][1]); // Output: 2
```

  This means:

  matrix[0][0] → 1

  matrix[0][1] → 2

matrix[1][0] → 3
matrix[1][1] → 4

Multi-Dimensional Array (3D and more)

- These arrays go beyond 2D—used in complex applications like games, simulations, etc.
- Example:
  int[][][] cube = new int[2][2][2];
  Here, cube is a 3D array with:
  2 layers
  Each layer has 2 rows
  Each row has 2 columns
  We can access elements like:
  cube[0][1][1] = 5;

Jagged Array:

What is a Jagged Array?

A jagged array is an array of arrays, but:

- Each sub-array can have a different size
- More flexible than regular 2D arrays
- Example:
  int[][] jagged = new int[3][];
  jagged[0] = new int[2]; // 2 elements
  jagged[1] = new int[4]; // 4 elements
  jagged[2] = new int[1]; // 1 element

So:
jagged[0] has 2 elements
jagged[1] has 4 elements
jagged[2] has 1 element

Use Case:

- Jagged arrays are useful when:
- We don’t know the exact number of columns for each row
- Example: Storing marks of students who have different number of subjects

Compiler in Java:

What is a Compiler?

- A compiler is a tool that converts your Java code (written in .java files) into bytecode (stored in .class files).
- Bytecode is not machine code. It is a special code that the JVM understands, not the operating system directly.

Why Bytecode?

- Bytecode allows Java to be platform-independent.
- You can write your code once and run it anywhere (on Windows, Linux, Mac) as long as JVM is installed.
- Example:

```
// File: Hello.java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

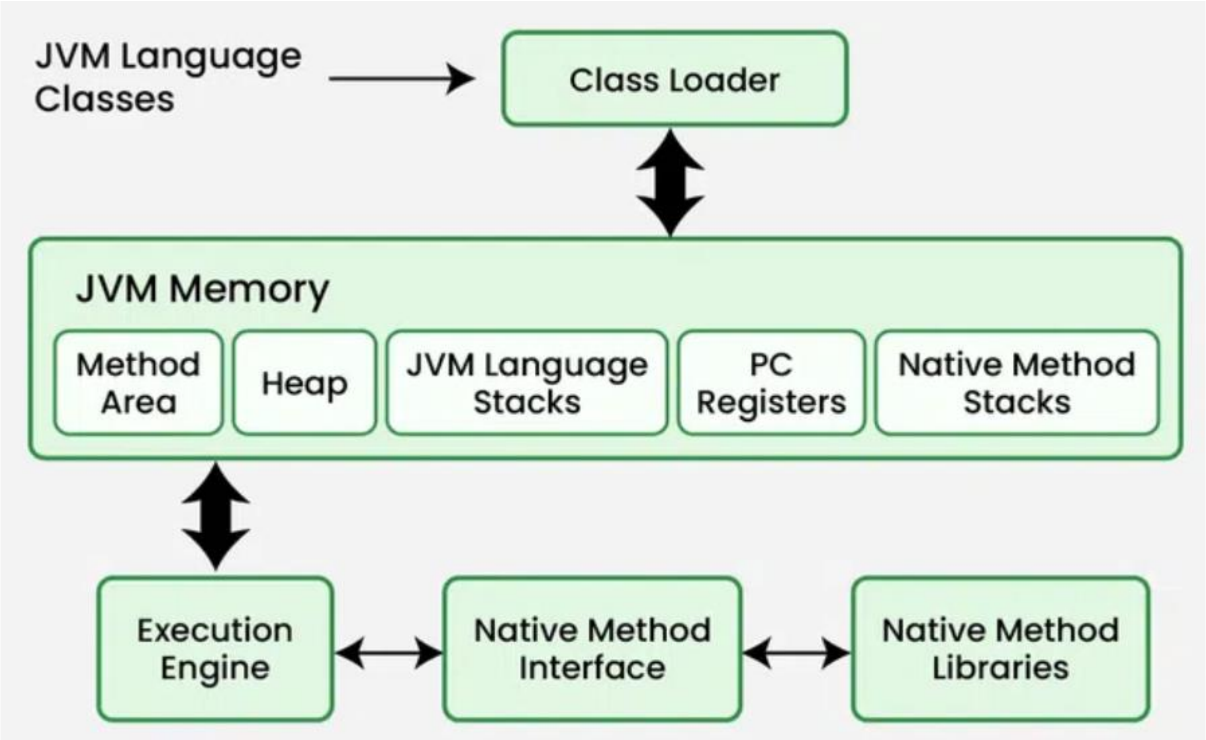
When you compile this:

```
javac Hello.java
```

It creates:

Hello.class ← This is bytecode

JVM (Java Virtual Machine):



What is JVM?

- JVM stands for Java Virtual Machine.

- It runs the bytecode (.class file) and converts it into machine code that your system understands.
- It makes Java platform-independent.

Main Components of JVM:

Component	Function
Class Loader	Loads .class files (bytecode) into memory.
Runtime Memory	Stores variables, objects, method calls, etc. during program execution.
Execution Engine	Actually executes the bytecode line by line.
Garbage Collector	Automatically cleans unused objects from memory to free up space.

Memory Areas in JVM:

- **Heap** – Stores all objects.
- **Stack** – Stores method calls and local variables.
- **Method Area** – Stores class-level data like method info, class names.
- **PC Register** – Keeps track of which instruction is being executed.
- **Native Method Stack** – Used when calling native (non-Java) code.

Reflection API:

What is the Reflection API?

- The Reflection API allows Java programs to examine and modify the structure of classes, methods, and fields at runtime.
- This is not possible normally in regular Java code.

What can we do with Reflection?

We Can...	Meaning
Access class name	Find the name of any class.
Get method names	Check which methods a class has.
Check fields	Find out what variables a class has.
Call methods	Call a method without using its name directly.
Create objects	Create a new object without using new.

Simple Example:

```
import java.lang.reflect.*;
public class Demo {
    public static void main(String[] args) throws Exception {
        // Load the class
        Class c = Class.forName("java.util.Date");

        // Get all methods
        Method[] methods = c.getDeclaredMethods();

        // Print method names
        for(Method m : methods) {
            System.out.println(m.getName());
        }
    }
}
```

What’s happening here?

- Class.forName("java.util.Date") → loads the Date class at runtime.
- getDeclaredMethods() → gets all methods declared in the class.
- We can loop through and print them.

Why use Reflection?

- For framework development (like Spring, Hibernate)
- For advanced tools (like debuggers, IDEs, testing tools like JUnit)
- When you need to handle classes or objects dynamically.

Note:

- Reflection is powerful but can be slow and less secure.
- Use it only when necessary.

Final Variables:

What is a final variable?

- A final variable is like a constant.
- Once you give it a value, you can't change it again.
- If you try to reassign it, the compiler will show an error.
- Example:
  ```
  final int x = 10;
  x = 20;  // ❌ Error: Cannot assign a new value to final variable 'x'
  ```

Rules for final Variables:

- **Must be initialized only once:** We must assign a value to a final variable once and only once.
- **We can initialize it in different ways:**
  - **When you declare it:**
    ```
    final int a = 5;
    ```
  - **Inside a constructor (for instance variables):**
    ```
    class MyClass {
        final int number;

        MyClass(int value) {
    ```

```
        number = value;  // ✅ Okay to assign here
    }
}
```

- **But once it's assigned, it cannot be changed again:**

```
number = 10;  // ❌ Error if you try to assign again
```

Final Can Also Be Used With:

1. Final Methods

- A final method cannot be overridden by a subclass.
- Useful when you want to lock the behavior of a method.
- Example:

```
class A {
    final void show() {
        System.out.println("Hello from A");
    }
}

class B extends A {
    // void show() { }  ❌ Error: Can't override final method
}
```

2. Final Classes

- A final class cannot be inherited.
- No other class can extend a final class.
- Example:

  ```
  final class Animal {
      void sound() {
          System.out.println("Animal sound");
      }
  }
  // class Dog extends Animal {}  ❌ Error: Cannot
  extend final class
  ```

In Short:

Keyword	Effect
final variable	Value cannot change after it's set.
final method	Method cannot be overridden in subclasses.
final class	Class cannot be extended/inherited.

Use Cases:

- Use final for constants like PI = 3.14.
- Use it to protect methods and classes from being changed by others.

Command Line Parameters:

What are Command Line Parameters?

- Command Line Parameters are values (arguments) you pass to your Java program when running it from the terminal or command prompt.
- These values are received by the main() method through the String[] args array.

Why use them?

- To give input to your program without writing code to take input using Scanner.
- Useful when running programs as scripts, in automation, or in batch jobs.

How do they work?

Let’s say you have this Java program:

```
public class MyProgram {
  public static void main(String[] args) {
    System.out.println("First argument: " + args[0]);
    System.out.println("Second argument: " + args[1]);
  }
}
```

Run it from the terminal:

```
java MyProgram Hello 123
```

What happens:

args[0] = "Hello"

args[1] = "123"

So the output will be:

```
First argument: Hello
Second argument: 123
```

Key Points:

- The parameters are always Strings (even numbers like 123 come as "123").
- You can convert them to numbers using Integer.parseInt(), if needed.
- If you try to access args[2] and there is no third input, it will cause an error (ArrayIndexOutOfBoundsException).

Example with conversion:

```
public class Sum {
  public static void main(String[] args) {
    int a = Integer.parseInt(args[0]);
    int b = Integer.parseInt(args[1]);
    System.out.println("Sum = " + (a + b));
  }
}
```

Run it like:

```
java Sum 10 20
```

Output:

```
Sum = 30
```

Use Cases:

Use Case	Example
Quick input	Testing simple programs with values
Automation	Running scripts with different input each time
Batch processing	Feeding multiple files or commands without user interaction

Wrapper Classes:

What are Wrapper Classes?

- Java has primitive data types like int, char, boolean, etc.

- But Java is an object-oriented language, and sometimes you need objects instead of primitives (e.g., for using in collections like ArrayList, which only store objects).
- Wrapper classes are used to wrap primitive types into objects so they can behave like objects.

List of Primitive Types and Their Wrapper Classes:

Primitive Type	Wrapper Class
int	Integer
char	Character
boolean	Boolean
double	Double
float	Float
long	Long
short	Short
byte	Byte

Example: Manual Wrapping and Unwrapping

```
int a = 10;

// Wrapping: converting primitive to object

Integer obj = Integer.valueOf(a);

// Unwrapping: converting object to primitive

int b = obj.intValue();

System.out.println("Wrapped object: " + obj);
System.out.println("Unwrapped value: " + b);
```

Autoboxing and Unboxing

Java makes it easier by automatically converting between primitives and objects.

Autoboxing:

- Automatically converts a primitive → object.
- Integer x = 5; // Behind the scenes: Integer.valueOf(5)

Unboxing:

- Automatically converts an object → primitive.
- int y = x;   // Behind the scenes: x.intValue()

Why Wrapper Classes are Useful:

Use Case	Reason
Collections	Like ArrayList, HashMap need objects, not primitives.
Null values	Primitives can't be null, but wrapper objects can.
Utility methods	Wrapper classes provide useful methods (e.g., Integer.parseInt()).