KOENIG

step forward

Core Java, Git, Build Tools & JUnit Testing

Java Recap & Object-Oriented Programming

Introduction to Java

- Java is a high-level, object-oriented, class-based programming language developed by Sun Microsystems (now owned by Oracle) in 1995.
- It is widely used for building:
  - Desktop applications
  - Web applications
  - Mobile applications (especially Android)
  - Enterprise software
  - Embedded systems

Installing and setting up IntelliJ IDEA Community Edition

- IntelliJ IDEA Community Edition is a free and open-source integrated development environment (IDE) developed by JetBrains.
- It is designed primarily for Java development, but also supports other JVM-based languages like Kotlin, Groovy, and Scala.
- Download and install the IntelliJ IDEA Community Edition
  - Go to the Official Website: https://www.jetbrains.com/idea/download
  - Choose Community Edition.
  - Click Download for your operating system (Windows/Linux/macOS).
  - Run the downloaded installer and follow the installation wizard.

Creating and running a Java project in IntelliJ

- Launch IntelliJ IDEA and Create New Project

1. Open IntelliJ IDEA.
2. Click “New Project”.
3. In the New Project Wizard:
    - Select Java (make sure JDK is installed and selected)
    - Choose a Project SDK (you can download a JDK from here too)
    - Click Next, then Finish

- If you don't have a JDK:
    - IntelliJ lets you download JDK automatically.
    - Recommended: JDK 21 (LTS) or higher

Creating and running a Java project in IntelliJ

- Create Java Class

1. In the Project Explorer, right-click src → New → Java Class.
2. Enter class name, e.g., Main.
3. Write your Java code inside:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Welcome to Java!");
    }
}
```

Creating and running a Java project in IntelliJ

- Run the Java Program

1. Right-click inside the code editor → Run 'Main.main()'
2. Or click the green ▶️ Run button at the top right.
3. The output will appear in the console at the bottom:

Welcome to Java!

Java Basics Recap

Java Basics Recap

- Variables (primitive vs reference types)
- Operators (arithmetic, relational, logical, bitwise)
- Control Flow (if-else, switch, loops)
- Methods (parameters, return types, overloading)

Variables (primitive vs reference types)

- A data type in Java defines the kind of data a variable can store, how much memory it uses, and the operations allowed on it.
- Data types ensure type safety during compilation.
- Data types prevent invalid operations (e.g., adding a number to a string incorrectly).
- Data types help the compiler allocate memory efficiently.
- In Java, data types are categorized into two main types
  - Primitive Data Types (Built-in)
  - Reference Data Types

Variables (primitive vs reference types)

- Primitive Data Types (Built-in)
  - These are the most basic data types provided by Java. They are not objects and store actual values.

Type	Size	Description	Example
byte	1 byte	Whole number from -128 to 127	byte b = 100;
short	2 bytes	Whole number from -32K to 32K	short s = 30000;
int	4 bytes	Common integer type	int x = 100000;
long	8 bytes	Large integer values	long l = 123456L;
float	4 bytes	Decimal, single precision	float f = 3.14f;
double	8 bytes	Decimal, double precision	double d = 3.14159;
char	2 bytes	Single character (Unicode)	char c = 'A';
boolean	1 bit	True/False	boolean b = true;

Variables (primitive vs reference types)

- Reference Data Types
  - These types refer to objects, not actual values. Memory stores references (addresses) of the object.

Type	Description	Example
String	Represents a sequence of characters	String s = "Hello";
Arrays	Group of similar data elements	int[] arr = {1,2,3};
Classes	Blueprint for objects	class Person {} Person p = new Person();
Interfaces	Reference to a set of methods (contract)	interface MyInterface {} class MyClass implements MyInterface {} MyClass obj = new MyClass();

Variable declaration, initialization, and type casting

- Variable Declaration
  - Declaring a variable means reserving memory space and specifying the data type.
- Syntax:

```
dataType variableName;
```

- Example:

```
int age;        // declaration
String name;    // declaration
```

Variable declaration, initialization, and type casting

- Variable Initialization
  - Initialization means assigning a value to the variable for the first time.
- Syntax:

  variableName = value;

  - Or declare and initialize in one line:

  dataType variableName = value;

- Example:
  - int age = 25;   // declare + initialize
  - String name = "John";   // declare + initialize

Variable declaration, initialization, and type casting

- Type Casting in Java
  - Type casting means converting a variable from one data type to another.
  - Java supports:
- Implicit Casting (Widening)
  - Automatically done when converting smaller to larger types.
- Example

```
int a = 10;
long b = a;   // implicit casting int → long (Safe)
```

Variable declaration, initialization, and type casting

- Explicit Casting (Narrowing)
  - Done manually when converting larger to smaller types.
- Example

```
double x = 9.99;
int y = (int) x;   // y becomes 9 - double → int (Risk of data loss)
```

Scope and lifetime of variables in different blocks

- In Java, the scope and lifetime of a variable depend on where it's declared — inside a method, block, class, or parameter list.
- Scope: Where a variable is accessible/visible in the program.
- Lifetime: How long the variable exists in memory (until it’s destroyed or out of scope).

Scope and lifetime of variables in different blocks

- Types of Variable Scope in Java

Scope Type	Declared Inside	Scope (Accessible From)	Lifetime
Local	Method or block	Only inside that method/block	From method/block entry to exit
Instance	Inside a class (no static)	All instance methods (via this)	As long as the object exists
Static (Class)	Inside a class with static	All static methods in the class	As long as class is loaded
Parameter	Inside method signature	Only inside the method	For the duration of the method

Scope and lifetime of variables in different blocks

- Local Variable

```
void show() {
    int count = 5;  // local variable
    System.out.println(count);
}
```

  - Scope: Only inside show()
  - Lifetime: Created when method is called, destroyed after method ends

Scope and lifetime of variables in different blocks

- Instance Variable

```
class Car {
    String color;  // instance variable
    void display() {
        System.out.println(color);
    }
}
```

  - Scope: Whole class (via object)
  - Lifetime: As long as object exists

Scope and lifetime of variables in different blocks

- Static Variable

```
class Counter {
    static int count = 0;  // static variable
    void increment() {
        count++;
    }
}
```

  - Scope: Shared by all objects
  - Lifetime: As long as the class is loaded in memory

Scope and lifetime of variables in different blocks

- Block Scope

```
if (true) {
    int x = 10;
    System.out.println(x); // OK
}
// System.out.println(x); // Error: x not visible here
```

  - Scope: Inside the { } block
  - Lifetime: Starts when block is entered, ends when block is exited

One-dimensional and multidimensional arrays with syntax and usage

- One-Dimensional (1D) Array
  - A 1D array is a list of elements of the same type arranged in a single row.
  - Use 1D arrays for linear data (e.g., marks, ages).
- Syntax:

```
// Declaration
dataType[] arrayName;
// Declaration + Initialization
dataType[] arrayName = new dataType[size];
// Declaration + Initialization + Values
dataType[] arrayName = {val1, val2, val3};
```

One-dimensional and multidimensional arrays with syntax and usage

- One-Dimensional (1D) Array
- Example:

```
int[] numbers = {10, 20, 30, 40};
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

- Output:

```
10
20
30
40
```

One-dimensional and multidimensional arrays with syntax and usage

- Multidimensional Arrays (2D, 3D, etc.)
  - A multidimensional array is an array of arrays.
  - Use 2D or multi-D arrays for matrix-like or tabular data (e.g., table, matrix).
- 2D Array Syntax:

```
// Declaration
dataType[][] arrayName;
// Declaration + Initialization
dataType[][] arrayName = new dataType[rows][columns];
// Declaration + Initialization + Values
dataType[][] matrix = {   {1, 2, 3}, {4, 5, 6}  };
```

One-dimensional and multidimensional arrays with syntax and usage

- Multidimensional Arrays - Example:

```
int[][] matrix = {    {1, 2},    {3, 4},    {5, 6}  };
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}
```

- Output:

```
1 2
3 4
5 6
```

Working with String in Java

- In Java, String is one of the most commonly used classes, representing a sequence of characters.
- Strings in Java are immutable, meaning once created, they cannot be changed.
- Declaring and Initializing Strings
  - String str1 = "Hello";   // String literal
  - String str2 = new String("World");   // Using constructor

Common String methods – length(), charAt(), equals(), concat(), substring()

- length()
  - Returns the number of characters in the string.

```
String str = "Java";
System.out.println(str.length());  // Output: 4
```

- charAt(int index)
  - Returns the character at the specified index (0-based).

```
String str = "Hello";
System.out.println(str.charAt(1));  // Output: 'e'
```

Common String methods – length(), charAt(), equals(), concat(), substring()

- equals(String anotherString)
  - Compares the contents of two strings (case-sensitive).

```
String a = "Java";
String b = "Java";
System.out.println(a.equals(b));  // true
```

- concat(String str)
  - Joins the specified string to the end of the original string.

```
String s1 = "Hello";
String s2 = "World";
String result = s1.concat(" ").concat(s2);
System.out.println(result);  // Output: Hello World
```

Common String methods – length(), charAt(), equals(), concat(), substring()

- substring(int beginIndex, int endIndex)
  - Extracts part of the string from beginIndex (inclusive) to endIndex (exclusive).

```
String str = "Programming";
System.out.println(str.substring(0, 6));  // Output: Progra
System.out.println(str.substring(6));     // Output: mming
```

String comparison, case conversion, and trimming methods

- String Comparison Methods
- equals(String str)
  - Compares content (case-sensitive).

```
"Java".equals("java");   // false
"Java".equals("Java");   // true
```

- equalsIgnoreCase(String str)
  - Compares content, ignoring case.

```
"Java".equalsIgnoreCase("java");  // true
```

String comparison, case conversion, and trimming methods

- String Comparison Methods
- compareTo(String str)
  - Lexicographically compares two strings.
  - Returns:
    - 0 if equal
    - Positive if first string is greater
    - Negative if first string is smaller

    "abc".compareTo("abd"); // -1

    "abc".compareTo("abc"); // 0

    "abd".compareTo("abc"); // 1
- compareToIgnoreCase(String str)
  - Same as compareTo() but ignores case.

String comparison, case conversion, and trimming methods

- Case Conversion Methods
- toLowerCase()
  - Converts all characters to lowercase.

```
String s = "HeLLo";
System.out.println(s.toLowerCase());  // hello
```

- toUpperCase()
  - Converts all characters to uppercase.

```
String s = "HeLLo";
System.out.println(s.toUpperCase());  // HELLO
```

String comparison, case conversion, and trimming methods

- Trimming Whitespace
- trim()
  - Removes leading and trailing whitespaces.

```
String s = "   Hello Java   ";
System.out.println(s.trim());  // "Hello Java"
```

Operators (arithmetic, relational, logical, bitwise)

- Arithmetic Operators
  - Used to perform basic mathematical operations.

Operator	Description	Example	Result
+	Addition	5 + 2	7
-	Subtraction	5 - 2	3
*	Multiplication	5 * 2	10
/	Division	5 / 2	2 (int)
%	Modulus (Remainder)	5 % 2	1

Operators (arithmetic, relational, logical, bitwise)

- Relational (Comparison) Operators
  - Used to compare two values.

Operator	Description	Example	Result
==	Equal to	a == b	true / false
!=	Not equal to	a != b	true / false
>	Greater than	a > b	true / false
<	Less than	a < b	true / false
>=	Greater or equal	a >= b	true / false
<=	Less or equal	a <= b	true / false

Operators (arithmetic, relational, logical, bitwise)

- Logical Operators
  - Used to combine multiple boolean expressions.

Operator	Description	Example	Result
&&	Logical AND	a > 3 && b < 10	true if both true
||	Logical OR	a > 3 || b < 10	true if any one true
!	Logical NOT	!(a > b)	Opposite of condition

Operators (arithmetic, relational, logical, bitwise)

- Bitwise Operators
  - Operate on bits (used for performance or low-level tasks).

Operator	Description	Example	Result
&	Bitwise AND	5 & 3	1
|	Bitwise OR	5 & 3	7
^	Bitwise XOR	5 ^ 3	6
~	Bitwise Complement	~5	-6
<<	Left shift	5 << 1	10
>>	Right shift	5 >> 1	2

Control Flow (if-else, switch, loops)

- if Statement
  - Used to execute a block of code only if a condition is true.
- Syntax:

```
if (condition) {
    // code to execute if condition is true
}
```

Control Flow (if-else, switch, loops)

- if-else Statement
  - Used to execute one block if condition is true, otherwise another block.
- Syntax:

```
if (condition) {
    // true block
} else {
    // false block
}
```

Control Flow (if-else, switch, loops)

- nested if-else Statement
    - Used when we have to make multiple decisions (conditions inside conditions).
- Syntax:

```
if (condition1) {
    if (condition2) {
        // inner true block
    } else {
        // inner false block
    }
} else {
    // outer false block
}
```

Control Flow (if-else, switch, loops)

- switch-case Statement
  - Used to test a variable against multiple constant values.
- Syntax:

```
switch (variable) {
    case value1:
        // code block
        break;
    case value2:
        // code block
        break;
    default:        // default code block
}
```

Control Flow (if-else, switch, loops)

- In Java, loops are used to execute a block of code repeatedly until a certain condition is met.
- for Loop Syntax:

```
for (initialization; condition; update) {
    // loop body
}
```

- while Loop
  - Executes as long as the condition is true.
- Syntax:

```
while (condition) {
    // loop body
}
```

Control Flow (if-else, switch, loops)

- do-while Loop
  - Executes at least once, even if the condition is false.
- Syntax:

```
do {
    // loop body
} while (condition);
```

Control Flow (if-else, switch, loops)

- In Java, break, continue, and return are control flow keywords used to change the normal flow of program execution, especially inside loops and methods.
- break Keyword
  - Used to exit a loop or switch statement immediately.
- Syntax:

```
break;
```

Control Flow (if-else, switch, loops)

- continue Keyword
  - Used to skip the current iteration and continue with the next iteration of the loop.
  - Syntax:

```
continue;
```

- return Keyword
  - Used to exit from a method and optionally return a value.
  - Syntax:

```
return;          // in void methods
return value;    // in methods with return type
```

OOP Essentials

OOP Essentials

- Classes & Objects, Constructors
- Inheritance (extends keyword, method overriding)
- Polymorphism (compile-time vs runtime)
- Abstraction (abstract classes, methods)
- Encapsulation (access modifiers, getters/setters)

Classes & Objects, Constructors

- Class
  - A class is a blueprint for creating objects.
  - It defines attributes (variables) and methods (functions).

```
public class Car {
    // Attributes (fields)
    String brand;
    int speed;
    // Method
    void startEngine() {
        System.out.println(brand + " engine started.");
    }
}
```

Classes & Objects, Constructors

- Object
  - An object is an instance of a class. It represents a real-world entity with state and behavior.

```
public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();        // Object creation
        car1.brand = "Honda";        // Accessing attribute
        car1.startEngine();          // Calling method
    }
}
```

Classes & Objects, Constructors

- Attributes
  - Attributes (also called fields or instance variables) define the state of an object.
  - Example:

```
String brand;
int speed;
```

- Methods
  - Methods define the behavior of an object. They are like functions defined inside a class.
  - Example:

```
void startEngine() {
        System.out.println("Engine started");
}
```

Classes & Objects, Constructors

- In Java, a constructor is a special method used to initialize objects.
- It has the same name as the class and no return type (not even void).
- Default Constructor
  - A default constructor is a constructor with no parameters.
  - If you don't define any constructor in a class, Java provides one automatically.
- Syntax:

```
public class Car {
    Car() { // Default constructor
        System.out.println("Car object created!");
    }
}
```

Classes & Objects, Constructors

- Parameterized Constructor
  - A parameterized constructor accepts arguments to initialize the object with specific values.
- Syntax:

```
public class Car {
    String color;
    Car(String c) { // Parameterized constructor
        color = c;
    }
}
```

Classes & Objects, Constructors

- In Java, classes define attributes (fields) and methods (functions).
- When you create an object from a class, that object becomes a reference to access these members (variables and methods) using the dot (.) operator.
- It allows encapsulation (grouping data and methods together).
- You interact with real-world entities as objects (e.g., Student, Car).
- Helps in organizing and reusing code.

Classes & Objects, Constructors

```
public class Car {
    String color;
    void drive() {
        System.out.println("The car is driving.");
    }
}
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();      // Object creation
        myCar.color = "Red";        // Accessing attribute
        myCar.drive();              // Calling method
    }
}
```

Inheritance (extends keyword, method overriding)

- Inheritance allows one class to inherit the properties and behaviors (fields and methods) of another class.
- It helps with code reusability and creating hierarchical relationships.
- Inheritance represented using **extends** keyword.
- It defines a parent-child or superclass-subclass relationship.
- It's also called class inheritance.

Inheritance (extends keyword, method overriding)

- Inheritance – Example
  - Dog IS-A Animal → Because Dog inherits from Animal.

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}
```

Inheritance (extends keyword, method overriding)

- Method Overriding
  - A subclass provides a specific implementation of a method that is already defined in its superclass.
  - Used to provide a different behavior for an inherited method.
  - Runtime Polymorphism

Inheritance (extends keyword, method overriding)

- Method Overriding - Example

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

Polymorphism (compile-time vs runtime)

- Polymorphism means many forms. In Java, it allows one interface to be used for a general class of actions. It comes in two types:
- Compile-Time Polymorphism (Static Binding)
  - Achieved by: Method Overloading
  - Method resolution happens at compile-time
  - Same method name, but different parameter types or counts
- Run-Time Polymorphism (Dynamic Binding)
  - Achieved by: Method Overriding
  - Method resolution happens at runtime
  - Involves inheritance and method overriding

Polymorphism (compile-time vs runtime)

- Method Overloading
  - Same method name, but different parameters (type, number, or order) within the same class.
  - Used to perform similar operations with different input types or counts.
  - Compile-Time Polymorphism

Polymorphism (compile-time vs runtime)

- Method Overloading - Example

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
    int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

Polymorphism (compile-time vs runtime)

- Method Overriding
  - A subclass provides a specific implementation of a method that is already defined in its superclass.
  - Used to provide a different behavior for an inherited method.
  - Runtime Polymorphism

Polymorphism (compile-time vs runtime)

- Method Overriding - Example

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

Abstraction (abstract classes, methods)

- Abstraction is the process of hiding implementation details and showing only the essential features of an object.
- It helps reduce complexity by letting the user focus on what an object does instead of how it does it.

Benefit	Description
Hides details	Focus only on relevant behavior of objects
Improves reuse	Common behavior can be shared via abstract classes/interfaces
Promotes testing	Makes unit testing and mocking easier

Abstraction (abstract classes, methods)

- Abstraction Using Abstract Classes – Example

```
abstract class Animal {
    abstract void sound(); // abstract method (no body)

    void eat() {
        System.out.println("This animal eats food.");
    }
}
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
```

Encapsulation (access modifiers, getters/setters)

- Encapsulation is one of the four pillars of Object-Oriented Programming (OOP).
- It is the mechanism of wrapping data (variables) and code (methods) together into a single unit (a class) and restricting access to some of the object's components.
- Data hiding is the practice of restricting direct access to internal class variables.
- It's implemented using the private access modifier.
- The class exposes controlled access through getter and setter methods.

Encapsulation (access modifiers, getters/setters)

- In Java, access modifiers define the scope (visibility) of classes, methods, constructors, and variables.
- They control where members of a class can be accessed from other classes.
- public
  - Access Level: Everywhere (same class, same package, outside package, subclass).
  - Use Case: When you want members to be accessible globally.

```
public class Car {
    public String model = "Tesla";
    public void start() {
        System.out.println("Car started");
    }
}
```

Encapsulation (access modifiers, getters/setters)

- private
  - Access Level: Only within the same class.
  - Use Case: When data/methods should not be accessed outside the class (encapsulation).

```
public class BankAccount {
    private double balance;

    private void calculateInterest() {
        // only accessible inside this class
    }
}
```

Encapsulation (access modifiers, getters/setters)

- protected
  - Access Level: Same class, Same package, Subclasses
  - Use Case: When members need to be visible to child classes or same package.

```
class Animal {
    protected void eat() {
        System.out.println("Animal eats");
    }
}
class Dog extends Animal {
    void sound() {
        eat(); // allowed due to protected
    }
}
```

Encapsulation (access modifiers, getters/setters)

- Default (no modifier)
  - Access Level: Package-private (accessible only within the same package).
  - Use Case: When you want members accessible only to other classes in the same package.

```
class Book {
    int pages = 100; // default access
    void read() {
        System.out.println("Reading book");
    }
}
```

Encapsulation (access modifiers, getters/setters)

Modifier	Same Class	Same Package	Subclass	Other Classes
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
default	Yes	Yes	No	No
private	Yes	No	No	No

Advanced OOP

Advanced OOP

- Interfaces & multiple inheritance
- Inner Classes (static, member, anonymous)
- Composition vs Inheritance (HAS-A vs IS-A relationship)

Interfaces & multiple inheritance

- An interface in Java is a blueprint of a class.
- It defines method signatures (abstract methods) without implementation.
- Classes implement interfaces to define the actual behavior.

Benefit	Explanation
Abstraction	Focuses on what the class should do, not how.
Multiple inheritance	A class can implement multiple interfaces, unlike extending classes.
Flexibility and decoupling	Code becomes more modular and testable.
Design pattern integration	Essential in Strategy, Factory, Observer, etc.

Interfaces & multiple inheritance

- Abstraction Using Interfaces – Example

```
interface Vehicle {
    void start();   // public & abstract by default
    void stop();
}
class Car implements Vehicle {
    public void start() {
        System.out.println("Car starts");
    }
    public void stop() {
        System.out.println("Car stops");
    }
}
```

Interfaces & multiple inheritance

- Differences: Abstract Class vs Interface

Feature	Abstract Class	Interface
Methods	Can have both abstract and non-abstract	Only abstract methods (Java 7), Default & static (Java 8+)
Fields	Can have fields	Only constants (public static final)
Inheritance	Single inheritance	Multiple inheritance supported
Constructors	Can have constructors	Cannot have constructors
Access Modifiers	Can use all	All methods are public by default

Interfaces & multiple inheritance

- In Java, interfaces allow abstraction and are the only way to achieve multiple inheritance of type.
- An interface is a reference type, similar to a class, that can contain:
  - Abstract methods (implicitly public and abstract)
  - Default and static methods (from Java 8)
  - Constants (public static final)

```
interface Animal {
    void makeSound(); // abstract method
}
```

Interfaces & multiple inheritance

- Multiple Inheritance via Interface
  - Java supports multiple inheritance of behavior using interfaces

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

interface B {
    void methodB();
}
```

```
// Multiple inheritance using interfaces
class C implements A, B {
    public void methodA() {
        System.out.println("From A");
    }
    public void methodB() {
        System.out.println("From B");
    }
}
```

Inner Classes (static, member, anonymous)

- In Java, a class can be defined inside another class. These are called inner classes. They help in logical grouping and encapsulation.
- Static Inner Class
  - Can access only static members of the outer class directly.
  - Does not require an object of the outer class to be created.
- Non-Static Inner Class
  - Can access both static and non-static members of the outer class.
  - Requires an instance of the outer class to be created.

Inner Classes (static, member, anonymous)

- Static Inner Class - Example

```
class Outer {
    static int outerStatic = 10;

    static class StaticNested {
        void display() {
            System.out.println("Static: " + outerStatic);
        }
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Outer.StaticNested obj = new Outer.StaticNested();
        obj.display();
    }
}
```

Inner Classes (static, member, anonymous)

- Non-Static Inner Class - Example

```
class Outer {
    int outerValue = 50;

    class Inner {
        void show() {
            System.out.println("Non-static: " + outerValue);
        }
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.show();
    }
}
```

Inner Classes (static, member, anonymous)

- Local Inner Class
  - A local inner class is defined inside a method or block and can only be accessed within that method/block.
  - Use Case:
    - Used for helper classes that should not be accessible outside the method, e.g., temporary tasks, validations, etc.

Inner Classes (static, member, anonymous)

- Anonymous Class
  - An anonymous inner class is a class with no name, used to instantiate and override a class or interface on the fly.
  - Use Case:
    - Used in event handling, threading, functional interfaces, and when you need a short implementation of a class/interface used only once.

Composition vs Inheritance (HAS-A vs IS-A relationship)

- Inheritance allows one class to inherit the properties and behaviors (fields and methods) of another class.
- It helps with code reusability and creating hierarchical relationships.
- IS-A Relationship (Inheritance)
  - Represented using extends keyword.
  - It defines a parent-child or superclass-subclass relationship.
  - It's also called class inheritance.

Composition vs Inheritance (HAS-A vs IS-A relationship)

- IS-A Relationship (Inheritance) – Example
  - Dog IS-A Animal → Because Dog inherits from Animal.

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}
```

Composition vs Inheritance (HAS-A vs IS-A relationship)

- HAS-A Relationship (Composition)
  - Represents object composition (one class contains another class).
  - No keyword is required; just use the object of another class as a field.
  - Used when one class uses another class.

Composition vs Inheritance (HAS-A vs IS-A relationship)

- HAS-A Relationship (Composition)
  - Car HAS-A Engine → Because it holds an instance of Engine.

```
class Engine {
    void start() {
        System.out.println("Engine starts.");
    }
}
class Car {
    Engine engine = new Engine(); // Car HAS-A Engine
    void drive() {
        engine.start();  // using Engine’s functionality
        System.out.println("Car is moving.");
    }
}
```

Handson Labs

Handson Labs

- Bank Account System: Implement deposits, withdrawals, and balance check using classes & methods.
- Employee Class Hierarchy: Create base class Employee and subclasses (Manager, Developer) demonstrating inheritance & polymorphism.
- Debugging in IDE: Set breakpoints, inspect variables, step into/step over methods.

Java APIs, Exceptions & Multithreading

Collections Framework

Collections Framework

- List (ArrayList, LinkedList)
- Set (HashSet)
- Map (HashMap)
- Queue (PriorityQueue)

What is Collections Framework?

- The Java Collections Framework is a unified architecture for storing, retrieving, and manipulating groups of objects (data collections) efficiently.
- Key Components of the Collections Framework
  - Interfaces
    - Abstract data structures (e.g., List, Set, Map)
  - Implementations
    - Concrete classes (e.g., ArrayList, HashSet, LinkedList, HashMap)
  - Algorithms
    - Utility methods for searching, sorting, etc. (e.g., Collections.sort())

List (ArrayList, LinkedList)

- List
  - A List is an ordered collection that allows duplicate elements.
- Key Features:
  - Maintains insertion order
  - Allows duplicate values
  - Access via index (0-based)
- Common Implementations:
  - ArrayList
  - LinkedList

List (ArrayList, LinkedList)

- ArrayList – Dynamic Array

```
import java.util.ArrayList;
public class ArrayListDemo {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Python");
        list.add("Java"); // Allows duplicates
        System.out.println("ArrayList: " + list); // [Java, Python, Java]
    }
}
```

  - Use case: Random access, frequent reads
  - Backed by array → Fast for get(index)

List (ArrayList, LinkedList)

- LinkedList – Doubly Linked List

```
import java.util.LinkedList;
public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();
        list.add("C");
        list.add("C++");
        list.addFirst("Assembly");
        list.addLast("Java");
        System.out.println("LinkedList: " + list); // [Assembly, C, C++, Java]
    }
}
```

  - Use case: Frequent insertion/removal
  - Slower random access, but fast insertion/deletion

Set (HashSet)

- Set
  - A Set is a collection of unique elements.
- Key Features:
  - No duplicates allowed
  - No guaranteed order (unless using LinkedHashSet)
  - Cannot access elements via index
- Common Implementations:
  - HashSet – no order
  - LinkedHashSet – insertion order

Set (HashSet)

- HashSet – Unique elements, no order

```
import java.util.HashSet;
public class HashSetDemo {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Apple"); // Duplicate ignored
        System.out.println("HashSet: " + set); // Output order not guaranteed
    }
}
```

  - Use case: Store unique items
  - Fast lookup, no duplicates

Map (HashMap)

- Map
  - A Map stores key-value pairs.
- Key Features:
  - Unique keys
  - Values can be duplicated
  - Not a subtype of Collection
- Common Implementations:
  - HashMap – no order
  - LinkedHashMap – insertion order

Map (HashMap)

- HashMap – Key-value pairs

```
import java.util.HashMap;
public class HashMapDemo {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();
        map.put(101, "Alice");
        map.put(102, "Bob");
        map.put(101, "Charlie"); // Replaces value for key 101
        System.out.println("HashMap: " + map); // {101=Charlie, 102=Bob}
        System.out.println("Value for key 102: " + map.get(102)); // Bob
    }
}
```

  - Use case: Lookup by key
  - Fast access, keys must be unique

Queue (PriorityQueue)

- A Queue is a collection that follows FIFO (First-In-First-Out) order.
- Defined in java.util.Queue interface.
- Common implementations:
  - LinkedList (basic queue)
  - PriorityQueue (priority-based ordering)
  - ArrayDeque (double-ended queue)

Queue (PriorityQueue)

- PriorityQueue
  - A PriorityQueue is a special type of queue in Java that orders elements based on priority rather than FIFO.
- Key Points
  - By default, it orders elements in natural order (ascending for numbers, alphabetical for strings).
  - Can accept a custom Comparator for different ordering.
  - Does not allow null elements.

Queue (PriorityQueue)

- PriorityQueue - Example

```
import java.util.*;
public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        pq.add(50);
        pq.add(10);
        pq.add(30);
        while (!pq.isEmpty()) {
            System.out.println(pq.poll()); // retrieves and removes smallest element
        }
    }
}
```

Exception Handling

Exception Handling

- Checked vs Unchecked exceptions
- try-catch-finally block
- Throwing & creating Custom Exceptions
- Best practices (specific exceptions, avoiding empty catch blocks)

What is Exception?

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
- Java provides a powerful exception-handling mechanism to gracefully manage errors like:
  - Division by zero
  - Accessing invalid array index
  - File not found
  - Null references, etc.

Checked vs Unchecked exceptions

- Java exceptions are mainly divided into two types:
- Checked Exceptions (Compile-Time Exceptions)
  - These are checked by the compiler at compile time.
  - The program won’t compile unless these are either handled with try-catch or declared using throws.
- Examples:
  - IOException
  - FileNotFoundException
  - SQLException

Checked vs Unchecked exceptions

- Unchecked Exceptions (Runtime Exceptions)
  - These are not checked by the compiler.
  - Occur due to programming mistakes, like dividing by zero or null access.
  - You can handle them, but it’s not mandatory.
- Examples:
  - ArithmeticException
  - NullPointerException
  - ArrayIndexOutOfBoundsException

try-catch-finally block

- The try-catch Block
  - In Java, the try-catch block is used to handle exceptions gracefully without crashing the program.
- Syntax:

```
try {
    // Code that might throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
}
```

try-catch-finally block

- The finally Block
  - The finally block is always executed after the try block (with or without a catch), regardless of whether an exception is thrown or not.
- Syntax:

```
try {
    // risky code
} catch (Exception e) {
    // handle exception
} finally {
    // cleanup code that always runs
}
```

try-catch-finally block

- Multiple Catch Blocks
  - You can handle different types of exceptions using multiple catch blocks.
  - Each block catches a specific exception.
- Syntax:

```
try {
    // risky code
} catch (ArithmeticException e) {
    // handle arithmetic error
} catch (ArrayIndexOutOfBoundsException e) {
    // handle array index error
} catch (Exception e) {
    // handle all other exceptions
}
```

try-catch-finally block

- Nested Try Blocks
  - A try block can be placed inside another try block. This is useful when different operations need separate exception handling.
- Syntax:

```
try {
    // Outer try block
    try {
        // Inner try block
    } catch (...) {
        // Inner catch
    }
} catch (...) {
    // Outer catch
}
```

Throwing & creating Custom Exceptions

- throw Keyword
  - The throw keyword is used to explicitly throw an exception (usually custom or runtime exception).
  - Syntax:

```
throw new ExceptionType("message");
```

- throws Keyword
  - The throws keyword is used in method declaration to indicate that the method might throw a checked exception, and the caller must handle it.
  - Syntax:

```
public void readFile() throws IOException {
    // code that might throw IOException
}
```

Throwing & creating Custom Exceptions

- A custom exception is a user-defined class that extends Java’s Exception or RuntimeException classes to indicate specific error conditions.
- Extend Exception (Checked Exception)

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

Throwing & creating Custom Exceptions

- Extend RuntimeException (Unchecked Exception)

```
public class NegativeAmountException extends RuntimeException {
    public NegativeAmountException(String message) {
        super(message);
    }
}
```

Exception Handling - Best practices

- Catch Only What You Can Handle
  - Don’t swallow exceptions just to make code compile.
  - If you can’t handle it meaningfully, rethrow or wrap it.

```
try {
    // risky code
} catch (IOException e) {
    // handle it properly (log, retry, fallback, etc.)
}
```

Exception Handling - Best practices

- Don’t Ignore Exceptions

```
catch (IOException e) {
    // BAD
}
```

- Always log or rethrow:

```
catch (IOException e) {
    logger.error("Failed to read file", e);
}
```

Exception Handling - Best practices

- Use Custom Exceptions for Clarity
  - Use InvalidAccountException instead of throwing a generic Exception.

```
class InvalidAccountException extends Exception {
    public InvalidAccountException(String msg) {
        super(msg);
    }
}
```

Exception Handling - Best practices

- Never Catch Exception or Throwable Directly
  - Catching Exception hides the real issue.
  - Catch specific exceptions (e.g., IOException, SQLException).
- Bad:

```
catch (Exception e) { }
```

- Good:

```
catch (FileNotFoundException e) { ... }
catch (IOException e) { ... }
```

Exception Handling - Best practices

- Fail Fast
  - If a method cannot proceed due to invalid input, throw an exception immediately rather than continuing with bad state.

```
if (amount < 0) {
    throw new IllegalArgumentException("Amount cannot be negative");
}
```

Functional Programming

Functional Programming

- Lambdas: (args) -> expression
- Streams API: map, filter, reduce, collect

Lambda expressions and method references

- In Java, Lambda Expressions and Method References simplify writing anonymous functions and make code more concise, especially when working with functional interfaces like Runnable, and Comparator.
- Lambda Expressions (Java 8+)
  - A lambda expression is a short block of code that takes in parameters and returns a value.
  - It can be used to implement methods of functional interfaces.
- Syntax:

```
(parameters) -> statement
or
(parameters) -> { statements }
```

Lambda expressions and method references

- Example 1: Without Lambda (Before Java 8)

```
Runnable r = new Runnable() {
    public void run() {
        System.out.println("Running");
    }
};
```

- Example 2: With Lambda

```
Runnable r = () -> System.out.println("Running");
```

Lambda expressions and method references

- Method References
  - Method references are a shorthand notation of a lambda expression to call a method.
- Syntax:
  - ClassName::methodName
- Example

```
List<String> names = Arrays.asList("Zara", "Asha", "Bala");
names.forEach(System.out::println);
```

Functional interfaces

- Functional Interface
  - A Functional Interface in Java is an interface that contains exactly one abstract method.
  - It can have any number of default or static methods.
  - Functional Interfaces are used as the basis for lambda expressions and method references.

Functional interfaces

- Example: Creating a Functional Interface

```
@FunctionalInterface
interface MyFunctionalInterface {
    void doSomething();  // Single abstract method
}
```

- Usage with Lambda

```
MyFunctionalInterface obj = () -> System.out.println("Doing something!");
obj.doSomething();
```

Stream API – filter, map, reduce, forEach, collect

- The Stream API (introduced in Java 8) is used to process collections of data (like List, Set, etc.) in a functional style using a sequence of operations like filtering, mapping, reducing, etc.
- Why use Stream API?
  - To perform bulk operations efficiently on collections
  - To write cleaner, more readable and declarative code
  - To enable pipelined processing (chaining operations)
  - To support parallel execution easily
- Stream Workflow

```
collection.stream().filter(...).map(...).forEach(...);
```

Stream API – filter, map, reduce, forEach, collect

- filter()
  - Filters elements based on a given condition (Predicate).

```
List<String> names = List.of("Ashok", "Anil", "Amit", "John");
names.stream()
    .filter(name -> name.startsWith("A"))
    .forEach(System.out::println);
```

Stream API – filter, map, reduce, forEach, collect

- map()
  - Transforms each element in the stream using a Function.

```
List<String> names = List.of("ashok", "john");
List<String> upper = names.stream()
                    .map(String::toUpperCase)
                    .collect(Collectors.toList());
System.out.println(upper);
```

Stream API – filter, map, reduce, forEach, collect

- reduce()
  - Reduces all elements in the stream to a single result (like sum).

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
int sum = numbers.stream()
            .reduce(0, (a, b) -> a + b);
System.out.println("Sum = " + sum);
```

Stream API – filter, map, reduce, forEach, collect

- forEach()
  - Performs an action for each element (like printing).

```
List<String> cities = List.of("Delhi", "Mumbai", "Chennai");
cities.stream().forEach(System.out::println);
```

Stream API – filter, map, reduce, forEach, collect

- collect()
  - Used to convert the stream into a collection (List, Set, Map, etc.).

```
List<String> names = List.of("Ashok", "Amit", "John");
List<String> filtered = names.stream()
                             .filter(n -> n.startsWith("A"))
                             .collect(Collectors.toList());
System.out.println(filtered);
```

Multithreading Basics

Multithreading Basics

- Creating threads: Thread vs Runnable
- Thread lifecycle & Synchronization

What is Multithreading?

- Multithreading is a programming feature that allows concurrent execution of two or more threads (lightweight subprocesses) in a program to improve performance and responsiveness, especially on multi-core processors.

Term	Description
Thread	A lightweight subprocess that runs concurrently with other threads.
Multithreading	The ability of a CPU or a single core to execute multiple threads concurrently.
Main thread	The initial thread started by the JVM when a program begins.
Concurrency	Performing multiple tasks at the same time (interleaved).
Parallelism	Performing multiple tasks simultaneously, especially on multiple CPU cores.

Creating threads: Thread vs Runnable

- Creating Threads By Extending Thread Class

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Running thread: " + Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();          // New state
        t1.start();                            // Now in Runnable → Running
    }
}
```

- You must call start() (not run() directly) to actually create a new thread.

Creating threads: Thread vs Runnable

- Creating Threads By Implementing Runnable Interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Running thread: " + Thread.currentThread().getName());
    }
    public static void main(String[] args) {
        Runnable r = new MyRunnable();
        Thread t = new Thread(r);          // New state
        t.start();                         // Runnable → Running
    }
}
```

  - This approach is preferred when your class needs to extend another class, because Java supports only single inheritance.

Thread lifecycle & Synchronization

- A thread in Java goes through the following five states:

State	Description
New	Thread object is created but not started yet.
Runnable	Thread is ready to run and waiting for CPU time.
Running	Thread is executing its task.
Blocked/Waiting	Thread is paused temporarily (e.g., waiting for a resource or sleep()).
Terminated	Thread has completed execution or was forcibly stopped.

Thread lifecycle & Synchronization

- What is Synchronization?
- In multithreading, multiple threads may try to read/write shared data at the same time.
- Without control, this can lead to race conditions.
- Synchronization ensures that only one thread can access a critical section (shared resource) at a time.

Thread lifecycle & Synchronization

- Synchronization - Example
  - Declaring a method synchronized ensures that only one thread can execute it at a time per object.

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

Thread lifecycle & Synchronization

- Synchronization - Example

```
public class SyncMethodExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });
        t1.start();        t2.start();
        t1.join();        t2.join();
        System.out.println("Final Count: " + counter.getCount());
    }
}
```

Handson Labs

Handson Labs

- Library Management System: Store books in a List, search/filter using Streams API.
- Custom Exception: Create InvalidBookException for invalid data input.
- Multi-threaded Ticket Booking System: Multiple threads booking seats.
- Employee Records with Stream API: Filter employees by salary, department, etc.

Git, Build Tools & JUnit Testing

Git Essentials

Git Essentials

- Git init, clone, add, commit, push, pull
- Branching & merging
- Handling merge conflicts

Introduction to Git

- Git is a distributed version control system (DVCS) used to track changes in source code during software development.
- What Git Does
  - Tracks changes: Keeps a history of who changed what and when.
  - Manages versions: Lets you switch between different versions of your project.
  - Collaboration: Multiple developers can work on the same project without overwriting each other’s work.
  - Branching & merging: Allows you to create isolated branches for features, bug fixes, or experiments, and later merge them.

Introduction to Git

- Key Features
  - Distributed: Every developer has a complete copy of the repository.
  - Fast & lightweight: Operations like commits, branching, and merging are efficient.
  - Reliable: Designed to handle large projects (Linux kernel is managed with Git).
  - Open-source: Free to use, created by Linus Torvalds in 2005.
- Why Git is Important
  - Ensures code safety by keeping full history.
  - Simplifies team collaboration with remote repositories (e.g., GitHub).
  - Supports parallel development using branches.
  - Provides rollback ability if something breaks.

Git init, clone, add, commit, push, pull

- git init
  - Use: Create a new local Git repository.
  - Command:

    $ git init
  - This initializes Git in your current project folder.
- git clone
  - Use: Copy (download) an existing remote repository to your local machine.
  - Command:

    $ git clone <repository-url>
  - Example:

    $ git clone https://github.com/user/project.git

Git init, clone, add, commit, push, pull

- git add
  - Use: Stage changes (prepare files to be committed).
  - Command:

```
$ git add <file-name>    # Add a single file
$ git add .              # Add all changes
```

- git commit
  - Use: Save changes to the local repository with a message.
  - Command:

```
$ git commit -m "Meaningful commit message"
```

Git init, clone, add, commit, push, pull

- git push
  - Use: Upload local commits to a remote repository.
  - Command:

    $ git push origin <branch-name>
  - Example:

    $ git push origin main
- git pull
  - Use: Fetch and merge changes from remote repository into local.
  - Command:

    $ git pull origin <branch-name>
  - Example:

    $ git pull origin main

Branching & merging

- Branching in Git
  - A branch is like a separate line of development.
  - By default, Git creates a branch called main (or master in older repos).
  - You can create other branches to work on new features or fixes without touching the main code.
  - Commands

```
$ git branch <branch-name>        # Create a new branch
$ git checkout <branch-name>      # Switch to that branch (older way)
$ git switch <branch-name>        # Switch to that branch (newer way)
$ git checkout -b <branch-name>  # Create and switch in one step
```

- Example: creates a branch called feature-login and switches to it

```
$ git checkout -b feature-login
```

Branching & merging

- Merging in Git
  - Merging combines changes from one branch into another (often into main).
- Commands
  - Switch to the branch you want to merge into:

    $ git switch main
  - Merge another branch:

    $ git merge feature-login

Branching & merging

- Merge Types
  - Fast-forward merge (simple case):
    - If no new commits exist on main since branching, Git just moves the pointer forward.
    - main → feature-login
  - Three-way merge (more common):
    - If both branches have new commits, Git combines them and creates a new commit called a merge commit.

Handling merge conflicts

- What is a Merge Conflict?
  - A merge conflict happens when Git cannot automatically decide which changes to keep.
- Typical case:
  - Developer A edits line 10 of app.js.
  - Developer B also edits line 10 of app.js in another branch.
  - When merging, Git sees two different edits → conflict.

Handling merge conflicts

- How to Detect Conflicts
  - When you merge:

    $ git merge feature-branch

  - You’ll see:

Auto-merging app.js

CONFLICT (content): Merge conflict in app.js

Automatic merge failed; fix conflicts and then commit the result.

Handling merge conflicts

- Conflict Markers in Files
  - Open the file, you’ll see something like:

```
<<<<<<< HEAD
console.log("Hello from main branch");
=======
console.log("Hello from feature branch");
>>>>>>> feature-branch
```

  - Between <<<<<<< HEAD and ======= → your current branch code (e.g., main).
  - Between ======= and >>>>>>> feature-branch → incoming branch code.

Handling merge conflicts

- How to Resolve
  - Open the conflicted file(s).
  - Decide which code to keep (or merge manually).
  - Mark conflict as resolved:

    ```
    $ git add app.js
    ```
  - Commit the merge:

    ```
    $ git commit
    ```
- Abort a Merge (if you messed up)

    ```
    $ git merge --abort
    ```
  - This cancels the merge and brings you back to the state before merging.

Handling merge conflicts

- VS Code for Easier Conflict Resolution
  - shows options:
    - "Accept Current Change"
    - "Accept Incoming Change"
    - "Accept Both Changes".

Build Tools

Build Tools

- Maven: POM.xml structure, dependencies, lifecycle (compile, test, package, install)
- Gradle: build.gradle scripts, dependency management
- Maven vs Gradle comparison

Maven: POM.xml structure, dependencies, lifecycle (compile, test, package, install)

- What is Maven?
  - Maven is a build automation and project management tool for Java.
  - Uses POM.xml to configure project details.
  - Handles dependencies (external libraries).
  - Provides a lifecycle for building, testing, packaging, and deploying apps.

Maven: POM.xml structure, dependencies, lifecycle (compile, test, package, install)

- POM.xml (Project Object Model)
  - The POM.xml is the heart of a Maven project.
  - It describes the project and configuration.
- Key tags:
  - groupId → unique project group (like company domain).
  - artifactId → name of the project.
  - version → version of the project.
  - packaging → type (jar, war, pom).
  - dependencies → libraries needed.

Maven: POM.xml structure, dependencies, lifecycle (compile, test, package, install)

- Dependencies
  - Maven manages external libraries from a central repository.
  - Defined inside <dependencies> in pom.xml.
- Example

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
<version>3.2.5</version>
</dependency>
```

  - Maven automatically downloads this library and its transitive dependencies.

Maven: POM.xml structure, dependencies, lifecycle (compile, test, package, install)

- Standard Mave Lifecycle Phases
  - compile → compile source code.
  - test → run unit tests.
  - package → bundle compiled code into jar/war.
  - verify → run integration tests.
  - install → put package in local Maven repository.
- Example commands:

```
$ mvn compile      # Compile source code
$ mvn test         # Run tests
$ mvn package      # Create JAR/WAR
$ mvn install      # Install to local repo
```

Gradle: build.gradle scripts, dependency management

- What is Gradle?
  - Gradle is a build automation tool (like Maven, but more flexible).
  - Uses Groovy (or Kotlin) scripts instead of XML.
  - Handles dependencies, builds, testing, and deployment.
  - Faster and more customizable than Maven.

Gradle: build.gradle scripts, dependency management

- build.gradle Script
  - The main file in a Gradle project is build.gradle.
  - It defines project settings, plugins, dependencies, and tasks.
- Key parts:
  - plugins → add functionality (Java, Spring Boot, etc.).
  - repositories → define where dependencies come from (Maven Central, JCenter, local).
  - dependencies → external libraries.
  - application → define the main class (for runnable apps).

Gradle: build.gradle scripts, dependency management

- Dependency Management
  - Gradle has different dependency configurations (scopes):
    - implementation → normal dependencies required at compile & runtime.
    - compileOnly → needed only at compile time.
    - runtimeOnly → needed only at runtime.
    - testImplementation → used only for tests.
- Example

```
dependencies {
implementation 'org.springframework.boot:spring-boot-starter-web:3.2.5'
compileOnly 'org.projectlombok:lombok:1.18.30'
runtimeOnly 'mysql:mysql-connector-java:8.0.33'
testImplementation 'org.junit.jupiter:junit-jupiter:5.10.2'
}
```

Gradle: build.gradle scripts, dependency management

- Running Gradle Tasks
  - Some common Gradle commands:

```
$ gradle build        # Compiles, tests, and packages app
$ gradle clean        # Deletes build folder
$ gradle test         # Runs unit tests
$ gradle run          # Runs main class (if application plugin applied)
$ gradle dependencies # Shows dependency tree
```

Maven vs Gradle comparison

Feature	Maven	Gradle
Configuration Style	XML (pom.xml)	Groovy/Kotlin DSL (build.gradle, build.gradle.kts)
Readability	Verbose but structured	Concise, flexible scripting
Performance	Slower (parses XML, no incremental builds by default)	Faster (incremental builds + build cache)
Flexibility	Convention over configuration (fixed lifecycle phases)	Highly customizable with tasks & plugins
Dependency Management	Uses Maven Central (default)	Uses Maven Central, JCenter, or custom repos
Plugins	Rich ecosystem of predefined plugins	More modern plugin system, can use Maven plugins too
Learning Curve	Easier for beginners (more rigid)	Slightly steeper (more freedom and scripting)
IDE Support	Excellent (Eclipse, IntelliJ, VS Code)	Excellent (Eclipse, IntelliJ, VS Code)
Popularity	Older, very widely used in enterprise	Growing fast, preferred in modern projects (Spring Boot, Android)
Best For	Standard enterprise Java apps with fixed workflows	Complex/large projects, Android, microservices, highly customizable builds

JUnit Testing

JUnit Testing

- Lifecycle: @BeforeEach, @AfterEach, @BeforeAll, @AfterAll
- Assertions: assertEquals, assertTrue, assertThrows
- Parameterized tests
- Basics of TDD (red → green → refactor cycle)

Overview of unit testing and role of JUnit in Java

- Unit testing is the process of testing individual units or components of a software application in isolation to ensure they perform as expected.
  - A unit typically means a method or class.
  - Performed by developers during development.
  - Helps identify bugs early in the development cycle.

Overview of unit testing and role of JUnit in Java

- Characteristics of Good Unit Tests

Characteristic	Description
Isolated	Tests only one method/class without external systems
Fast	Executes quickly and frequently
Repeatable	Same result every time it's run
Independent	Doesn’t rely on the outcome of other tests
Automated	Can run without manual steps

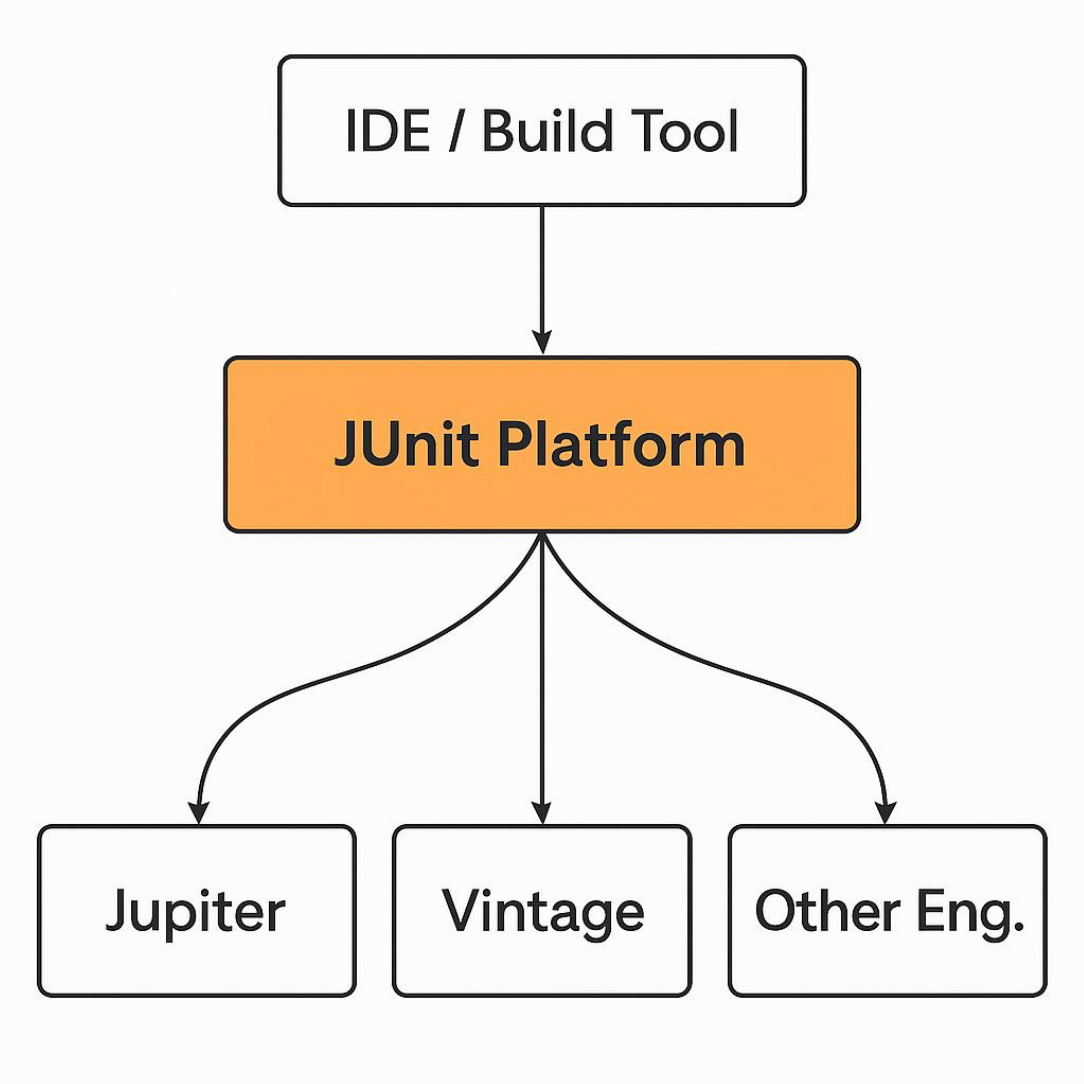
Overview of unit testing and role of JUnit in Java

- JUnit is the standard testing framework for Java.
- It provides annotations, assertions, and test runners to help you write and manage unit tests effectively.
- Role of JUnit in Unit Testing

Feature	How JUnit Helps
Test annotations	Like @Test, @BeforeEach, @AfterEach
Assertions	Like assertEquals, assertTrue, assertThrows
Test organization	Helps group and structure test cases
Automation support	Runs tests via IDEs, build tools (Maven), CI tools
Reporting	Provides pass/fail status with error details

Understanding JUnit 5 architecture – Jupiter, Platform, Vintage

- JUnit 5 is a modular and extensible testing framework, designed to overcome the limitations of JUnit 4 and support modern development needs like Java 8+ features, and dynamic tests.



Understanding JUnit 5 architecture – Jupiter, Platform, Vintage

- JUnit 5 is composed of three main sub-projects:
- JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage
- JUnit Platform
  - The foundation of JUnit 5
  - Launches and discovers tests
  - Provides a TestEngine API so any testing framework like JUnit can plug in
  - Integrates with IDEs, build tools (Maven), and CI tools
  - Think of it as the "runner and orchestrator" of all tests.

Understanding JUnit 5 architecture – Jupiter, Platform, Vintage

- JUnit Jupiter
  - The new programming model and test engine for writing tests using JUnit 5 annotations and Java 8+ features
  - Provides:
    - @Test, @BeforeEach, @AfterEach, etc.
    - Nested tests (@Nested)
    - Parameterized tests
  - This is where you write your modern JUnit 5 tests.

Understanding JUnit 5 architecture – Jupiter, Platform, Vintage

- JUnit Vintage
  - Allows backward compatibility with JUnit 3 and JUnit 4 tests
  - Provides a test engine that runs older JUnit 3/4 test classes inside the JUnit 5 environment
  - Use this when migrating old test suites to JUnit 5 gradually.

Environment setup in IntelliJ IDEA with JUnit 5 dependencies

Step 1: Create a New Maven Project

- Open IntelliJ IDEA
- Click on File → New → Project
- Enter Product Name: junit5demo
- Select Build System: Maven
- Click Finish

- Your project will now be created with the standard Maven structure.

Environment setup in IntelliJ IDEA with JUnit 5 dependencies

Step 2: Add JUnit 5 Dependencies to pom.xml

- Open the pom.xml file and add the following inside <dependencies>:

```
<dependencies>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.10.0</version>
  <scope>test</scope>
 </dependency>
</dependencies>
```

- IntelliJ will auto-download the dependencies. If not, right-click the project and select Reload Maven Project.

Environment setup in IntelliJ IDEA with JUnit 5 dependencies

Step 3: Create the Main Class

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Environment setup in IntelliJ IDEA with JUnit 5 dependencies

Step 4: Create a Test Class

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
public class CalculatorTest {
    @Test
    void testAdd() {
        Calculator calc = new Calculator();
        assertEquals(5, calc.add(2, 3));
    }
}
```

Environment setup in IntelliJ IDEA with JUnit 5 dependencies

- Step 5: Run the Test
  - Right-click the CalculatorTest.java file
  - Click Run 'CalculatorTest'
  - You should see a green checkmark if the test passes.

Lifecycle: @BeforeEach, @AfterEach, @BeforeAll, @AfterAll

- @BeforeAll, @AfterAll, @BeforeEach, @AfterEach, annotations control the setup and teardown process of test cases.

Annotation	Runs...	Purpose
@BeforeAll	Once before all tests	Set up shared resources
@AfterAll	Once after all tests	Clean up shared resources
@BeforeEach	Before each test method	Initialize/reset test data
@AfterEach	After each test method	Clean up after each test
@Test	Marks a method as a test case	Execute test logic

Assertions: assertEquals, assertTrue, assertThrows

Assertion	Description	Example
assertEquals	Asserts that two values are equal	assertEquals(4, 2 + 2)
assertNotEquals	Asserts that two values are not equal	assertNotEquals(5, 2 + 2)
assertTrue	Asserts that a condition is true	assertTrue(5 > 2)
assertFalse	Asserts that a condition is false	assertFalse(3 > 5)
assertNull	Asserts that the value is null	assertNull(obj)
assertNotNull	Asserts that the value is not null	assertNotNull(obj)
assertArrayEquals	Asserts two arrays are equal	assertArrayEquals(arr1, arr2)
assertThrows	Asserts that an exception is thrown	assertThrows(Exception.class, ...)

Parameterized tests

- @ParameterizedTest – Test with Multiple Data Inputs
  - Use this when you want to run the same test with different inputs, avoiding code duplication.

```
public class ParamTest {
    @ParameterizedTest
    @ValueSource(ints = {1, 2, 3, 4, 5})
    void testEvenNumbers(int number) {
        assertTrue(number > 0);
    }
}
```

Basics of TDD (red → green → refactor cycle)

- TDD stands for Test-Driven Development.
- It is a software development practice where you write tests before writing the actual code.
- TDD Cycle (Red-Green-Refactor)
  - RED – Write a test for a new feature. The test will fail initially because the functionality doesn’t exist yet.
  - GREEN – Write the minimum code necessary to make the test pass.
  - REFACTOR – Clean up the code (improve structure, remove duplication) while ensuring tests still pass.
  - This cycle is repeated for every small piece of functionality.

Basics of TDD (red → green → refactor cycle)

- Example (TDD in Java with JUnit)

Step 1 – Write a failing test (RED):

```java
@Test
void shouldAddTwoNumbers() {
    Calculator calculator = new Calculator();
    assertEquals(5, calculator.add(2, 3)); // Fails, Calculator not implemented yet
}
```

Basics of TDD (red → green → refactor cycle)

- Example (TDD in Java with JUnit)

Step 2 – Write code to pass test (GREEN):

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Step 3 – Refactor if needed (REFACTOR):

- Code is already simple, no need to refactor.

Basics of TDD (red → green → refactor cycle)

- Benefits of TDD
  - Ensures high test coverage
  - Produces clean, modular, and reliable code
  - Catches bugs early
  - Provides living documentation (tests show how the code is expected to behave)
  - Makes refactoring safer

Basics of TDD (red → green → refactor cycle)

- TDD vs Traditional Development

Traditional Development	TDD
Write code first, then test	Write test first, then code
Bugs discovered later	Bugs caught early
Often low test coverage	High test coverage
Harder to refactor	Refactoring is safer

Handon Labs

Handon Labs

- Git Repo Setup: Initialize repo, create branches, push to GitHub.
- Maven Project Creation: Create & run simple HelloWorld app.
- Convert Maven → Gradle Project: Add Gradle build scripts.
- JUnit Tests for Services: Write unit tests for Calculator or Employee Service class.

Happy Learning :)