Topics –

1. Java basics, primitive data types, variables, operators, date, Flow control, Conditions, Switch, Lop
2. OOPs Concepts
3. Exception Handling, Collection, File I/O

Great question! In Java, **classes** and **objects** are the foundation of object-oriented programming. Here's a clear breakdown:

**🧱 What is a Class?**

A **class** is like a blueprint or template. It defines the structure and behavior that its objects will have.

* Think of a class as a recipe for making cookies.
* It specifies what ingredients (fields/variables) and steps (methods/functions) are needed.
* But the recipe itself isn’t a cookie—it’s just the instructions.

**Example:**

public class Car {

String color;

int speed;

void drive() {

System.out.println("The car is driving.");

}

}

This Car class defines two properties (color, speed) and one behavior (drive()).

**🚗 What is an Object?**

An **object** is an actual instance of a class. It’s the real thing created using the blueprint.

* If the class is the recipe, the object is the cookie you bake from it.
* You can create multiple objects from the same class, each with its own unique data.

**Example:**

public class Main {

public static void main(String[] args) {

Car myCar = new Car(); // Creating an object of Car

myCar.color = "Red";

myCar.speed = 100;

myCar.drive(); // Output: The car is driving.

}

}

Here, myCar is an object of the Car class. It has its own color and speed.

**🧠 Quick Summary**

| **Concept** | **Description** | **Analogy** |
| --- | --- | --- |
| Class | Blueprint for creating objects | Recipe |
| Object | Instance of a class | Cookie made from the recipe |

Want to try writing your own class and object? I can help you build one step-by-step!

Awesome—this trio is essential to understanding how Java works. Let’s break it down clearly:

**⚙️ JVM (Java Virtual Machine)**

* **What it does:** Executes Java bytecode (compiled .class files).
* **Role:** It’s the engine that runs Java programs.
* **Platform-specific:** Each OS has its own JVM implementation.
* **Key tasks:** Loads code, verifies it, executes it, and manages memory (like garbage collection).

🧠 Think of JVM as the interpreter that makes your Java code run on any machine.

**📦 JRE (Java Runtime Environment)**

* **What it includes:** JVM + core libraries + other components needed to run Java apps.
* **Purpose:** Allows you to *run* Java programs, but not write or compile them.
* **Use case:** Ideal for users who just want to execute Java applications.

🧃 Analogy: JRE is like a smoothie—JVM is the blender, and the libraries are the ingredients.

**🛠️ JDK (Java Development Kit)**

* **What it includes:** JRE + development tools (like javac, java, javadoc, debugger).
* **Purpose:** Used to *develop* Java applications.
* **Use case:** Essential for developers writing and compiling Java code.

🧰 Analogy: JDK is the full toolbox. It lets you build the app, not just run it.

**🔁 How They Work Together**

1. You write Java code (.java file).
2. JDK compiles it into bytecode (.class file).
3. JRE (via JVM) runs the bytecode on your machine.

**🧭 Quick Comparison Table**

| **Component** | **Includes** | **Purpose** | **Used By** |
| --- | --- | --- | --- |
| JVM | — | Runs bytecode | Everyone |
| JRE | JVM + Libraries | Runs Java apps | End users |
| JDK | JRE + Dev tools | Develop Java apps | Developers |

Want to see a real example of how these work together in a simple Java program? I can walk you through it!

Here’s your classic **Hello World** program in Java—the perfect starting point for any budding developer:

// File name: HelloWorld.java

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, World!");

}

}

**🧠 What’s Happening Here?**

* public class HelloWorld: Defines a class named HelloWorld. The file name must match this class name.
* public static void main(String[] args): This is the **entry point** of any Java application. The JVM looks for this method to start execution.
* System.out.println("Hello, World!");: Prints the message to the console.

**🚀 How to Run It**

1. Save the code in a file named HelloWorld.java.
2. Open a terminal or command prompt.
3. Compile it using:
4. javac HelloWorld.java
5. Run it using:
6. java HelloWorld

You’ll see:

Hello, World!

Want to tweak it to print your name or add a little flair? I can help jazz it up!

Here’s a simple and complete Java program that takes **two user inputs** and performs **all basic arithmetic operations**: addition, subtraction, multiplication, division, and modulus.

import java.util.Scanner;

public class ArithmeticOperations {

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

// Taking input from user

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

int num1 = input.nextInt();

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

int num2 = input.nextInt();

// Performing arithmetic operations

int sum = num1 + num2;

int difference = num1 - num2;

int product = num1 \* num2;

int quotient = num1 / num2; // Integer division

int remainder = num1 % num2;

// Displaying results

System.out.println("Sum: " + sum);

System.out.println("Difference: " + difference);

System.out.println("Product: " + product);

System.out.println("Quotient: " + quotient);

System.out.println("Remainder: " + remainder);

}

}

**🧠 Notes:**

* This program uses Scanner to read input from the user.
* Division here is **integer division**. If you want a **decimal result**, you can cast one of the numbers to double:
* double quotient = (double) num1 / num2;

Want to extend this to handle floating-point numbers or add error handling (like division by zero)? I’d be happy to help you level it up!

Java has a rich set of **data types** that help define the kind of data a variable can hold. These are broadly categorized into **primitive** and **non-primitive** types.

**🔹 Primitive Data Types (8 total)**

These are the most basic types and store simple values directly in memory:

| **Type** | **Description** | **Size** | **Example** |
| --- | --- | --- | --- |
| byte | 8-bit signed integer | 1 byte | byte b = 100; |
| short | 16-bit signed integer | 2 bytes | short s = 1000; |
| int | 32-bit signed integer | 4 bytes | int i = 50000; |
| long | 64-bit signed integer | 8 bytes | long l = 123456789L; |
| float | 32-bit floating point | 4 bytes | float f = 3.14f; |
| double | 64-bit floating point | 8 bytes | double d = 3.14159; |
| char | Single 16-bit Unicode character | 2 bytes | char c = 'A'; |
| boolean | Logical value: true or false | ~1 byte | boolean flag = true; |

**🔸 Non-Primitive Data Types**

These are reference types that point to objects in memory:

* **String** – e.g., String name = "Java";
* **Arrays** – e.g., int[] numbers = {1, 2, 3};
* **Classes** – Custom types you define
* **Interfaces** – Abstract types for shared behavior
* **Objects** – Instances of classes

**🧠 Why It Matters**

Choosing the right data type:

* Saves memory (byte vs int)
* Improves performance
* Makes your code more readable and maintainable

Switch case  
  
public class DayFinder {

public static void main(String[] args) {

int day = 4;

switch (day) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

case 3:

System.out.println("Wednesday");

break;

case 4:

System.out.println("Thursday");

break;

case 5:

System.out.println("Friday");

break;

case 6:

System.out.println("Saturday");

break;

case 7:

System.out.println("Sunday");

break;

default:

System.out.println("Invalid day number");

}

}

}

public class Calculator {

// Overloaded method: adds two integers

public int add(int a, int b) {

return a + b;

}

// Overloaded method: adds three integers

public int add(int a, int b, int c) {

return a + b + c;

}

// Overloaded method: adds two doubles

public double add(double a, double b) {

return a + b;

}

public static void main(String[] args) {

Calculator calc = new Calculator();

System.out.println("Sum of 2 ints: " + calc.add(5, 10));

System.out.println("Sum of 3 ints: " + calc.add(5, 10, 15));

System.out.println("Sum of 2 doubles: " + calc.add(5.5, 10.5));

}

}

An **abstract class** in Java is like a partially built house—it provides a structure, but leaves some rooms unfinished so subclasses can complete them. It’s a powerful tool for enforcing design and promoting reuse.

**🧱 What Is an Abstract Class?**

* Declared using the abstract keyword.
* Cannot be instantiated directly.
* Can contain:
  + **Abstract methods** (no body—must be implemented by subclasses)
  + **Concrete methods** (with body—can be inherited or overridden)
  + **Constructors**, **fields**, and even **static methods**

**✅ Example: Abstract Class in Action**

abstract class Animal {

// Abstract method (no body)

abstract void makeSound();

// Concrete method

void sleep() {

System.out.println("Zzz...");

}

}

class Dog extends Animal {

// Implementing abstract method

void makeSound() {

System.out.println("Woof!");

}

}

public class Main {

public static void main(String[] args) {

Animal myDog = new Dog();

myDog.makeSound(); // Output: Woof!

myDog.sleep(); // Output: Zzz...

}

}

**🧠 Why Use Abstract Classes?**

* To define a **common base** for related classes.
* To enforce that certain methods **must be implemented**.
* To provide **shared code** while allowing customization.

**🔍 Key Rules**

| **Feature** | **Abstract Class** |
| --- | --- |
| Can have constructors | ✅ Yes |
| Can have abstract methods | ✅ Yes |
| Can have concrete methods | ✅ Yes |
| Can be extended | ✅ Yes |
| Can be instantiated | ❌ No |

If you're curious how abstract classes compare to interfaces—or want to build a real-world example like a shape hierarchy or employee system—I’d be happy to help you sketch it out!

import java.io.\*;

public class ExceptionDemo {

public static void main(String[] args) {

// ArithmeticException

try {

int a = 10;

int b = 0;

int result = a / b;

} catch (ArithmeticException e) {

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

}

// ArrayIndexOutOfBoundsException

try {

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

System.out.println(arr[5]);

} catch (ArrayIndexOutOfBoundsException e) {

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

}

// FileNotFoundException

try {

FileReader fr = new FileReader("nonexistentfile.txt");

} catch (FileNotFoundException e) {

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

}

// NullPointerException (Name type exception)

try {

String name = null;

System.out.println("Name length: " + name.length());

} catch (NullPointerException e) {

System.out.println("NullPointerException caught: Name is null");

}

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

}

}

Creating a **custom exception** in Java allows you to define your own error types tailored to your application’s logic. It’s perfect when built-in exceptions like NullPointerException or IllegalArgumentException don’t quite capture the problem.

🛠️ How to Create a Custom Exception

You can create a custom exception by extending either:

* Exception → for **checked exceptions**
* RuntimeException → for **unchecked exceptions**

// Step 1: Define the custom exception

class InvalidNameException extends Exception {

public InvalidNameException(String message) {

super(message);

}

}

// Step 2: Use the custom exception

public class NameValidator {

public static void validateName(String name) throws InvalidNameException {

if (name == null || name.trim().isEmpty()) {

throw new InvalidNameException("Name cannot be null or empty.");

}

System.out.println("Valid name: " + name);

}

public static void main(String[] args) {

try {

validateName(""); // Try with invalid name

} catch (InvalidNameException e) {

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

}

}

}

ArrayList

import java.util.ArrayList;

import java.util.List;

public class ShoppingCart {

public static void main(String[] args) {

// 🛒 Create a shopping cart

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

// ➕ Add items to the cart

cart.add("Laptop");

cart.add("Smartphone");

cart.add("Headphones");

// 🔁 Iterate and print cart items

System.out.println("Shopping Cart Items:");

for (String item : cart) {

System.out.println("- " + item);

}

}

}

Vector

import java.util.Vector;

public class VectorExample {

public static void main(String[] args) {

// 📦 Create a Vector of Strings

Vector<String> cart = new Vector<>();

// ➕ Add items to the Vector

cart.add("Laptop");

cart.add("Smartphone");

cart.add("Headphones");

// 🔁 Iterate through the Vector

System.out.println("Shopping Cart Items:");

for (String item : cart) {

System.out.println("- " + item);

}

// 👀 Access an item by index

System.out.println("First item: " + cart.get(0));

// ❌ Remove an item

cart.remove("Smartphone");

System.out.println("After removing Smartphone: " + cart);

}

}

Great question! Vector and ArrayList are both part of Java’s Collection Framework and implement the List interface, but they differ in several important ways. Here's a clear comparison to help you decide which one to use:

**⚔️ Vector vs ArrayList in Java**

| **Feature** | **ArrayList** | **Vector** |
| --- | --- | --- |
| **Thread Safety** | ❌ Not synchronized (not thread-safe) | ✅ Synchronized (thread-safe) |
| **Performance** | 🚀 Faster (no overhead from locking) | 🐢 Slower due to synchronization |
| **Growth Strategy** | 📈 Grows by 50% when full | 📈 Grows by 100% (doubles in size) |
| **Legacy Status** | ✅ Modern (introduced in JDK 1.2) | 🕰️ Legacy class (from JDK 1.0) |
| **Traversal Options** | 🔁 Uses Iterator only | 🔁 Uses Iterator and Enumeration |
| **Use Case** | Ideal for single-threaded apps | Suitable for multi-threaded scenarios |

**🧠 Which Should You Use?**

* Use **ArrayList** when:
  + You don’t need built-in thread safety.
  + Performance is a priority.
  + You want modern, flexible code.
* Use **Vector** when:
  + You need thread-safe operations out of the box.
  + You’re maintaining legacy code.

💡 *Pro Tip*: If you need a thread-safe ArrayList, you can wrap it using:

List<String> safeList = Collections.synchronizedList(new ArrayList<>());

public class MyLinkedList {

// Node class

static class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

// Head of the list

Node head = null;

// ➕ Add element at the end

public void add(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

} else {

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

// ❌ Remove element by value

public void remove(int data) {

if (head == null) return;

if (head.data == data) {

head = head.next;

return;

}

Node current = head;

while (current.next != null && current.next.data != data) {

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

}

}

// 📋 Print the list

public void printList() {

Node current = head;

System.out.print("Linked List: ");

while (current != null) {

System.out.print(current.data + " ");

current = current.next;

}

System.out.println();

}

// 🚀 Main method to test

public static void main(String[] args) {

MyLinkedList list = new MyLinkedList();

list.add(10);

list.add(20);

list.add(30);

list.printList(); // Output: Linked List: 10 20 30

list.remove(20);

list.printList(); // Output: Linked List: 10 30

}

}

VIVA

Diff betn next and nextline

Great question! In Java, the Scanner class provides both next() and nextLine() methods to read user input—but they behave quite differently. Here's a breakdown:

| **Method** | **Reads Until...** | **Can Read Spaces?** | **Cursor Behavior** | **Common Use Case** |
| --- | --- | --- | --- | --- |
| next() | First whitespace | ❌ No | Stays on same line | Reading single words or tokens |
| nextLine() | End of line (\n) | ✅ Yes | Moves to next line | Reading full lines or sentences |

**🧪 Example:**

Scanner sc = new Scanner(System.in);

// Using next()

System.out.print("Enter a word: ");

String word = sc.next(); // Input: Hello World → Output: "Hello"

// Using nextLine()

System.out.print("Enter a sentence: ");

String sentence = sc.nextLine(); // Input: Hello World → Output: "Hello World"

**⚠️ Gotcha:**

If you use next() before nextLine(), the leftover newline character (\n) from next() can cause nextLine() to read an empty string. To fix this, add an extra sc.nextLine() to consume the newline.

Want to see a real-world example where this causes bugs? I can show you how to avoid that trap.