Akhil java notes

**1.Pre-oops concepts**

Java software was developed by **sun-micro systems** in the year **1994** and released into industry in **1995**.

The vendor for java software is initially sun-micro-systems but now it is maintained by **ORACLE**.

The father of java software is **james gosling**.

🡺Java is a high-level, object-oriented programming language used to build:

**Desktop applications**

These applications will runs on local machine,which does not runs on both browser and server & even we cannot share any information through the internet.

Ex:- VLC media player or km player, Ms office.

**Web applications**

These applications will runs on both server and browser withrespect to the network and we can share any information through the internet.

Ex:- Gmail,Facebook,internet banking etc.

**Mobile apps**

Mobile apps are applications (software) that you install and use on your smartphone or tablet like:

WhatsApp,Instagram,Zomato,Flipkart,YouTube

🡺Java software is mainly classified into three categories

**1.JSE(Java standard edition) :-** we can develop desktop applicatons.

This is the core/basic Java — used to build standalone desktop applications.

It includes:

Core Java (OOP, Collections, Exception Handling, etc.)

GUI (Swing, AWT) , Multithreading

Input/Output (IO) , JDBC (Database connectivity)

Example apps: Calculator, media player, small tools

**2.JEE(Java enterprise edition):-** we can develop distributed applications

It includes:

Servlets and JSP

Web services

EJB (Enterprise JavaBeans)

Frameworks like Spring, Hibernate

**Example apps:** Online shopping systems, banking software, e-commerce sites

**3.JME (Java mobile edition):-** By using JME we can develop mobile applications

**🡺Java features:-**

| **Feature** | **Description** |
| --- | --- |
| **Simple** | **Easy syntax, no pointers** |
| **Object-Oriented** | **Everything is based on classes & objects** |
| **Platform Independent** | **Runs on any OS via JVM** |
| **Secure** | **No pointers, sandbox security** |
| **Robust** | **Exception handling, garbage collection** |
| **Portable** | **Same code runs on any platform** |
| **High Performance** | **Uses JIT compiler for faster execution** |
| **Multithreaded** | **Multiple tasks run in parallel** |
| **Distributed** | **Supports network applications** |
| **Dynamic** | **Classes loaded at runtime** |

**JVM :-** java virtual machine 🡺 class loader subsystem + runtime data areas + execution engine + natve lbrares

**Jdk** :- java development kit 🡺 development tools + jre 🡺javc javp javaw java🡺 ivanni jdk lo unna bin folder lo untay

**Jre** :- java runtime environment🡺runtime tools + librares( java internal classes)

**Jdk 🡺jre🡺jvm**

**Jdk loo .java files 🡺 jre loki .classes ga marathay🡺jvm loki vacheysaraki byte code ga change avtundhi**

* JVM is the engine that runs Java bytecode
* It reads .class files (bytecode) and converts them to machine code (native code) using:

Interpreter (executes bytecode line by line)

JIT compiler (Just-In-Time) (converts full blocks to machine code for speed)

JVM makes Java platform-independent same bytecode runs on any OS (Windows, Linux, Mac) if JVM is installed.

**JDK (Java Development Kit) is a full toolkit for Java developers. It includes:**

* Java compiler (javac)
* Java Runtime Environment (JRE)
* Tools like jar, javadoc etc.

You need JDK to write and compile code.

**javac Hello.java for compilation.**

**This creates:  
Hello.class — which contains bytecode, not machine code**

* **JRE is needed to run Java programs (but not to compile)**
* **It includes:**
  + **JVM**
  + **Core Java libraries (like java.lang, java.util, etc.)**

**JRE = JVM + Libraries → allows bytecode to run properly.**

**What are Data Types in Java?**

**Data types** define the type of data a variable can store.  
**Example:** int, float, String, etc.

Java is a strongly typed language, so every variable must have a data type.

Java Data Types are Divided into 2 Categories:

+---------------------+

| Data Types |

+---------------------+

/ \

+-----------+ +-------------+

| Primitive | | Non-Primitive |

+-----------+ +-------------+

**1. Primitive Data Types (8 types)**

**These are the basic built-in data types. There are 8 total:**

| **Type** | **Size** | **Stores** | **Example** |
| --- | --- | --- | --- |
| **byte** | **1 byte** | **Small integers (-128 to 127)** | **byte b = 10;** |
| **short** | **2 bytes** | **Small integers** | **short s = 1000;** |
| **int** | **4 bytes** | **Whole numbers (default)** | **int x = 5000;** |
| **long** | **8 bytes** | **Large integers** | **long l = 100000L;** |
| **float** | **4 bytes** | **Decimal numbers (low precision)** | **float f = 5.6f;** |
| **double** | **8 bytes** | **Decimal numbers (high precision)** | **double d = 99.99;** |
| **char** | **2 bytes** | **Single character** | **char c = 'A';** |
| **boolean** | **1 bit** | **true or false** | **boolean b = true;** |

**Default types:**

* Integer → int
* Decimal → double
* Boolean → false
* Char → Unicode character

**2. Non-Primitive (Reference) Data Types**

**These refer to objects, not actual values.**

**Examples:**

* **String – a sequence of characters**
* **Arrays – like int[] arr = new int[5];**
* **Classes – your own custom types**
* **Interfaces**

**String name = "Akhil";**

**int[] marks = {90, 80, 70};**

**Variables**

Variables are like a storage containers that hold data values. When you create a variable you give it a specific type like int for integers or string for text.

Once you assign a value to that variable, You can use it throughout the your code and you can also change the value later if needed.

**Types of variables:-**

Base on the purpose and position of declaration all variables are divided into 3 types.

1) instance variables ==== inside class outside the method === manam access cheyyali antey object ni create cheyskuni access cheysovali

2) static variables === same instance varable laganey but static key word use cheystamu===denni manam direct ga access cheyskovachu ey method lo aina

3) local variables === ey variables inside the method untay === only aah method loney access cheyskogalamu

**Instance variables**

If the value of a variable is varied from object to object. Such type of variables are called

instance variables. For every object a separate copy of instance variables will be created.

Instance variables will be crated at the time of object creation and will be destroyed at the

time of object destruction i.e the scope of instance variables is exactly same as the scope of

object.

We have to declare instance variables with in the class but outside of any method (or)

constructor or block.

For the instance variables no need to perform initialization. JVM will always provide values.

**static variable**

if the value of a variable is fixed for all objects then it is not recommended to declare that variable at instance level. Such type of variables we have to declare at class level by using static keyword. For the static variables a single copy will be created at class level and shared by all objects of that class. Static variables should be declared with in the class but outside of any method or block or constructor. Static variables will be created at the time of class loading and destroyed at the time of unloading. We can access static variables either by using class name or by using object reference using class name is recommended.

**LocalVariables**

If we are declaring a variable with in a method or constants or block such type of variables are called local variables. Local variables also known as temporary variable or ‘automatic variable’. Local variables will be created as the part of method execution and will be destroyed once the method completes. For the local variables JVM won’t provide any default values. Before using a local variable compulsory we should perform initialization explicitly otherwise compile time error.

**What is a Method in Java?**

A **method** is a **block of code** that performs a specific task.  
You write it **once** and **use it many times**.

Think of it like a machine — you give input, it works, and gives output.

**Why Use Methods?**

* To **reuse code**
* To keep code **organized and readable**
* To perform tasks like **calculations, printing, or processing data**

We can classify Java methods into 2 main categories:

1.Pre defined methods

2.User defined methods

**Syntax:-** returntype/Non-return-type methodname (Parameters or arguments);

**Static vs non static**

* Static properties are shared and belong to the class. Only one copy is created, and they are used for common values.
* Non-static properties are unique for each object. Each object gets its own copy, and they are used for storing object-specific data.
* You access static properties using class name, and non-static properties using object name.
* Static is used when the data is same for all, non-static is used when data is different for each object.

**Real-Time Example: Student & College**

Imagine you are building a Java program for a college.

* Every **student** has their own **name** and **roll number**
* But all students study in the **same college**, like **"JNTU"**

**Non-Static Properties (Instance)**

* These are different for every student.
* Example:
  + name
  + rollNumber

Each student will have **their own name and roll number**, so these should be **non-static**.

**Static Property (Shared by All)**

* The college name is the same for all students.
* So we don’t want to store it separately for every student.
* We use a **static variable** for college name.

**Flow controls**

**Flow control** (also called **control flow**) is how a Java program **decides what to execute next** — line by line, conditionally, or repeatedly.

It controls the **order in which instructions run** in your program.

**Types of Flow Control in Java**

Flow control is mainly divided into **3 types**:

**1. Decision Making (Conditional statements)**

**2. Looping (Repetition statements)**

**3. Branching (Jump statements)**

**Decision Making (Conditional Statements)**

Used to execute different blocks of code based on conditions.

🡺 **if statement**

if (age >= 18) {

System.out.println("Eligible to vote");

}

**🡺if-else statement**

if (marks >= 50) {

System.out.println("Pass");

} else {

System.out.println("Fail");

}

**Oops concepts in java:-**

**Java is a pure object-oriented language, and it is based on 6 main OOPs concepts:**

1. Class
2. Object
3. Inheritance
4. Abstraction
5. Encapsulation
6. Polymorphism

**Class:-**

A class is like a blueprint or template. It defines how an object will look and behave — but it is not real.

Think of a class as a plan for creating objects.

Real-Life Example:

* A Car class defines properties like color, brand, speed, and methods like start(), stop().
* But this class itself is not a car — it’s just a plan to make cars.

class Car {

String brand;

int speed;

void start() {

System.out.println("Car started");

}

}

**Object:-**

An object is a real-world entity created using the class.  
It has its own values and can perform actions.

You can create many objects from a single class.

Real-Life Example:

* Car car1 = new Car();  
  Here, car1 is a real car made from the Car class.

**Number of ways to create object:-**

* Using new keyword
* Using Class.forName().newInstance() (Reflection)
* Using clone() method
* Using Deserialization
* Using Factory methods (e.g., Integer.valueOf())

**Inhertance:-**

* The process of reusing the properties from one class into another class by using predefined keyword called extends
* Whatever class will give its properties to another class then it is called either base/super/parent class.
* Whatever class will reuse or aquires the properties from some other class then it is called either derived or sub/child class.

**Types of Inheritance in Java**

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple inheritance (only we can achieve with interfaces )

**Single Inheritance**

One class inherits from another class (one parent → one child)

Here we will reuse the properties from single base class to single derived class

class Parent {

void show() {

System.out.println("Parent class");

}

}

class Child extends Parent {

void display() {

System.out.println("Child class");

}

}

**2. Multilevel Inheritance**

A class inherits from a class, which itself inherits from another class  
(grandparent → parent → child)

class GrandParent {

void method1() {

System.out.println("GrandParent");

}

}

class Parent extends GrandParent {

void method2() {

System.out.println("Parent");

}

}

class Child extends Parent {

void method3() {

System.out.println("Child");

}

}

**3. Hierarchical Inheritance**

Multiple classes inherit from the same parent class  
(one parent → many children)

class Parent {

void display() {

System.out.println("Parent class");

}

}

class Child1 extends Parent {

void method1() {

System.out.println("Child1");

}

}

class Child2 extends Parent {

void method2() {

System.out.println("Child2");

}

}

**4. Multiple Inheritance (Not supported with classes)**

Java does not support multiple inheritance with classes.

class A {

void show() {}

}

class B {

void show() {}

}

// Error: Cannot inherit from two classes

class C extends A, B {

// Not allowed

}

A combination of more than one type — achieved in Java only via **interfaces**

interface A {

void methodA();

}

interface B {

void methodB();

}

class C implements A, B {

public void methodA() {

System.out.println("From A");

}

public void methodB() {

System.out.println("From B");

}

}

**4. Abstraction**

**Abstraction** means showing **only important things** and **hiding the details**.

Helps in **focusing on what to do**, not how to do it.

**Real-Life Example:**

* When you **use an ATM**, you only press buttons — you don’t know what happens inside.
* ATM hides internal code and only shows options like **Withdraw, Balance Check**.

**5. Encapsulation**

**Encapsulation** means **hiding the data** inside a class and giving access using **getters and setters**.

Keeps data **safe and secure**.

**Real-Life Example:**

* A **TV remote** has buttons, but you can't see the circuits inside.
* You control the TV using the remote — but don’t access internal wiring.

**6. Polymorphism**

**Polymorphism** means **one thing behaving in different ways**.

There are two types:

* **Compile-time (Method Overloading)**
* **Run-time (Method Overriding)**

**Real-Life Example:**

* A **person** can be a **son at home**, **employee at office**, **friend with buddies** — same person, different roles.

Method Overloading (Same method name, different parameters)

Method overriding same method name and same parameters

**Constructor in Java**

A **constructor** is a **special method** that is **automatically called** when an object is created.  
Its job is to **initialize the object** (set starting values).

**Key Points:**

* Constructor name is always **same as the class name**
* It **does not have a return type** (not even void)
* It is called **automatically** when we create an object
* If you don’t write one, Java gives a **default constructor**

There are **2 main types** of constructors in Java:

1. **Default Constructor**
2. **Parameterized Constructor**

**1. Default Constructor**

* No arguments
* Java creates this automatically if you don’t write any constructor

class Car {

Car() {

System.out.println("Default Constructor called");

}

}

**2. Parameterized Constructor**

* Takes arguments (values) to set object data

class Student {

String name;

int age;

Student(String n, int a) {

name = n;

age = a;

}

void display() {

System.out.println(name + " - " + age);

}

}

**Constructor Overloading**

You can create **multiple constructors** with different parameters  
 This is called **constructor overloading**

**Some Keywords in java**

**This** Keyword

The **this** keyword refers to the current object of the class.  
It is used to avoid confusion when local variable names and instance variable names are the same.

* To refer to **current object’s variable**
* To call **current class method or constructor**
* To **pass current object** as a parameter

**Super** keyword

The super keyword refers to the **parent (superclass) object**.  
It is used to **access parent class variables, methods, or constructor**.

* To call **parent class method**
* To access **parent class variable**
* To call **parent class constructor** (super())

**Interface**

An interface is a fully abstract structure:

* Only contains method declarations (no body)
* All methods are public and abstract by default
* A class must use implements to use it.

we use an interface when we want to define common behaviour that multiple or unrelated classes should have.

For **example,** ifwe have a different classes like a bird and an airplane and they both need fly method and we could define a flyable interface.

A **real world example** is something like a payment processor interface.If you have multiple payment methods in an application(like credit card,pay pal or bank transfer),you can have each of them implements the payment processor interface ensuring they all have payment processor method,even though the internal implementation might differ.

**Abstract class**

**An abstract** class is a class which can contain the **abstract methods without body and concrete methods with a body** and is defined with the keyword **abstract.**

Used to achieve abstraction**.**

**An abstract class** is used when we have classes that are closely related and share common code but we still want parts of their implementation to be flexible.

In a real world example scenario for using an abstract class could be in a framework for creating different types of vehicles. You might have an abstract class called vehicle that has some implemented methods like methods like startEngine or stopengine but also abstract methods like fueltype that each specific vehicle(like a car or a motorcycle) would need to define in its own way.

**Access modifiers:-**

Access modifiers in java tells us how accessible the class members like methods,variables and constructors from other classes.

We have 4 main types:-

public, private, protected, and default (no keyword).

The most open modifier is **public**, which means the code can be accessed from anywhere—whether it's another class, another file or even another package. For **example**, if you make a method or variable **public**, it becomes **globally** available.

On the other hand, private is the most restrictive. It allows access only within the same class. This is useful when you want to hide data or methods from other classes and make your class secure. **For example, in a BankAccount class, you may want to make the balance variable private so no one else can change it directly. You can give access through public getter and setter methods instead.we can provide public methods like deposit and withdrawl.**

The protected modifier is a bit more flexible. It allows access within the same package (like default) but also in subclasses that are outside the package. This is mostly used in inheritance when you want child classes to access some members of the parent class, but still keep them hidden from unrelated classes.

If you don’t use any access modifier, Java applies the default access level. This means the member can only be accessed by classes in the same package. It’s neither public nor private—just package-level visibility. It’s useful when you are building a group of related classes and want them to work closely without exposing them to the outside world.

| **Modifier** | **Same Class** | **Same Package** | **Subclass (Different Package)** | **Other Classes** |
| --- | --- | --- | --- | --- |
| public | ✅ | ✅ | ✅ | ✅ |
| protected | ✅ | ✅ | ✅ | ❌ |
| *(default)* | ✅ | ✅ | ❌ | ❌ |
| private | ✅ | ❌ | ❌ | ❌ |

**Exception handling:-**

In Java, **exception handling** is a way to **handle errors** that happen **while the program is running**, so the program doesn't crash suddenly.

An **exception** is an event (usually an error) that **interrupts the normal flow** of a program. For example, dividing a number by zero or trying to open a file that doesn’t exist causes exceptions.

Java provides a powerful system to handle such errors using **try, catch, finally, throw, and throws** keywords.

Without exception handling, your program may crash when an error happens. With exception handling, you can:

* Handle errors gracefully
* Show user-friendly messages
* Avoid full program crashes
* Keep the application running safely

1.**try**

Used to wrap risky code (code that might cause an error).

2. **catch**

Used to handle the exception if it happens. You can have multiple catch blocks for different types of exceptions.

3. **finally**

This block always runs, whether an exception occurs or not. Used to clean up, like closing files or releasing memory.

4. **throw**

Used to manually throw an exception.

**5. throws**

Used in method signature to **declare** that a method might throw exceptions.

**Types of exceptions:**

| **Exception** | **Meaning** |
| --- | --- |
| ArithmeticException | Dividing by zero |
| NullPointerException | Accessing a null object |
| ArrayIndexOutOfBoundsException | Invalid array index |
| NumberFormatException | Wrong string to number conversion |
| IOException | Input/output operation failed |
| FileNotFoundException | File not found |

**Arrays:**

An array is a container that holds multiple values of the same type in a single variable.

Instead of creating many variables like:

**int a = 10;**

**int b = 20; 🡺 int[] numbers = {10, 20, 30};**

**int c = 30;**

**Key Points**

* Arrays store same data type values.
* The size is fixed (once created, can’t change).
* Array index starts from 0.
* Each element is accessed using the index.
* Arrays help reduce code and organize data.
* Use **loops** to access all elements.

**int[] arr; or int arr[];** 🡺 for declaring array

**arr = new int[5];** // creates an array of size 5 🡺 For memory allocation

**int[] arr = {10, 20, 30, 40, 50};**for assigning values

**arr.length** for array length or to find no of elements in an array

for (int i = 0; i < arr.length; i++) {

System.out.println(arr[i]); **for loop**

}

**Strings**

* A String is a sequence of characters enclosed in double quotes.
* Example: "Hello", "Java123", "@GPT!"
* Used to store and handle textual data.

**How to Create a String**

1. **Using string literal** (simple and preferred):

String name = "Akhil";

**2.Using new keyword**:

String city = new String("Dallas");

**Key Features of Strings**

* Strings belong to the **java.lang.String class**.
* Strings are **immutable** (once created, they cannot be changed).
* Strings are handled as **objects**, not primitive types.

**Common String Methods**

1. length() – Returns the number of characters.

"Java".length(); // 4

2.charAt(index) – Returns character at the given index.

"Java".charAt(0); // 'J'

3.toUpperCase() / toLowerCase() – Converts case.

"java".toUpperCase(); // "JAVA"

4. equals() – Compares strings (case-sensitive).

"Java".equals("Java"); // true

5. equalsIgnoreCase() – Compares strings ignoring case.

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

6. concat() – Joins two strings.

"Hello".concat(" World"); // "Hello World"

7.substring(start, end) – Extracts part of string.

"Hello".substring(1, 4); // "ell"

8. contains() – Checks if string contains a part.

"Welcome".contains("come"); // true

**Immutability of Strings**

* Strings cannot be changed once created.
* Any operation that modifies a string creates a **new string object**.

String s = "Hello";

s = s + " World"; // New object "Hello World" created

**Mutable Alternative: StringBuilder**

* Use **StringBuilder** or **StringBuffer** to modify strings.
* These are **mutable** and faster for many edits.

StringBuilder sb = new StringBuilder("Hi");

sb.append(" there");

System.out.println(sb); // "Hi there"

**Multithreading**

* **Multithreading** means **running two or more parts of a program at the same time**.
* Each part is called a **thread**, and each thread runs **independently**.
* Java supports multithreading to improve **performance** and **responsiveness** of applications.

**Why Use Multithreading?**

* To do multiple tasks at the same time (like downloading a file and updating the progress bar).
* To make programs faster and more efficient.
* Useful in games, servers, animations, real-time apps, etc.

**What is a Thread?**

* A **thread** is a **lightweight** unit of a process.
* One Java program (called a process) can have many threads running inside it.
* Example: Watching a YouTube video (one thread plays video, another downloads, another plays audio).

**Two ways to create threads: extends Thread or implements Runnable.**

| **Method** | **Description** |
| --- | --- |
| **start()** | **Starts the thread and calls run()** |
| **run()** | **Code that thread will execute** |
| **sleep(ms)** | **Pauses thread for given milliseconds** |
| **join()** | **Waits for thread to finish** |
| **isAlive()** | **Checks if thread is still running** |

**Thread Life Cycle**

1. **New – Thread created**
2. **Runnable – Ready to run**
3. **Running – Currently running**
4. **Blocked/Waiting – Temporarily paused**
5. **Terminated – Finished**

**Synchronization (Handling Conflicts)**

* When multiple threads access the same data, it can cause issues (called race conditions).
* To prevent this, Java provides synchronization using **synchronized** keyword.

**synchronized void print() {**

**// only one thread can enter this block at a time**

**}**