**Variable:**

A **variable** is a container that holds data. It has a **name**, a **type**, and a **value**.

**Real-Life Example of Variables: Coffee Machine ☕**

Imagine you have a **coffee machine**. It has different components like:

* **Water Tank** (stores water)
* **Coffee Beans Container** (stores coffee beans)
* **Sugar Container** (stores sugar)

Each of these components **stores some values**, just like variables in Java

**Syntax of Variable Declaration**

dataType variableName = value;

Example :-

int age = 20; // Stores an integer value

double price = 99.99; // Stores decimal values

char grade = 'A'; // Stores a single character

String name = "Gauri"; // Stores a sequence of characters

boolean isJavaFun = true; // Stores true or false

**🔴 Types of Variables in Java (With Range & Size)**

In Java, variables are classified into **3 types** based on **scope** and **usage**:

1. **Local Variables**
2. **Instance Variables (Non-Static Variables)**
3. **Static Variables (Class Variables)**

Additionally, based on **data type**, variables are divided into **Primitive** and **Non-Primitive**.

**1️. Local Variables**

* **Defined inside a method, constructor, or block**.
* Only accessible within that **method/block** (temporary storage).
* Must be **initialized before use**.

public class Example {

public static void main(String[] args) {

int age = 25; // Local variable (inside method)

System.out.println("Age: " + age);

}

}

**Key Point:** Cannot access age outside main() method.

**2️.Instance Variables (Non-Static Variables)**

* Declared **inside a class but outside methods**.
* Each object gets its **own copy** of instance variables.
* Initialized to **default values** if not assigned.

class Student {

String name; // Instance variable

int age; // Instance variable

public void display() {

System.out.println("Name: " + name + ", Age: " + age);

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student();

s1.name = "Gauri";

s1.age = 22;

s1.display();

}

}

**🔹 Key Point:** Each Student object will have its own name and age.

**3️.Static Variables (Class Variables)**

* Declared using static keyword.
* Shared among **all objects** of the class (single memory allocation).
* Can be accessed using **class name**.

✅ **Example:**

class Company {

static String companyName = "Google"; // Static variable

}

public class Main {

public static void main(String[] args) {

System.out.println("Company: " + Company.companyName);

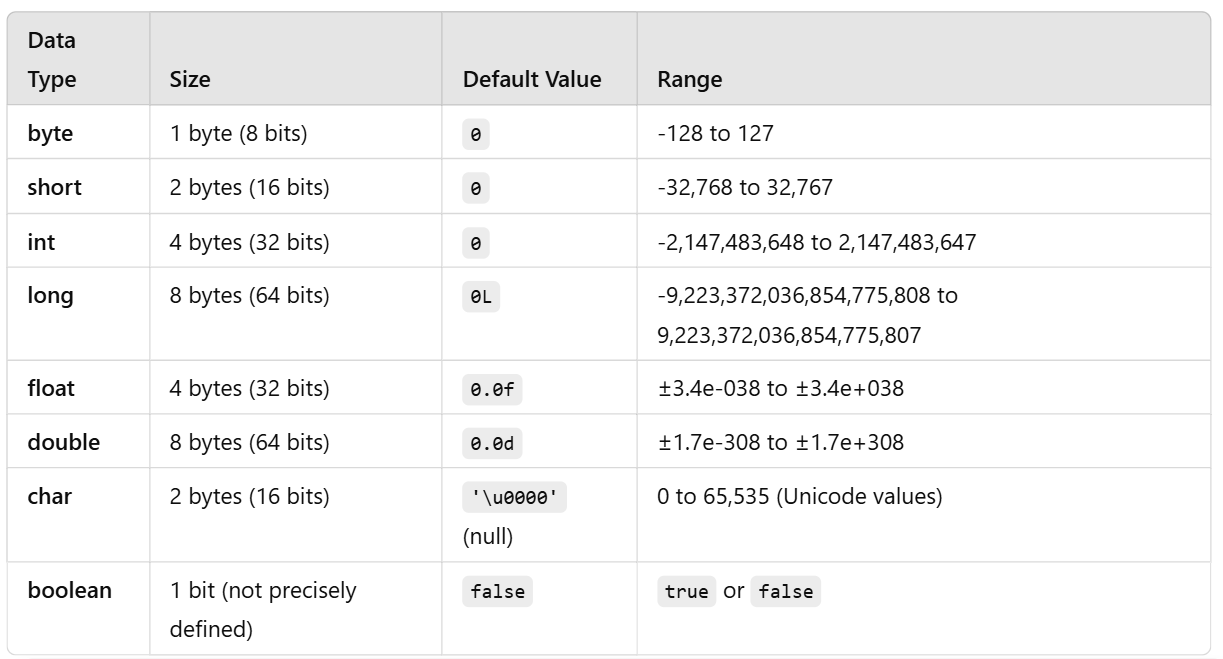
}

}

**🔹 Key Point:** Same companyName is shared across all instances.

**🔷 Primitive Data Types in Java (Size & Range)**

Java has 8 primitive data types, each with a specific size and range.



public class DataTypesExample {

public static void main(String[] args) {

byte smallNum = 100;

int largeNum = 50000;

long bigNum = 10000000000L;

float decimalNum = 3.14f;

double preciseDecimal = 3.1415926535;

char grade = 'A';

boolean isJavaFun = true;

System.out.println("Byte: " + smallNum);

System.out.println("Integer: " + largeNum);

System.out.println("Long: " + bigNum);

System.out.println("Float: " + decimalNum);

System.out.println("Double: " + preciseDecimal);

System.out.println("Character: " + grade);

System.out.println("Boolean: " + isJavaFun);

}

}

**🔷 Non-Primitive Data Types**

These are more complex than primitive types and **can store multiple values**.



✅ **Example Using Non-Primitive Data Types:**

public class NonPrimitiveExample {

public static void main(String[] args) {

String name = "Java"; // String

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

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

System.out.println("First Number: " + numbers[0]);

}

}

Sure! Let's break down **public static void main(String args[])** in the **simplest way** using **real-life examples** 🚀

**💡 Think of it Like This:**

Imagine you are starting a **new project in a company**. You need a **main manager** to begin the work. In Java, the main method is that **main manager** that starts the program.

**🔎 Breaking it Down - Word by Word**

| **Java Keyword** | **Real-Life Meaning** | **Simple Explanation** |
| --- | --- | --- |
| public | 🔓 **Public Door** | Anyone can enter (accessible by everyone). |
| static | 🏢 **Company Rule** | No need to create an employee (object), anyone can follow the rule. |
| void | 📬 **No Return Parcel** | The function does not return anything back. |
| main | 🏁 **Main Manager** | The starting point of the project (program). |
| String args[] | 📦 **Command Line Inputs** | Information given to the program when starting (not always needed). |

**📌 Real-Life Example**

**Imagine a Company Starting Work**

1️⃣ **public** → The **company building** is **open to everyone** (anyone can enter).  
2️⃣ **static** → There is a **company rule** that applies to **everyone** (no need to hire a special person).  
3️⃣ **void** → The company **doesn’t send** any **parcel (return value)** after starting work.  
4️⃣ **main** → The **main manager** who starts the **daily work**.  
5️⃣ **String args[]** → If the manager wants to receive **extra instructions** from the **CEO** (command-line arguments).

**📌 Java Code + Real-Life Connection**

public class Company {

public static void main(String args[]) {

System.out.println("Company Work Started!");

}

}

✅ **Output:**

Company Work Started!

🔹 main() starts the work like a **manager** in a company.

**🔹 Deep Dive Into Each Word (with Example)**

**🔴 1️.public → Open for Everyone**

public class Bank {

public static void main(String args[]) {

System.out.println("Bank is Open to Everyone");

}

}

✅ **Output:**

Bank is Open to Everyone

📌 public means **anyone can access** the main method.

**🔵 2️.static → No Need to Create an Object**

public class Calculator {

public static void main(String args[]) {

System.out.println("Calculator is working!");

}

}

✅ **Output:**

Calculator is working!

📌 static means we **don't need to create an object** to run main().

**🟢 3️.void → No Return Value**

public class Message {

public static void main(String args[]) {

System.out.println("This method does not return anything.");

}

}

✅ **Output:**

This method does not return anything.

📌 void means this method **does not return any value**.

**🟣 4️.main → The Starting Point**

public class Start {

public static void main(String args[]) {

System.out.println("Main Method Started!");

}

}

✅ **Output:**

Main Method Started!

📌 main is the **starting point of the program**.

**🟠 5️.String args[] → Command Line Arguments**

public class Greeting {

public static void main(String args[]) {

System.out.println("Hello, " + args[0] + "!");

}

}

📌 Run the program with:

java Greeting Gauri

✅ **Output:**

Hello, Gauri!

📌 args[] takes **extra inputs** from the user when running the program.

**🎯 Summary Table**

|  |  |  |
| --- | --- | --- |
| Java Keyword | Meaning | Real-Life Example |
| public | **Accessible by everyone** | A **shop** that is open for all customers. |
| static | **Can be used without creating an object** | **Electricity** is available in every house **without a special request**. |
| void | **Does not return anything** | A **robot** that only works but **does not give anything back**. |
| main | **Starting point** | The **CEO** who **starts the company** daily. |
| String args[] | **Command-line inputs** | Giving **extra instructions** to a **manager** before starting work. |

**🚀 Final Thoughts**

* public static void main(String args[]) is **mandatory** in Java programs.
* It is the **starting point** where execution **begins**.
* Each word has **special meaning** and **importance**.

**✅ Arrays in Java (Beginner to Interview Level)**

An **array** is a collection of elements of the **same data type**, stored in **contiguous memory locations**.

**🔹 Why Use Arrays?**

* Instead of declaring **multiple variables** (like int a, b, c;), we can store multiple values **in a single array**.
* Arrays provide **faster access** to data using an **index**.
* Useful when we have **a large number of elements** (like processing student marks, temperatures, etc.).

**🔹 Declaring & Initializing an Array**

// 1️.Declaration

int[] numbers; // Preferred way

// OR

int numbers[]; // Also valid

// 2️.Allocation (Memory Reserved)

numbers = new int[5]; // An array of size 5 is created

// 3️.Initialization

numbers[0] = 10;

numbers[1] = 20;

numbers[2] = 30;

numbers[3] = 40;

numbers[4] = 50;

**📌 Shortcut: Declare and Initialize in One Step**

int[] numbers = {10, 20, 30, 40, 50}; // Array of size 5

**🔹 Accessing Array Elements**

System.out.println(numbers[0]); // Prints 10

System.out.println(numbers[3]); // Prints 40

**🔹 Iterating Over an Array (Using Loops)**

**✅ Using a for loop**

int[] numbers = {10, 20, 30, 40, 50};

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

System.out.println(numbers[i]);

}

**✅ Using an Enhanced for-each loop**

for(int num : numbers) {

System.out.println(num);

}

👉 **Advantage:** No need to use an index.

**🔹 Types of Arrays**

**1️.Single-Dimensional Array**

* A simple list of values.

int[] marks = {85, 90, 78, 88, 76};

**2️.Multi-Dimensional Array (2D Array)**

* Used for storing **table-like data**.

int[][] matrix = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

✅ **Accessing 2D Array Elements**

System.out.println(matrix[0][1]); // Prints 2

System.out.println(matrix[2][2]); // Prints 9

**🔹 Common Interview Questions**

1️.**What is the default value of an array in Java?**  
✅ **Answer:**

* **For int array** → 0
* **For float array** → 0.0
* **For boolean array** → false
* **For String or Object array** → null

2️.**How to find the length of an array?**  
✅ **Answer:**

int[] arr = {10, 20, 30, 40};

System.out.println(arr.length); // Output: 4

3️.**How to copy an array in Java?**  
✅ **Answer:**  
Using Arrays.copyOf() method:

int[] arr1 = {1, 2, 3, 4, 5};

int[] arr2 = Arrays.copyOf(arr1, arr1.length);

System.out.println(Arrays.toString(arr2)); // Output: [1, 2, 3, 4, 5]

**🔴 Exercise 1: Find Maximum in an Array**

**👉 Write a Java program to find the largest number in an array.**

public class MaxArray {

public static void main(String[] args) {

int[] numbers = {10, 25, 30, 5, 90, 70};

int max = numbers[0]; // Assume first element is max

for(int i = 1; i < numbers.length; i++) {

if(numbers[i] > max) {

max = numbers[i]; // Update max if a larger number is found

}

}

System.out.println("Maximum number is: " + max);

}

}

✅ **Output:**

Maximum number is: 90

**🔴 Exercise 2: Reverse an Array**

**👉 Write a Java program to reverse an array without using another array.**

public class ReverseArray {

public static void main(String[] args) {

int[] numbers = {1, 2, 3, 4, 5};

System.out.println("Original Array:");

for(int num : numbers) {

System.out.print(num + " ");

}

// Reverse Logic

int start = 0, end = numbers.length - 1;

while(start < end) {

int temp = numbers[start];

numbers[start] = numbers[end];

numbers[end] = temp;

start++;

end--;

}

System.out.println("\nReversed Array:");

for(int num : numbers) {

System.out.print(num + " ");

}

}

}

✅ **Output:**

Original Array:

1 2 3 4 5

Reversed Array:

5 4 3 2 1

**✅ Strings in Java (Beginner to Interview Level)**

A **string** in Java is a **sequence of characters** enclosed in double quotes (" "). It is an **immutable** object, meaning once created, its value **cannot be changed**.

**🔹 Declaring & Initializing Strings**

// Using String literal (Recommended)

String name = "Java";

// Using new keyword (Not Recommended)

String name2 = new String("Java");

✅ **Difference:**

* **String literals** are stored in the **String Pool** for memory optimization.
* new String("Java") creates a **new object in heap memory**, even if "Java" already exists.

**🔹 Common String Methods**

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Description | Example | Output |
| .length() | Returns length of string | "Java".length() | 4 |
| .charAt(index) | Returns character at index | "Java".charAt(1) | 'a' |
| .toUpperCase() | Converts to uppercase | "java".toUpperCase() | "JAVA" |
| .toLowerCase() | Converts to lowercase | "JAVA".toLowerCase() | "java" |
| .trim() | Removes leading & trailing spaces | " Java ".trim() | "Java" |
| .substring(start, end) | Extracts part of string | "Hello".substring(1, 4) | "ell" |
| .contains("str") | Checks if a string contains another | "Java".contains("av") | true |
| .indexOf('c') | Returns index of character | "Java".indexOf('v') | 2 |
| .replace('a', 'o') | Replaces characters | "Java".replace('a', 'o') | "Jovo" |
| .equals("str") | Compares strings (case-sensitive) | "Java".equals("java") | false |
| .equalsIgnoreCase("str") | Compares strings (case-insensitive) | "Java".equalsIgnoreCase("java") | true |

**🔹 String Concatenation**

String firstName = "Gauri";

String lastName = "Chiddarwar";

String fullName = firstName + " " + lastName;

System.out.println(fullName);

✅ **Output:**

Gauri Chiddarwar

🔹 **Using concat()**

String fullName = firstName.concat(" ").concat(lastName);

System.out.println(fullName);

**🔹 String Comparison**

**✅ 1. Using == (Reference Comparison)**

String s1 = "Java";

String s2 = "Java";

System.out.println(s1 == s2); // true (Same memory reference)

String s3 = new String("Java");

System.out.println(s1 == s3); // false (Different memory reference)

**✅ 2. Using .equals() (Content Comparison)**

System.out.println(s1.equals(s3)); // true (Same content)

**✅ 3. Using .compareTo() (Lexicographical Comparison)**

String s1 = "Apple";

String s2 = "Banana";

System.out.println(s1.compareTo(s2)); // -1 (Apple comes before Banana)

System.out.println(s2.compareTo(s1)); // 1 (Banana comes after Apple)

System.out.println(s1.compareTo("Apple")); // 0 (Same string)

**🔹 String Immutability (Why String is Immutable?)**

String str = "Hello";

str.concat(" World");

System.out.println(str); // "Hello" (String is not changed)

✅ **Explanation:**

* concat(" World") creates a **new string**, but str still refers to "Hello".
* This **prevents accidental changes** and **improves performance** (String Pool optimization).

**🔹 StringBuilder & StringBuffer (Mutable Strings)**

If you need **mutable** (modifiable) strings, use **StringBuilder** or **StringBuffer**.

**✅ 1. Using StringBuilder (Faster)**

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

System.out.println(sb); // "Hello World"

**✅ 2. Using StringBuffer (Thread-Safe)**

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World");

System.out.println(sb); // "Hello World"

**📌 Difference:**

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | String | StringBuilder | StringBuffer |
| Mutable? | ❌ No | ✅ Yes | ✅ Yes |
| Thread-Safe? | ✅ Yes | ❌ No | ✅ Yes |
| Performance | ❌ Slow | ✅ Fast | 🔸 Medium |

**🔹 Interview Questions on Strings**

1️.**Why is String immutable in Java?**  
✅ **Answer:**

* For **security** (used in passwords, database connections).
* **Thread-safety** (multiple threads can use strings safely).
* **String Pool optimization** (reduces memory usage).

2️.**How to reverse a string in Java?**  
✅ **Answer:**

public class ReverseString {

public static void main(String[] args) {

String str = "Java";

String reversed = new StringBuilder(str).reverse().toString();

System.out.println("Reversed: " + reversed);

}

}

✅ **Output:**

Reversed: avaJ

3️.**How to check if a string is a palindrome?**  
✅ **Answer:**

public class PalindromeCheck {

public static void main(String[] args) {

String str = "madam";

String reversed = new StringBuilder(str).reverse().toString();

if(str.equals(reversed)) {

System.out.println("Palindrome");

} else {

System.out.println("Not a Palindrome");

}

}

}

✅ **Output:**

Palindrome

4️.**How to count vowels and consonants in a string?**  
✅ **Answer:**

public class CountVowels {

public static void main(String[] args) {

String str = "Hello World";

str = str.toLowerCase();

int vowels = 0, consonants = 0;

for(char ch : str.toCharArray()) {

if(ch >= 'a' && ch <= 'z') { // Ignore spaces & special characters

if("aeiou".indexOf(ch) != -1) {

vowels++;

} else {

consonants++;

}

}

}

System.out.println("Vowels: " + vowels);

System.out.println("Consonants: " + consonants);

}

}

✅ **Output:**

Vowels: 3

Consonants: 7

**🔴 Exercise 1: Find the First Non-Repeating Character**

👉 **Write a program to find the first unique character in a string.**  
🔹 **Example Input:** "programming"  
🔹 **Expected Output:** 'p'

**🔴 Exercise 2: Check if Two Strings are Anagrams**

👉 **Write a program to check if two strings are anagrams.**  
🔹 **Example:** "listen" & "silent" → ✅ **Anagrams**

**✅ Object-Oriented Programming (OOPs) in Java**

**Object-Oriented Programming (OOP)** is a programming paradigm based on the concept of **objects** that contain **data** (fields) and **methods** (functions). It helps in writing modular, reusable, and scalable code.

**🔹 4 Pillars of OOPs**

| **Concept** | **Definition** | **Real-Life Example** |
| --- | --- | --- |
| **Encapsulation** | Wrapping data and methods into a single unit (class) | Medicine capsule (hides the drug inside) |
| **Abstraction** | Hiding implementation details and exposing only essential features | Driving a car (you don't need to know the engine mechanism) |
| **Inheritance** | Acquiring properties and behavior from a parent class | Child inheriting traits from parents |
| **Polymorphism** | The ability to take multiple forms (method overloading & overriding) | A person behaving differently in different situations (at home vs. office) |

**🔴 1. Encapsulation (Data Hiding)**

**✅ Definition:**

Encapsulation means **binding the data (variables) and code (methods)** together within a class and **restricting direct access** to some components.

**✅ Key Points:**

✔ Data members are **private**.  
✔ Access is provided using **public getter & setter methods**.

**✅ Example: Encapsulation in Java**

class BankAccount {

private double balance; // Private variable (data hiding)

// Constructor

public BankAccount(double initialBalance) {

balance = initialBalance;

}

// Getter Method

public double getBalance() {

return balance;

}

// Setter Method

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

}

}

}

// Main Class

public class EncapsulationExample {

public static void main(String[] args) {

BankAccount account = new BankAccount(5000);

account.deposit(1500);

System.out.println("Balance: " + account.getBalance()); // Output: Balance: 6500

}

}

✅ **Why Encapsulation?**  
✔ Protects data from **unauthorized access**.  
✔ **Improves code maintainability**.  
✔ Allows **controlled modifications**.

**🔴 2. Abstraction (Hiding Implementation)**

**✅ Definition:**

Abstraction is **hiding implementation details** and exposing only the necessary functionality.

✅ **Example in Real Life:**

* **ATM Machine** – You just insert a card and enter the PIN (you don’t need to know how internal banking works).
* **Car** – You just drive without worrying about how the engine works.

**✅ Abstract Class Example:**

abstract class Vehicle {

abstract void start(); // Abstract method (no implementation)

}

class Car extends Vehicle {

@Override

void start() {

System.out.println("Car is starting with a key...");

}

}

class Bike extends Vehicle {

@Override

void start() {

System.out.println("Bike is starting with a self-start...");

}

}

// Main Class

public class AbstractionExample {

public static void main(String[] args) {

Vehicle myCar = new Car();

myCar.start(); // Output: Car is starting with a key...

}

}

✅ **Why Abstraction?**  
✔ Reduces code complexity.  
✔ Focuses on **what** an object does rather than **how**.

**🔴 3. Inheritance (Code Reusability)**

**✅ Definition:**

Inheritance allows a **child class** to acquire the properties & behaviors of a **parent class**.

✅ **Example in Real Life:**

* A **child inherits** the traits (height, eye color) from **parents**.

**✅ Types of Inheritance in Java**

| **Type** | **Example** |
| --- | --- |
| **Single Inheritance** | A → B |
| **Multilevel Inheritance** | A → B → C |
| **Hierarchical Inheritance** | A → {B, C} |
| **Multiple Inheritance (via Interface)** | A, B → C |

**✅ Example: Single Inheritance**

class Animal {

void eat() {

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

}

}

class Dog extends Animal {

void bark() {

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

}

}

// Main Class

public class InheritanceExample {

public static void main(String[] args) {

Dog d = new Dog();

d.eat(); // Inherited from Animal

d.bark(); // Defined in Dog

}

}

✅ **Why Inheritance?**  
✔ Promotes **code reusability**.  
✔ Improves **code organization**.

**🔴 4. Polymorphism (Many Forms)**

**✅ Definition:**

Polymorphism allows a **single interface** to be used for **different types of actions**.

✅ **Example in Real Life:**

* A **person behaves differently** as a student, employee, and friend.

**✅ Types of Polymorphism**

| **Type** | **Definition** | **Example** |
| --- | --- | --- |
| **Method Overloading** | Same method name, different parameters | add(int, int) & add(int, int, int) |
| **Method Overriding** | Child class redefines parent class method | Car.start() overrides Vehicle.start() |

**✅ Example: Method Overloading**

class MathOperations {

int add(int a, int b) {

return a + b;

}

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

return a + b + c;

}

}

// Main Class

public class PolymorphismExample {

public static void main(String[] args) {

MathOperations obj = new MathOperations();

System.out.println(obj.add(5, 10)); // Output: 15

System.out.println(obj.add(5, 10, 15)); // Output: 30

}

}

**✅ Example: Method Overriding**

class Animal {

void makeSound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("Dog barks");

}

}

// Main Class

public class OverridingExample {

public static void main(String[] args) {

Animal myDog = new Dog();

myDog.makeSound(); // Output: Dog barks

}

}

✅ **Why Polymorphism?**  
✔ **Increases flexibility** in code.  
✔ **Reduces code duplication**.

**🔥 OOPs Interview Questions**

**1️⃣ What is the difference between abstraction and encapsulation?**

| **Feature** | **Abstraction** | **Encapsulation** |
| --- | --- | --- |
| **Definition** | Hiding implementation details | Data hiding using private access |
| **Purpose** | Show only essential details | Restrict direct access |
| **Example** | abstract class, interface | private variables with getter/setter |

**2️⃣ Why is Java not 100% OOP?**

✅ **Answer:** Java uses **primitive data types (int, char, etc.)** which are not objects.

**3️⃣ Can a constructor be final in Java?**

✅ **Answer:** No, because constructors **cannot be inherited**, so there is no need to make them final.

**4️⃣ What is the difference between this and super?**

| **Keyword** | **this** | **super** |
| --- | --- | --- |
| **Used for** | Current class | Parent class |
| **Calls** | Constructor/method of the same class | Constructor/method of the parent class |

**✅ Next Step: Start with Java Classes & Objects in Detail?**

Would you like to continue with **real-life examples of classes & objects**, or solve **OOP-based exercises** first? 🚀

Great! Let's dive into **Classes & Objects** in Java with real-life examples and exercises. 🚀

**🔹 Understanding Classes & Objects**

**✅ Definition:**

* **Class**: A blueprint or template for creating objects.
* **Object**: An instance of a class that holds actual data.

✅ **Real-Life Example:**  
Think of a **class** as a "Car Model" and an **object** as a "Specific Car" based on that model.

**Example:**

| **Car Model (Class)** | **Actual Car (Object)** |
| --- | --- |
| Toyota Corolla | Red Corolla with Number Plate MH-12-3456 |
| Tesla Model S | White Tesla with Number Plate KA-09-7890 |

**🔴 1. Creating a Class and Object**

**✅ Example:**

class Car {

// Data Members (Attributes)

String brand;

int speed;

// Method (Behavior)

void display() {

System.out.println("Car Brand: " + brand);

System.out.println("Speed: " + speed + " km/h");

}

}

// Main Class

public class CarExample {

public static void main(String[] args) {

Car myCar = new Car(); // Creating an Object

myCar.brand = "Toyota";

myCar.speed = 120;

myCar.display(); // Calling Method

}

}

✅ **Output:**

Car Brand: Toyota

Speed: 120 km/h

✅ **Key Points:** ✔ **Class**: Defines attributes (brand, speed) and behavior (display()).  
✔ **Object**: Created using new.  
✔ **Accessing Data Members**: Using . operator (myCar.brand = "Toyota";).

**🔴 2. Constructors (Special Method to Initialize Objects)**

✅ **A constructor is a method that gets called automatically when an object is created.**

**✅ Example: Constructor in Java**

class Student {

String name;

int age;

// Constructor

Student(String studentName, int studentAge) {

name = studentName;

age = studentAge;

}

// Method

void display() {

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

System.out.println("Age: " + age);

}

}

// Main Class

public class ConstructorExample {

public static void main(String[] args) {

Student s1 = new Student("Gauri", 22); // Constructor Called

s1.display();

}

}

✅ **Output:**

Name: Gauri

Age: 22

✅ **Why Use a Constructor?** ✔ Automatically initializes values.  
✔ Reduces manual work (no need for setter methods).

**🔴 3. Types of Constructors**

| **Type** | **Definition** | **Example** |
| --- | --- | --- |
| **Default Constructor** | No parameters, assigns default values | Student() {} |
| **Parameterized Constructor** | Takes parameters to initialize values | Student(String name, int age) {} |
| **Copy Constructor** | Creates a new object by copying an existing one | Student(Student obj) {} |

✅ **Example: Default Constructor**

class Person {

String name;

// Default Constructor

Person() {

name = "Unknown";

}

void display() {

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

}

}

public class DefaultConstructor {

public static void main(String[] args) {

Person p1 = new Person();

p1.display(); // Output: Name: Unknown

}

}

**🔴 4. this Keyword (Referring to Current Object)**

✅ **Used to refer to the current object's attributes/methods.**

**✅ Example:**

class Employee {

String name;

// Constructor

Employee(String name) {

this.name = name; // Referring to the current object's name variable

}

void display() {

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

}

}

public class ThisExample {

public static void main(String[] args) {

Employee e1 = new Employee("Rohan");

e1.display();

}

}

✅ **Output:**

Employee Name: Rohan

✔ **this.name** differentiates between local and instance variables.

**🔴 5. Static Keyword (Class-Level Variables & Methods)**

**✅ Definition:**

* **Static variables** belong to the **class**, not individual objects.
* **Static methods** can be called without creating an object.

**✅ Example:**

class Company {

static String companyName = "Infosys"; // Static Variable

static void showCompany() { // Static Method

System.out.println("Company: " + companyName);

}

}

public class StaticExample {

public static void main(String[] args) {

Company.showCompany(); // No need to create an object

}

}

✅ **Output:**

Company: Infosys

✔ **Why static?**  
✔ Saves memory (single copy for all objects).  
✔ Can be accessed without an object.

**🔴 Interview Questions (Classes & Objects)**

**1️⃣ What is the difference between a class and an object?**

| **Feature** | **Class** | **Object** |
| --- | --- | --- |
| **Definition** | A blueprint/template | A real-world instance |
| **Memory** | No memory allocated | Memory allocated when created |
| **Example** | Car (model) | myCar = new Car() (actual car) |

**2️⃣ Can we have a constructor in an abstract class?**

✅ **Yes!** An abstract class can have a constructor, but it cannot be instantiated directly.

**3️⃣ Can we overload a constructor in Java?**

✅ **Yes!** Java supports **constructor overloading** by changing the number/types of parameters.

✔ **Example:**

class Student {

String name;

int age;

// Constructor Overloading

Student() { // Default Constructor

name = "Unknown";

age = 0;

}

Student(String n, int a) { // Parameterized Constructor

name = n;

age = a;

}

void display() {

System.out.println("Name: " + name + ", Age: " + age);

}

}

public class ConstructorOverload {

public static void main(String[] args) {

Student s1 = new Student(); // Calls Default Constructor

Student s2 = new Student("Gauri", 22); // Calls Parameterized Constructor

s1.display();

s2.display();

}

}

✅ **Output:**

Name: Unknown, Age: 0

Name: Gauri, Age: 22

**🔴 Exercise: Practice Questions**

**✅ Beginner-Level**

1️⃣ **Create a Book class with attributes: title, author, price. Add a constructor and a method to display details.**  
2️⃣ **Create a BankAccount class with a balance. Implement deposit & withdraw methods.**

**✅ Interview-Level**

3️⃣ **Can a Java class have multiple constructors? If yes, write an example.**  
4️⃣ **Can a class have both static and non-static methods? Explain.**

**✅ Next Step: Move to Inheritance or Solve More OOP Exercises? 🚀**