**CORE OOPs CONCEPTS**

1. **Class and Object**
2. **Encapsulation**
3. **Inheritance**
   * Single
   * Multilevel
   * Hierarchical
   * (Interface-based) Multiple
   * Hybrid (via Interface)
4. **Polymorphism**
   * Compile-time (Method Overloading)
   * Run-time (Method Overriding)
5. **Abstraction**
6. **Interface vs Abstract Class**
7. **Constructors and Constructor Overloading**
8. **Destructors (C++, not Java)**
9. **Method Overloading vs Overriding**
10. **Access Modifiers** – private, public, protected, (default in Java)

**✅ ADVANCED OOPs CONCEPTS**

1. **'this' and 'super' keywords**
2. **Constructor Chaining**
3. **Static vs Non-static**
4. **Final Keyword (Java)** / **Const Keyword (C++)**
5. **Inner Classes (Static, Non-static, Anonymous, Local)**
6. **Covariant Return Types**
7. **Instance Initialization Block / Static Block**
8. **Object Class Methods**

* equals()
* hashCode()
* toString()
* clone()

1. **Immutable Classes**
2. **Deep Copy vs Shallow Copy**
3. **Cloning / Copy Constructors**
4. **Upcasting and Downcasting**
5. **Diamond Problem**
6. **Multiple Inheritance Handling in Java (Interfaces)**
7. **Enum in OOP (Java-style enums)**
8. **Instanceof Operator**

**JAVA SPECIFIC QUESTIONS:**

1. **Garbage Collection (GC Roots, finalize())**
2. **Wrapper Classes and Autoboxing**
3. **Serialization and Deserialization**
4. **Transient Keyword**
5. **Volatile Keyword**
6. **Java Memory Model (Heap, Stack, etc.)**
7. **Thread-local Variables**
8. **ClassLoaders and Class Loading Mechanism**
9. **Reflection API**
10. **Access Modifiers and their Scope**
11. **Static Initialization Blocks**
12. **Finalize Method (deprecated)**
13. **Enum Classes**
14. **constant folding** or **interning**.
15. **method hiding**.

**Java 8 Features**:

* Lambda Expressions
* Streams API
* Functional Interfaces

**14.Optional Class**

**15.Concurrency (Executor Service, Callable, Future, Thread Pool)**

**16.Thread Safety and Synchronization**

**CROSS-LANGUAGE COMPARISON (Useful for Interviews)**

17.Differences in OOP across Java, Python, C++, C

18.Why Java is not 100% OOP (due to primitives)

19.Why Python is multi-paradigm

20.Why C++ supports procedural + OOP

21.Why C is not OOP

**Language-Specific Questions**

(Depending on which language you are using: Java, C++, Python, etc.)

* Java: JVM Internals, JIT Compiler, Collections Framework (ArrayList, HashMap, etc.)
* C++: Pointers, Memory leaks, Virtual Functions, Pure Virtual Functions
* Python: Duck Typing, Dynamic Typing, Decorators, Magic Methods

**programming paradigm** is a style or way of programming that defines how problems are solved and how programs are structured

**1.What is OOP (Object-Oriented Programming)?**

OOP is a programming paradigm that organizes software design around objects rather than functions and logic.

**Example**: In OOP, we represent real-world entities (like **Car**, **Person**, or **BankAccount**) as objects. These objects have attributes (data) and behaviors (functions or methods) that model real-world interactions.

**Why do we need OOP?**

| **Without OOP (Traditional Programming)** | **With OOP** |
| --- | --- |
| Programs become large and messy | Programs are organized around Objects |
| Difficult to manage code | Easy to manage, scale and reuse code |
| Hard to update or add new features | Easy to add new features by extending classes |
| Same code copied multiple times | Reusability through Inheritance and Polymorphism |
| Example: C language | Example: Java, C++, Python (OOP support) |

Without OOP → you write functions separately: addProduct(), removeProduct(), calculateBill() — no connection between them.

each function operates on global variables or input parameters. There is no clear structure or encapsulation of data and behavior. All the logic is written in a sequence, and managing large-scale applications becomes difficult as the code grows.

Example for programm without oops:

Is given in next page:explanation is given here

**Issues without OOP:**

1. **No Structure**: Functions (addProduct(), removeProduct(), etc.) are separate, with no logical connection between them.
2. **Data Management**: There are global variables (products, cart), making it hard to manage the data and state.
3. **Difficulty in Extension**: Adding new features like discounts, product variants, or inventory management would involve modifying existing functions, making the code messy and hard to maintain.
4. **No Encapsulation**: There's no separation of concerns between the data (products, cart) and behavior (functions to modify or calculate data).

ndA screen shot of a computer program

AI-generated content may be incorrect.

**With OOP (Object-Oriented Approach)**

In the OOP approach, we use objects to represent real-world entities like Product, Cart, and User. Each object has its own **attributes** (data) and **methods** (behavior). The code becomes modular, easier to extend, and more maintainable.

A screen shot of a computer program

AI-generated content may be incorrect.

**Advantages of OOP:**

1. **Clear Structure**: We have separate classes for Product, Cart, and User, each handling its own data and behavior. Each object encapsulates both attributes (like name, price) and behaviors (like add\_product, remove\_product).
2. **Encapsulation**: Each class has its own internal state, and data is protected from unauthorized access by methods (e.g., add\_product(), remove\_product(), calculate\_bill()).
3. **Modularity and Maintainability**: Code is modular. If we need to add a new feature (e.g., product discounts), we can easily extend the Product or Cart class without affecting other parts of the system.
4. **Scalability**: As the app grows, it's easy to add new objects (e.g., Payment, Order) and behaviors (e.g., payment methods, order history). We just add or extend classes without disrupting existing functionality.
5. **Reusability**: The Product class is reusable, and if we need to create a new Product instance, we can do so easily. The Cart class can be used for multiple users.

**How does OOP help in building a project?**

**OOP makes project building better by:**

* **Dividing problem into small parts (Objects)**
* **Protecting data** (using private variables + methods)
* **Reusing code** (through Inheritance)
* **Changing behavior easily** (through Polymorphism)
* **Making sure each class has a clear job** (through Abstraction)

**Functions**

**A function** is a **block of code** designed to perform a **specific task**, which can be called **independently** without being tied to any object or class.

**Function** → Independent action.

**Function** = Function **outside** a class.

**Methods:**

**A method** is a **function that belongs to an object or a class** and is used to perform actions related to that specific object or class.

**Method** → Action tied to an object or class.

**Method** = Function **inside** a class.

IN JAVA :

* In **Java**, everything is organized around **classes** and **objects**.
* In Java, a function is called a **method** when inside a class. It is defined using the **method signature**, including return type, method name, and parameters.
* **Methods** in Java are functions that are **always inside a class** (or an interface).
* There are no "functions" like in C, Python, or JavaScript, where you can have a function **outside a class**.

Function Definition and Calling

A **function** is a block of code that performs a specific task. In most languages, you **define** the function and then **call** it when you want to execute it.

**Function Definition (Declaring a Function):**

* When you define a function, you tell the program **what the function does**, including its **name**, **parameters** (if any), and **return type** (if any

**Function Calling:**

* **Calling** a function is when you actually **invoke** the function to perform its task.
* This is where you provide the **actual values** (called **arguments**) to the function and get the **result**.

**What is the difference between a function in C, Java, Python, and C++?**

**C:**

* Functions in C are **standalone**, not tied to any class or object.
* No **object-oriented** features.
* Functions are declared outside any class or structure.

**Java:**

* In Java, functions are always defined as **methods** inside **classes**.
* Every method must be part of a class and is associated with an object (unless static).
* **Method signature** includes the return type, method name, and parameters.

**Python:**

* Functions in Python are also **standalone**, like C.
* Python is **dynamically typed**, meaning no need to declare the type of function parameters or return type explicitly.
* Functions are defined using the def keyword and can be called with or without objects.

**C++:**

* Functions in C++ can be **standalone** (global functions) or part of a **class** as **member functions**.
* Like C, you define functions outside classes, but you can also define member functions inside classes (like Java).

**Can a function exist outside of a class in Java?**

No, **functions in Java** cannot exist outside of a **class**. Java is a **purely object-oriented language**, and everything must be a part of a class. So, functions are **methods** in Java and are always inside classes.

**How does function overloading work in Java, C++, and Python?**

**Java and C++:**

* **Function overloading** means defining multiple functions with the **same name** but different **parameters** (number or type of arguments).

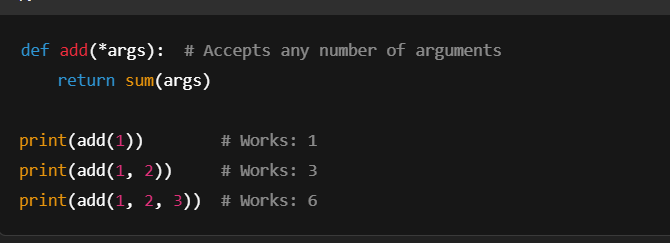
A screen shot of a computer program

AI-generated content may be incorrect.

**Python:**

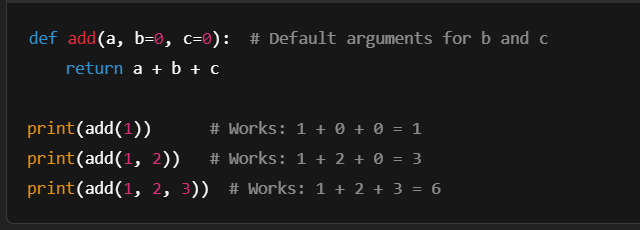
* Python does **not support function overloading** natively. However, you can mimic it by using default arguments or variable-length arguments.
* Python is a **dynamically typed** language, meaning function signatures are not defined explicitly by types. The most flexible way to handle multiple arguments is using **default parameters**, **variable-length arguments**, or conditional checks inside the function. Therefore, Python doesn’t require function overloading.

**Python provides flexibility** to mimic overloading through **default arguments** or **variable-length arguments**.



**Using Default Arguments:**

You can achieve similar behavior to overloading by assigning default values to parameters.



In this case, you can **omit** parameters, and the function will still work, just like overloading, but it’s achieved with **default values**.

Why doesn't Python support function overloading?

Python is a dynamically typed language. This means that when you call a function, Python doesn't need to know the exact types of arguments ahead of time — it infers types during runtime.

* No strict type system: Unlike Java or C++, Python doesn’t require you to declare types for function parameters, and it doesn’t care about the types when calling a function.
* Function names are unique: In Python, if you define a function with the same name more than once, the last definition of the function will overwrite the earlier ones. Python doesn't allow multiple definitions of a function with the same name, so function overloading would break that rule.

A screenshot of a computer

AI-generated content may be incorrect.

**Memory Management**

* **Java**: **Automatic Garbage Collection** (memory is freed automatically when no longer in use).
* **Python**: **Automatic Garbage Collection** (like Java, manages memory automatically).
* **C**: **Manual Memory Management** (you must allocate and free memory manually).
* **C++**: **Manual Memory Management** (similar to C, uses new/delete or malloc/free).

**. Performance**

* **Java**: Generally slower than C/C++ due to its bytecode execution in the JVM (Java Virtual Machine).
* **Python**: **Slowest** due to being an interpreted language.
* **C**: **Fastest** due to being a compiled language that directly accesses hardware resources.
* **C++**: Fast but has some overhead compared to C due to its **object-oriented features**.

**Platforms**

* **Java**: **Platform-independent** (writes once, runs anywhere — using the JVM).
* **Python**: **Cross-platform**, runs on any system with a Python interpreter installed.
* **C**: Platform-dependent, requires recompilation for different systems.
* **C++**: Similar to C, platform-dependent but can be made cross-platform with appropriate coding practices.

**. Object-Oriented Support**

* **Java**: Fully **Object-Oriented** (everything is inside a class, no procedural code outside).
* **Python**: Supports **Object-Oriented** programming but allows procedural and functional styles as well.
* **C**: No built-in support for **Object-Oriented Programming**; it’s purely **procedural**.
* **C++**: Supports both **procedural** and **Object-Oriented** programming.

A screenshot of a computer program

AI-generated content may be incorrect. **Statically Typed**

A **statically typed** language requires that the **data type** of a variable is known at **compile time** (before the program runs). This means the variable’s type is defined explicitly, and you must declare the type of data the variable will hold (e.g., int, String, float).

**Java and C++ are statically typed languages.**

**Example**:

**Java**: int age = 25; (Here, the type int is explicitly declared).

**C++**: float salary = 50000.0; (Here, float is explicitly declared).

In these languages, if you try to assign a value of the wrong type (like assigning a String to an int), the program won't compile, and you'll get an error.

**Advantages of Statically Typed Languages:**

* **Type Checking at Compile Time**: Errors related to types are caught early, making the code safer and reducing runtime errors.
* **Optimized Performance**: Since the types are known during compilation, the compiler can optimize the code more effectively, leading to better performance.

DYNAMIC TYPED LANGUAGE:

A **dynamically typed** language does **not require** you to declare the **data type** of a variable explicitly. Instead, variables can be assigned values of any type at runtime (while the program is executing). The type is inferred based on the value assigned to the variable.

**Python is a dynamically typed language.**

* **Example**:
  + age = 25 (Python automatically understands that age is an integer).
  + salary = 50000.0 (Python understands salary is a floating-point number).

In **Python**, you can reassign a variable with a value of a different type without any issues:

* age = 25 (an integer)
* age = "twenty-five" (a string)

**Advantages of Dynamically Typed Languages:**

* **Ease of Use**: You don't have to worry about explicitly specifying types, which makes the code faster to write and easier to read.
* **Flexibility**: You can easily change the type of a variable during runtime.

**Disadvantages:**

* **Potential Runtime Errors**: Since types are determined at runtime, type-related errors may not be caught until the program is running, which can make debugging harder.
* **Performance**: These languages tend to be slower than statically typed languages because the interpreter has to perform type checks at runtime.

**2.Are C++, Java, and Python fully object-oriented (OOP)?**

| **Language** | **Fully OOP?** | **Why/Why Not** |
| --- | --- | --- |
| **Java** | Almost Yes | Everything in class, except primitive types |
| **C++** | No | Allows procedural programming (no class needed) |
| **Python** | No | Supports OOP, procedural, functional styles |
|  |  |  |

Procedural = Sequential Steps + Organized with Functions

Procedural programming is a **style of programming** where the code is organized as a **sequence of instructions**, **grouped into functions**, and **executed step-by-step**.

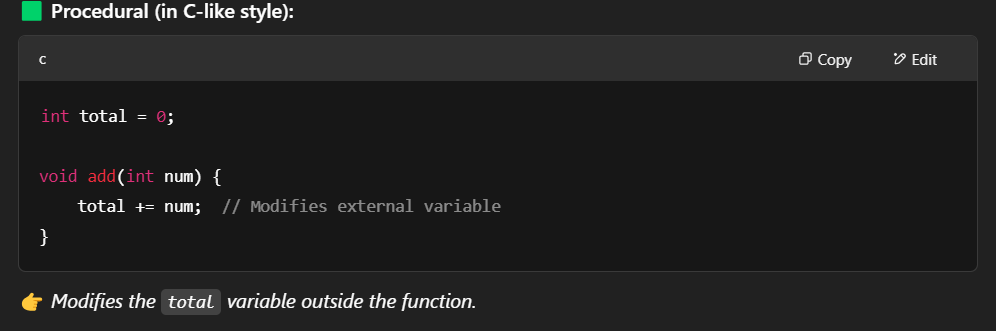
**🧑‍🍳 Procedural Approach:=>mutable**

Imagine you’re cooking a dish using a **recipe**:

1. Take ingredients
2. Chop them
3. Fry them
4. Add spices
5. Serve

Here, each step **changes the state** of the ingredients (raw → cooked). You're **mutating** the original food step by step.

🟢 *This is like Procedural Programming:*  
You write instructions (functions) that change the data (like ingredients) step by step.

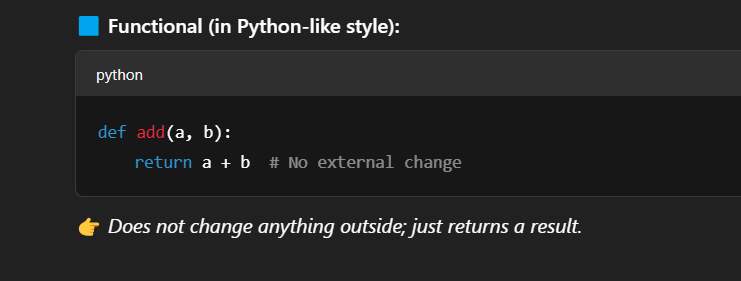


**Functional Approach:=>immutable data**

Now imagine a nutrition lab where:

* You **don’t modify** the original food.
* You use **formulas** to calculate calories, nutrients, etc., based on the **input**.
* You never change the original data — you just **create new results**.

🟢 *This is like Functional Programming:*  
You pass data into a function, and it returns a new result without changing anything. **No side effects.**



**Why This Matters in Interviews?**

* **Procedural**: Easy for small tasks, but harder to maintain in large applications because of side effects.
* **Functional**: Easier to test, debug, and scale because each function is **independent** and **pure**.

OOPS IN JAVA

SCENARIO:

**📌 Situation:**

A **teacher** asked to store data for 5 students:

* Roll Numbers
* Names
* Marks

**❌ Initial Approach (Procedural Style):**

You created **3 separate arrays**:

int[] rollNo = new int[5];

String[] names = new String[5];

int[] marks = new int[5];

But this caused problems:

Difficult to **associate** rollNo, name, and marks together.

Hard to **manage**, **sort**, or **display** a single student's full info.

No clear structure; all data is **scattered**.

Improved Approach with OOP:

You realized the better way is to **group related properties** into a single unit.

Create a Student Class:

class Student {

int rollNo;

String name;

int marks;

}

Access via Objects:

Student s1 = new Student();

s1.rollNo = 101;

s1.name = "Alice";

s1.marks = 95;

Now, one object s1 represents **all** details of a single student.

**✅ Benefits of Using Class (OOP Concept):**

| **Procedural (Old)** | **Object-Oriented (New)** |
| --- | --- |
| Separate arrays | One class groups all data (encapsulation) |
| Confusing when managing data | Easy to manage individual student data |
| No real-world structure | Mirrors real-world entities (Student object) |
| Repetitive and messy code | Clean, reusable, and modular code |

**🧠 Real-World Justification:**

A class is like a **blueprint** — once created, you can make **many students (objects)** from it.  
It helps to **encapsulate** all relevant properties into a single, manageable structure.

CLASS:

EG:1 A class is a structure that combines related data — like roll number, name, and marks of a student — into one unit. Instead of storing them in separate arrays, we define a Student class and access all the information using objects. This makes data handling more organized and realistic, just like how each student in real life has all details together."

EG:2:

A **car** has properties like **engine**, **price**, and **seats**.  
This acts like a **template** that **every car company follows** to design their own models.

Although **every car has the same type of properties**, the **values of these properties vary** from car to car:

* engine: Could be **Petrol**, **Diesel**, or **Electric**
* seats: Could be **3**, **4**, or **5**
* price: Varies based on model — ₹5 lakhs, ₹10 lakhs, ₹20 lakhs, etc.

So while the structure (template) is the same, the data inside changes — **this is exactly where OOP and classes are useful**.

Class -----------🡪 Template for car(Defines the structure: engine, seats, price)

Object ----------🡪 Real car(An instance of the class with real values)

**Why OOP Is Efficient Here:**

* Without classes: You would create **separate arrays** for engine, price, seats — which leads to confusion and scattered data.
* With OOP: **All related data is grouped** in one object, making the code **clean, organized, and scalable**.

**Key OOP Concepts in This Example:**

* **Encapsulation**: All car details (engine, seats, price) are bundled inside the Car class.
* **Object Creation**: swift and tesla are two real-world car instances with their own unique data.
* **Code Reusability**: The same class can be reused for all cars, reducing repetition.
* **Modularity**: Easy to manage and update the car data.

Class is a template of object

Object are instance of class

A **class is a concept**, while an **object is the reality** built from that concept.

A **class** is a blueprint or design of an entity. An **object** is a real-world instance created from that class. For example, *Human* is a class, but *when baby born the it becomes physical -*are objects of that class. Objects bring the class to life in memory.

Three properties of an object:

* + State-Data or attributes of an object.
  + Identity-Unique reference that identifies each object.
  + Behaiour-Actions or methods that an object can perform.

**State (Attributes/Properties)**

The **state** of an object represents the **data** or **attributes** that define the object at any given point in time. These attributes are typically represented by **variables** (fields or properties) within the class.

* **State = Data held by the object**

**Example:** For a **Car** object, the state might include attributes like:

* Engine type (e.g., petrol, diesel, electric)
* Price (e.g., 20,000 USD)
* Number of seats (e.g., 4)

These attributes hold specific values that represent the state of that car at the moment.

**Identity**

The **identity** of an object is what distinguishes it from other objects, even if they have the same state. It's a unique identifier for the object. This identity can be internal (in the form of memory location or reference) or external (like an ID or name). The identity allows us to tell different instances apart.

* **Identity = Unique reference of the object**

**Example:**

* Two **Car objects** might have the same **state** (same engine, same price, same seats), but they are **distinct objects** in memory. One car might have the identity of "Car1" and another might be "Car2" (even though they have identical attributes).

**Behavior (Methods/Functions)**

The **behavior** of an object is what it **can do** — this refers to the **methods or functions** that are defined in the class. The behavior is often influenced by the object's state. It determines the **actions** that the object can perform, which often cause changes in its state.

* **Behavior = The actions the object can perform**

**Example:** For a **Car object**, the behavior could be:

* start(): This method could change the state of the car by turning on the engine.
* stop(): This could change the state by turning off the engine.
* drive(): This might cause a change in the car’s position, speed, or fuel level.

DOT OPERATOR:

You have a Student class, and you want to access the rollno of a particular student (student1). The rollno is an **instance variable** of the Student class. Each **student object** will have its own rollno, and you access it using the **dot operator**.

The **dot (.) operator** is often referred to as the **member access operator** or **field access operator**.

USAGE:

The dot operator (.) is used to **access** the **state** (instance variables) or **behavior** (methods) of the object referenced by student1.

NEW KEYWORD:

Student student1;

This line is a **declaration**, not object creation. It declares a **reference variable** named student1 of type Student.

 **Declaration Only (No Object Created Yet)**:

* Student student1; tells the compiler:  
  “I’m going to refer to an object of type Student using the variable student1.”
* At this point, **no memory is allocated in the heap** for a Student object.
* This **reference variable (student1) is stored in stack memory**.
* But it doesn't point to any actual object yet — it's like an empty hand ready to hold a Student object.

 **Reference Variable**:

* A **reference variable** in Java holds the memory **address of an object** (not the object itself).
* Think of it as a label or pointer that “refers” to where the object lives in memory.

when a **reference variable** like student1 is declared but **not initialized**, its default value is **null**.

Student student1; // Only declared

System.out.println(student1); // Output: null

**Why?**

* In Java, all **reference variables** (i.e., variables that refer to objects) default to null if not explicitly initialized.
* This means student1 is not yet pointing to any actual Student object in the heap.

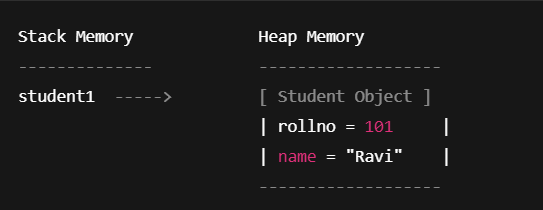
Trying to access any member (like student1.rollno) **without initializing** will cause a:

NullPointerException

Student student1 = new Student();

**What happens here:**

1. **new Student()**
   * This creates a new object of class Student.
   * Memory is **allocated at runtime** on the **heap**.
   * The new keyword returns the **reference (memory address)** of the newly created object.
2. **student1**
   * This is a **reference variable**, stored in the **stack memory**.
   * It holds the reference (or address) to the actual object in the heap.



IMPORTANT:

Student s= new Student();

All the things at left hand side works In complie time

And on right side happen in run time

**Default Values in Java (After Object Creation)**

When you create an object using new in Java, **all instance variables are automatically initialized with default values**, based on their type:

| **Type** | **Default Value** |
| --- | --- |
| int, byte, short, long | 0 |
| float | 0.0f |
| double | 0.0d |
| char | \u0000 (null character) |
| boolean | false |
| Reference types (objects, arrays, etc.) | null |

**What Is Student() in Student student1 = new Student();?**

Student() is a **constructor** — a special method in Java that is called **when an object is created** using the new keyword.

Constructor:

**Purpose of a Constructor:**

* Initializes the object at the time of creation.
* Can assign **default** or **custom values** to the object's variables.

📌 Types of Constructors in Java:

* + Default Constructor
  + Parameterized Constructor

What is a Default Constructor?

A **default constructor** is a constructor with **no parameters**. It is automatically provided by Java **only if you don’t define any constructor** in your class.

**What Happens When You Do: Student s = new Student();**

This has **two parts**:

1. **new Student()**
   * ✅ Allocates memory for the object in the **heap memory**.
   * ✅ Calls the **default constructor** to initialize the object.
   * ❗ If no constructor is defined and you're using new ClassName(), Java will insert a default one behind the scenes.
2. **Student s**
   * This declares a **reference variable** s in the **stack memory** that points to the object in the heap.

**🔹 Key Point:**

* new → Allocates memory.
* Student() → Calls the constructor to **initialize** the object.

Constructor Not Found Error

**When it happens:** You **define a parameterized constructor**, but forget to define a default constructor and then try to create an object with new ClassName() (no arguments).

class Student {

constructor Student in class Student cannot be applied to given types;

required: String

found: no arguments

Student(String name) {

// Parameterized constructor only

}

}

public class Main {

public static void main(String[] args) {

Student s = new Student(); // ❌ Error: constructor not found

}

}

**Incorrect Constructor Signature**

**When it happens:** You define the constructor with the wrong parameters or wrong access modifier.

class Student {

private Student() {

// Private constructor

}

}

Student s = new Student(); // ❌ Error: Student() has private access

The this Keyword in Java:

the this keyword is used to refer to the **current instance** of the class — i.e., the object that is being created or the object that is executing a particular method.

**What Does this Do?**

* **Refers to the Current Object:** The this keyword refers to the current object on which the method or constructor is being invoked.
* **Helps Differentiate Between Instance Variables and Parameters:** Inside a constructor or method, the this keyword is often used to differentiate between the **instance variables** of the object and the **parameters** passed to the constructor or method. This is important when the names of the parameters and instance variables are the same.

Hey when you create object the current object name is replaced the this keyword with the object name for explanation

Student vinu = new Student;

The constructor:

Student(String name,int age)

{

Vinu.name=name

Vinu.age =age

}

Student(String name,String age)

{ this.name=name

This.age=age

}

**What is a Copy Constructor?**

A **copy constructor** is a constructor that creates a new object by copying the values from another object of the same class.

|  |  |
| --- | --- |
| class Student {  String name;  int age;  // Parameterized Constructor  Student(String name, int age) {  this.name = name;  this.age = age;  }  // Copy Constructor  Student(Student other) {  this.name = other.name;  this.age = other.age;  }  } | Student s1 = new Student("Kunal", 20);  Student s2 = new Student(s1); // s2 is a copy of s1 |

**Copy Constructor in C++ vs Java**

**✅ In C++:**

* When you write:

Student s2 = s1;

* It creates a **new object** s2 with its **own memory**, and **copies values** from s1.
* This is done by the **compiler-provided copy constructor**.
* So, changing s2 won’t affect s1.

If u change s1.name then it won’t affect s2.name in c++

**In Java:**

* When you write:

Student s2 = s1;

* It **does NOT create a new object**. It just **copies the reference**.
* Both s1 and s2 point to the **same object in memory**.
* So, changes in s2 will reflect in s1.

Student s1 = new Student("Kunal", 20);

Student s2 = new Student(s1); // New object with same data, different memory

| **Feature** | **C++** | **Java** |
| --- | --- | --- |
| Copy Constructor | Provided by compiler | Not provided automatically |
| s2 = s1; | New object (copied values) | Same object (shared reference) |
| Separate memory? | Yes | No (unless explicitly copied) |

Constructor Chaining in Java Using this()

When one constructor calls another constructor in the same class using this(...).

|  |  |
| --- | --- |
| public class Student {  String name;  int age;  int marks;  // Constructor 1  Student() {  this("Default", 18, 0); // calling constru 2  System.out.println("No-arg constructor called");  } | // Constructor 2  Student(String name, int age, int marks) {  this.name = name;  this.age = age;  this.marks = marks;  System.out.println("Parameterized constructor called");  }  } |

Student s = new Student(); when u do this the default constructor called but in that this keyword call the parameterized constructor

**Is Java pass-by-value or pass-by-reference?**

➡️ **Java is always pass-by-value**, even for objects.

This means:

* When you pass a variable to a method, **Java passes a copy of the value**.
* For **primitive types**, the value itself is copied.
* For **objects**, the **reference (memory address)** is copied, **not the object itself**.

**1. Primitive types (int a = 10):**

void swap(int a, int b) {

int temp = a;

a = b;

b = temp;

}

* a and b are **copies** of the original variables.
* The original values do **not** change after calling swap().

📌 **Why?** Because primitives are **passed by value**, and that value is just the num

**. Wrapper class (Integer a = 10):**

void swap(Integer a, Integer b) {

Integer temp = a;

a = b;

b = temp;

}

* Still, a and b are **copies of the references**.
* You are just swapping the copied references — **the original variables don't change**.

**Why?**

* Java passes the **reference by value**, not the actual reference.
* Integer is **immutable**, so you cannot change its value once created.

When you declare a primitive variable as final, you **must initialize it**, and **you cannot change its value** later.

final int x = 10;

x = 20; // ❌ Error: cannot assign a value to final variable 'x'

When you declare an object reference as final, you **must initialize it**, and **you cannot point it to a different object**.  
**But**, you **can modify the object's internal state** (if the object is mutable).

final Student s = new Student("Kunal", 20);

s.name = "Rahul"; // ✅ Allowed: modifying internal state

s = new Student(); // ❌ Error: cannot assign a new object to 's'

| **Type** | **Can assign later?** | **Can modify after assign?** | **Example** |
| --- | --- | --- | --- |
| final int x | ❌ No | ❌ No | x = 10; x = 20; ❌ error |
| final Student s | ❌ No | ✅ Yes (state change) | s.name = "X"; ✅ allowed |

**Real-world Analogy:**

Think of final like **gluing a remote to one TV**:

* You **cannot switch the remote to a different TV** (can't assign a new object).
* But you **can change the channel on that TV** (modify the object's fields).

|  |  |
| --- | --- |
| import java.util.\*;  public class SwapList {  public static void main(String[] args) {  List<Integer> list = new ArrayList<>();  list.add(10); // index 0  list.add(20); // index 1  swap(list, 0, 1);  System.out.println(list); // Output: [20, 10]  }  public static void swap(List<Integer> list, int i, int j) {  Integer temp = list.get(i);  list.set(i, list.get(j));  list.set(j, temp);  }  }  **Why this works:**   * The reference to the **same ArrayList object** is passed into the method. * ArrayList is a **mutable object**, meaning its **contents** can be changed. * You're **not swapping the references**, you're **modifying the data inside the object**. | public class SwapInt {  public static void main(String[] args) {  Integer a = 10;  Integer b = 20;  swap(a, b);  System.out.println("a = " + a + ", b = " + b); // Output: a = 10, b = 20  }  public static void swap(Integer x, Integer y) {  Integer temp = x;  x = y;  y = temp;  }  }   * Here, you're swapping **copies of the references**, and **can't modify the actual Integer object** (because it's immutable). |

**☕ What is Java Garbage Collector?Java Garbage Collector (GC)** is a part of the Java Runtime Environment (JRE) that **automatically manages memory**.  
It **frees up memory** by deleting objects that are **no longer reachable** or used by the program.

**🔁 How It Works:**

1. You create objects using new.
2. Java tracks references to objects.
3. When no part of the code references an object anymore, it becomes **eligible for garbage collection**.
4. The **GC runs automatically** in the background and reclaims that memory.
5. You can *suggest* garbage collection by calling System.gc(), but **you can’t force it**.

**✅ Benefits of Java Garbage Collection:**

* 🛠 **No need to manually free memory.**
* 🚫 Avoids memory leaks and dangling pointers.
* 🔒 Improves **safety** and **security**.

public class GarbageCollectionDemo {

static class TestObject {

int value;

TestObject(int value) {

this.value = value;

} @Override

protected void finalize() throws Throwable {

/// before destroying the object in heap it print this

System.out.println("Object with value " + value + " is destroyed.");

}

}

public static void main(String[] args) {

TestObject obj;

for (int i = 1; i <= 10000000; i++) {

obj = new TestObject(i); // Previous object is eligible for GC

System.out.println(i + " objects created...");

System.gc(); // Suggest GC to run (not guaranteed)

}

}

}

}In this program we have only one reference variable obj but using for loop we create many object memory using new TestObject(i) and we assign that to one reference variable so the first created memory are eligible for garbage collection.

 finalize() is called **before** an object is garbage collected.

 System.gc() **requests** garbage collection but **does not guarantee** it will happen immediately.

 In modern Java (Java 9+), finalize() is **deprecated** and discouraged for actual resource management due to performance and reliability issues — but it's fine for demonstration.

<https://chatgpt.com/share/68139c37-25a4-8006-8778-141909aee970>

**📦 What is a Package in Java?**

In Java, a **package** is like a **folder (directory)** on your computer. It is used to **group related classes** together and keep your project organized.

Think of it like compartments in a cupboard — each compartment (package) stores different types of items (classes), so there’s no confusion.

**🧭 Why Use Packages?**

* To **avoid name conflicts**: You cannot create two classes with the **same name in the same package**, just like you can’t have two files with the same name in the same folder.
* It allow to create class of same in different package but not same package
* To **organize your code** better (like arranging files into folders).
* To provide **access control** (you can define which classes are visible to others).

package com.kunal.packages.b;

This line at the top of a Java file tells us the **exact location (path)** where the class belongs. Think of it like the **address or folder path** on your computer.

**Meaning of package com.kunal.packages.b;**

* This defines a **package hierarchy** (like nested folders):

com/

└── kunal/

└── packages/

└── b/

* The Java file where this line appears **must be saved inside** the folder b, which is inside packages, which is inside kunal, which is inside com.

**Why This Matters:**

* It helps Java **organize and locate** the class.
* When you import this class somewhere else, you use:

import com.kunal.packages.b.MyClass;

**What is static in Java?**

* **static means shared/common to all objects** of a class.
* It belongs to the **class itself**, not to any specific object.
* **Memory is allocated only once**, when the class is loaded — not every time an object is created.

**Key Points:**

* Static **variables**: Same value shared by all objects.
* Static **methods**: Can be called without creating an object.
* Static **blocks**: Run once when the class is loaded.

static long population; // Static variable shared across all instances

public Human(int age, String name, int salary, boolean married) {

this.age = age;

this.name = name;

this.salary = salary;

this.married = married;

// Correct way to access static variable

Human.population++;

// Don't use: this.population++; // This compiles, but it's bad practice

}

public void displayInfo() {

System.out.println("Name: " + name + ", Age: " + age + ", Salary: " + salary +

", Married: " + married + ", Total Population: " + Human.population);

}public static void main(String[] args) {

Human h1 = new Human(25, "Kunal", 50000, false);

Human h2 = new Human(30, "Vinu", 60000, true);

Human h3 = new Human(22, "Ravi", 45000, false);

h1.displayInfo();

h2.displayInfo();

h3.displayInfo();

}📝 Output (Sample):

Name: Kunal, Age: 25, Salary: 50000, Married: false, Total Population: 3  
Name: Vinu, Age: 30, Salary: 60000, Married: true, Total Population: 3  
Name: Ravi, Age: 22, Salary: 45000, Married: false, Total Population: 3

**🔹 Why this is not used with static in Java:**

* this refers to the **current object** — it works only with instance (non-static) variables or methods.
* But static members belong to the **class itself**, not any particular object.
* Therefore, **using this with a static variable like population would give an error or warning**, because this is not available in a static context.

Human.population += 1; // ✅ Correct

this.population += 1; // ❌ Not recommended, may show warning or confusion

 Use class name (ClassName.staticVar) to access static variables.

 Don't use this with static members — they are class-level, not tied to any one object.

🟩 Why is main method static in Java?

public static void main(String[] args)

☑️ 1. It is the entry point of any Java program.   
☑️ 2. Java needs a starting point — something to run before any object is created.   
☑️ 3. If main was not static, Java would need to create an object first — but to do that, it would need to run some code — and that code would be in main — this becomes a circular problem.   
☑️ 4. Static means it belongs to the class, not to any object — so JVM can call it directly using the class name without creating an object.

📌 Example:  
When you write:

java MyClass

JVM looks for:  
public static void main(String[] args)

And runs it like:  
MyClass.main(args); // No object created

🟨 Summary:

* main is static so it can be called without creating an object.
* main is public so JVM (which is outside the class) can access it.
* main returns void because JVM doesn't expect any return.
* String[] args allows command-line arguments.

Why can’t static methods access non-static members directly?

✳️ Because static methods belong to the class — not to any particular object.

✳️ Non-static members (variables/methods) belong to an instance (i.e., object) of the class.

➡️ So, from a static method, you don’t know which object (instance) to refer to — unless you explicitly create one.

public class Demo {

int a = 10; // non-static

static int b = 20; // static

public static void staticMethod() {

System.out.println(b); // ✅ static accessing static – allowed

// System.out.println(a); ❌ Error – cannot access non-static from static

Demo obj = new Demo(); // ✅ Create object

System.out.println(obj.a); // ✅ Now it works

}

public void nonStaticMethod() {

System.out.println(a); // ✅ non-static can access non-static

System.out.println(b); // ✅ non-static can access static

}

}

 static method → only knows about static data.

 non-static method → can access both static and non-static.//beacus it is conform that for that non static method obj created then only it can call static and non static method

 To use non-static members inside static, create an object.

In other words you cannot access non static stuff without referencing their location in a static context

Static Variable Initialization in Java

* Static variables belong to the class, not to any specific object.
* Since constructors are called only when objects are created, they cannot be used to reliably initialize static variables (as static variables should be initialized even if no objects are created).
* That's why we use a static block to initialize static variables.

🟦 Static Block:

* A static block is executed only once — when the class is loaded into memory.
* It's useful for complex static variable initialization.

public class Example {

static int count;

// Static block – runs only once when the class is loaded

static {

System.out.println("Static block executed");

count = 100;

}

// Constructor – runs every time an object is created

public Example() {

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

}

public static void main(String[] args) {

System.out.println("Main method started");

System.out.println("Initial Count: " + count);

Example obj1 = new Example();

Example obj2 = new Example();

Example obj3 = new Example();

System.out.println("Objects created.");

}

OUTPUT:

Static block executed

Main method started

Initial Count: 100

Constructor called

Constructor called

Constructor called

Objects created.

// 1. Non-static Outer, Non-static Inner  
class Outer1 {  
class Inner1 {  
void display() {  
System.out.println(" ✅ Inner1: Non-static inner of non-static outer");  
}  
}  
OUTPUT:

Outer1 outer1 = new Outer1(); // Create an object of outer class

Outer1.Inner1 inner1 = outer1.new Inner1(); // Create inner object through outer

inner1.display();

✅ This is valid. Since Inner1 is non-static, you must create it using an instance of Outer1.  
❌ Invalid example (commented out):

Outer1.Inner1 wrongInner1 = new Outer1.Inner1(); // ❌ Error!

You cannot create a non-static inner class directly — it needs an instance of the outer class.

Non-static Outer, Static Inner

// 2. Non-static Outer, Static Inner  
class Outer2 {  
static class Inner2 {  
void display() {  
System.out.println(" ✅ Inner2: Static inner of non-static outer");  
}  
}

OUTPUT:

Outer2.Inner2 inner2 = new Outer2.Inner2(); // ✅ Valid

inner2.display();

✅ This works because Inner2 is a static nested class. You can access it using the outer class name directly — no outer object needed.

// 3. Static Outer (actually still just a top-level class), Non-static Inner  
static class Outer3 {  
class Inner3 {  
void display() {  
System.out.println(" ✅ Inner3: Non-static inner of static-like outer");  
}  
}

 Outer3 is not inside any other class — so it's a top-level class.

 Java does not allow static class Outer3 {} unless it's nested inside another class.

 Inner3 is a regular non-static inner class — so it needs an Outer3 instance to be created.

As outer3 doesnot have outer class so static is not possible

So technically:

* ❌ You cannot define a static outer class.
* ✅ You can define static inner classes.
* ✅ Outer3 is not static — it's just a regular top-level class.

IN detail :

**1. Static Inner Class**

In Java, there are two types of inner classes:

* **Non-static inner class (regular inner class)**: This class requires an instance of the outer class to be instantiated.
* **Static inner class**: This class behaves like a normal class, but it is nested inside another class. The key thing about a static inner class is that it **does not require an instance of the outer class** to be instantiated.

class Outer2 {

static class Inner2 {

void display() {

System.out.println("This is the static inner class method");

}

public static void main(String[] args) {

// Creating an instance of the static inner class

Inner2 inner2 = new Inner2(); // We need to create an instance here because display() is non-static

inner2.display(); // Now we can call the method on the created object

}

}

}

}In this code:

* Outer2 is the outer class.
* Inner2 is the **static inner class** (marked with static).
* display() is a **non-static method** in Inner2.

Since Inner2 is static, you **don't need an object** of Outer2 to create an object of Inner2.

Why Do We Create Objects?

Here, display() is a **non-static** method in Inner2. So, to call it, we **must create an object** of Inner2. After that, we can call the display() method on that object.

* **Reason we need an object**: Because display() is non-static, it depends on an instance (object) of Inner2 to work. Java will not let you call display() without creating an object of Inner2.

Example 2: **Static method display()**:

class Outer2 {

static class Inner2 {

static void display() {

System.out.println(" ✅ Inner2: Static inner of non-static outer");

}

}

public static void main(String[] args) {

// Calling static method directly

Inner2.display(); // We can directly call the static method without creating an object

}

}

**Summary:**

* Static methods/variables are **class-level**, not object-level.
* **Resolved at compile time** using the **reference type**, not object type.
* No runtime polymorphism (overriding) with static methods — it's called **method hiding**.

Singleton Class:

**What Is the Singleton Pattern?**

The **Singleton pattern** ensures that a class has **only one instance** in the entire application and provides a **global access point** to it.

In short:  
➡️ **One object, reused everywhere**.

package com.kunal.singleton;

public class Singleton {

// Step 1: Create a private static instance of the class

private static Singleton instance;

// Step 2: Make the constructor private to prevent instantiation

private Singleton() {

System.out.println("Singleton instance created");

}

// Step 3: Provide a public static method to get the instance

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton(); // Create only if not already created

}

return instance;

}// **First time:** instance is null → it creates the object.

//**Next times:** It simply returns the same object — **no new object** is created.

package com.kunal.singleton;

public class Main {

public static void main(String[] args) {

Singleton obj1 = Singleton.getInstance();

Singleton obj2 = Singleton.getInstance();

System.out.println(obj1 == obj2); // true, both refer to the same object

}

}

///private constructor: This **prevents anyone** from creating a new instance with new Singleton().

// private static Singleton instance; A static variable holds the **only instance** of the class.

* It's shared among all callers.
* **🎯 Benefits of Singleton**

| **Feature** | **Benefit** |
| --- | --- |
| ✅ **Memory Efficiency** | Only one object is created, saving memory. |
| ✅ **Global Access** | Can access the same instance anywhere using Singleton.getInstance(). |
| ✅ **Controlled Instantiation** | Prevents uncontrolled creation of objects using new. |
| ✅ **Consistent State** | One object means all parts of your app use the same state. |
| ✅ **Useful in Logging, Config, DB Connection** | Commonly used for things that should only be created once. |

FOUR PILLAR OF OOPS:

* **Encapsulation**
* **Inheritance**
* **Polymorphism**
* **Abstraction**

**What is Inheritance?**

Inheritance allows a **child class** (also called subclass or derived class) to **acquire the properties and behaviors (methods)** of a **base class** (also called superclass or parent class).

| **Feature** | **Explanation** |
| --- | --- |
| Inherits Properties | Child class gets access to base class methods |
| Adds New Features | Child class can define its own additional methods |
| Can Override Methods | Child can customize base class behavior |
| Promotes Code Reuse | No need to rewrite shared logic |

**The Parent Class Initialization in the Child Class**

When you create an object of the **child class**, Java automatically takes care of **initializing the parent class** before it initializes the **child class**. The parent class's constructor is called **implicitly** before the child class's constructor is executed.

This is how Java ensures that the parent class is properly **initialized** first, and then the child class can access the parent class's properties or methods.

class Parent {

int age;

// No default constructor!

Parent(int age) {

this.age = age;

}

}class Child extends Parent {

// No call to super() here!

Child() {

// Compiler will try to insert super() — but no Parent() exists!

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

}

**What Happens?**

❗**Compilation Error**

Because Java **automatically tries to call super()** in the Child constructor — which means it looks for a **no-argument constructor** in the Parent class.

But in this case, **Parent has only a parameterized constructor**, and **no default constructor exists**.

**🧠 Why Java Fails:**

1. Java **requires** the **parent class to be fully constructed first**, even if the child class doesn’t use any parent fields directly.
2. If **no super(...) is written**, Java tries to add super() by default.
3. But if the **parent has no no-arg constructor**, the compiler gives an error:

constructor Parent in class Parent cannot be applied to given types;

required: int

found: no arguments

✅ Solution Options:

Option 1: Add a super(...) in child

class Child extends Parent {

Child() {

super(25); // pass value to parent manually

System.out.println("Child constructor");}}

Option 2: Add a no-arg constructor in parent

class Parent {

int age;

Parent() {

this.age = 0; // default value

}

Parent(int age) {

this.age = age;

}

}

**Constructor Chaining**

* When a class extends another class, **constructors are called in a chain**:
  + Parent → Child → Grandchild (if multi-level inheritance).
* This ensures that **parent class state is set up before** the child class adds anything.
* Java **always tries to call the parent’s constructor first** (either implicitly or via super(...)).

**✅ this(...) vs super(...)**

| **Keyword** | **Purpose** |
| --- | --- |
| this(...) | Calls **another constructor in the *same*** class |
| super(...) | Calls the **constructor of the *parent*** class |

**⚠️ Why Can't You Use Both Together?**

* In Java, **this(...) or super(...) must be the first statement** in any constructor.
* You **can't have two first statements** — so you can’t use **both** in the same constructor.

✅ **Correct use of super(...)**

class Parent {

Parent(int x) {

System.out.println("Parent constructor: " + x);

}

}

class Child extends Parent {

Child() {

super(10); // ✅ First line — allowed

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

}

}

✅ **Correct use of this(...)**

class A {

A() {

this(5); // ✅ Calls another constructor in same class

System.out.println("Default constructor");

}

A(int x) {

System.out.println("Parameterized constructor: " + x);

}

}

❌ **Incorrect: Using both super(...) and this(...)**

class A extends B {

A() {

this(5); // ❌ First line is this(...)

super(10); // ❌ ERROR! Can't have super after this

}

A(int x) {

super(20); // ✅ This is fine since it's the first line here

}

}

ERROR:

Constructor call must be the first statement in a constructor

ProperExample:

class Parent {

Parent(int x) {

System.out.println("👨‍👦 Parent constructor called with x = " + x);

}

}

class Child extends Parent {

// Default constructor of Child

Child() {

this(100); // ✅ Calls Child(int x) constructor

System.out.println("👶 Child default constructor");

}

// Parameterized constructor of Child

Child(int x) {

super(x); // ✅ Calls Parent constructor with x

System.out.println("👶 Child constructor with x = " + x);

}

}

**🔄 Flow of Execution:**

Child c = new Child();

Produces this output:

👨‍👦 Parent constructor called with x = 100

👶 Child constructor with x = 100

👶 Child default constructor

|  |  |
| --- | --- |
| // Parent.java  class Parent {  private int a; // ❌ Cannot access directly in Child  int b, c; // ✅ Default access (package-private)  // Default constructor  Parent() {  a = -1;  b = -1;  c = -1;  System.out.println("Parent default constructor called");  }  // Parameterized constructor  Parent(int a, int b, int c) {  this.a = a;  this.b = b;  this.c = c;  System.out.println("Parent parameterized constructor called");  }  // Getter for private variable 'a'  public int getA() {  return a;  }  }  // Child.java  class Child extends Parent {  int d; | // Default constructor  Child() {  super();  d = -1;  System.out.println("Child default constructor called");  }  // Parameterized constructor  Child(int a, int b, int c, int d) {  super(a, b, c);  this.d = d;  System.out.println("Child parameterized constructor called");  }  void display() {  // System.out.println("a = " + a); // ❌ Compile-time error: 'a' has private access  System.out.println("a = " + getA()); // ✅ Correct way to access private variable  System.out.println("b = " + b + ", c = " + c + ", d = " + d);  }  }  // Main.java  public class Main {  public static void main(String[] args) {  System.out.println("=== Using default constructor ===");  Child c1 = new Child();  c1.display();  System.out.println("\n=== Using parameterized constructor ===");  Child c2 = new Child(10, 20, 30, 40);  c2.display();  }  } |

=== Using default constructor ===

Parent default constructor called

Child default constructor called

a = -1

b = -1, c = -1, d = -1

=== Using parameterized constructor ===

Parent parameterized constructor called

Child parameterized constructor called

a = 10

b = 20, c = 30, d = 40

**Key Points about Constructor Calls in Java:**

1. **The super() call** is used to invoke the **parent class constructor**.
2. **The this() call** is used to invoke another **constructor** in the **same class** (it can be used to call a parameterized constructor from the default one, for instance).
3. **Order of calls**:
   * You **cannot call both super() and this()** in the same constructor, because **both of them must be the first statement**.
   * **You can call super() only once**, and **it must be called before this()** (or vice versa).

Example of error:

class B {

B(int x) {

System.out.println("B constructor with value " + x);

}

}

class A extends B {

A() {

this(5); // ❌ ERROR! Cannot call 'this()' before 'super()'

super(10); // ❌ ERROR! Cannot call 'super()' after 'this()'

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

}

A(int x) {

System.out.println("A constructor with value " + x);

}

}

**❌ Why this causes an error:**

* The this(5) call is meant to call the **parameterized constructor of the child class**, but it **cannot happen** before super() because the parent class must be initialized first.

Correct order:

class B {

B(int x) {

System.out.println("B constructor with value " + x);

}

}

class A extends B {

A() {

super(10); // First, call the parent constructor

this(5); // Then, call the child constructor (optional, based on child needs)

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

}

A(int x) {

System.out.println("A constructor with value " + x);

}

}

Output:

B constructor with value 10

A constructor with value 5

A constructor

Example:

class Parent {

int a, b, c;

// Parameterized constructor for Parent

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

this.a = a;

this.b = b;

this.c = c;

System.out.println("Parent constructor initialized: a = " + a + ", b = " + b + ", c = " + c);

}

}

class Child extends Parent {

int d;

// Child constructor

Child(int a, int b, int c, int d) {

super(a, b, c); // This must be the first statement

this.d = d; // Initialize child-specific variable

System.out.println("Child constructor initialized: d = " + d);

}

void display() {

System.out.println("a = " + a + ", b = " + b + ", c = " + c + ", d = " + d);

}

}

public class Main {

public static void main(String[] args) {

// Create an object of Child class

Child child = new Child(1, 2, 3, 4);

child.display();

}

}

OUTPUT:

Parent constructor initialized: a = 1, b = 2, c = 3

Child constructor initialized: d = 4

a = 1, b = 2, c = 3, d = 4

**What Happens Here:**

1. **The call to super(a, b, c)**: It initializes the parent class first (because the Parent constructor takes the parameters a, b, c). This is required before the child class performs its own initialization.
2. **The call to this.d = d**: After the parent class is initialized, the child class can initialize its own variables. In this case, this.d = d initializes the child-specific variable d.

What Happens If You Try to Change the Order?

If you try to change the order, for example, by placing this.d = d before super(a, b, c), you'll get a compilation error:

class Child extends Parent {

int d;

// Invalid constructor

Child(int a, int b, int c, int d) {

this.d = d; // This would be illegal because super() must be called first

super(a, b, c); // This would cause a compile-time error

System.out.println("Child constructor initialized: d = " + d);

}

}

**Error**: super() must be the first statement in the constructor.

can't we assign a Parent object to a Child reference like:  
Child c =new Parent() and why Java restricts access to child-specific members through a parent reference?

**Statement: Child c = new Parent();**

This is **NOT allowed** in Java. It causes a **compile-time error**:

Child c = new Parent(); // ❌ Error: incompatible types

Notes:

Normallly the left hand side define what are accessible in simple term

For example:

Parent p =new Child()

IT means all content in parent class is accessible but the newly added content in child is not accessible because

"**The reference type determines what you can access**"

So here the reference type is parent so what are the content in parent is accessible but not the content extra in child

Child c = new Parent(); // ❌ Error: incompatible types

This is not possible because whenever you create object like new Parent() it call the parent constructor only and initialized it but child construcuotr not called and it not initailzes so it is no possible

Parent p =new child();

When the object created it call the child() which child cosntrucuotr which in turn call the parent class construcutor so it is possible that parent content is initialized

|  |  |
| --- | --- |
| class Parent {  int a = 10;  void show() {  System.out.println("Parent show()");  }  }  class Child extends Parent {  int b = 20;  void display() {  System.out.println("Child display()");  }  @Override  void show() {  System.out.println("Child's overridden show()");  }  } | public class Main {  public static void main(String[] args) {  Parent p = new Child(); // ✅ Reference type: Parent, Object type: Child  p.show(); // ✅ Calls Child's show() — method overriding works at runtime (dynamic dispatch)  // p.display(); // ❌ Compile-time error: display() not in Parent  // System.out.println(p.b); // ❌ Compile-time error: 'b' not in Parent  Child c = new Child(); // ✅ Both ref and object are Child  c.show(); // ✅ Child's show()  c.display(); // ✅ Child's method  System.out.println(c.b); // ✅ Child's variable  }  } |

If a method is overridden in the child class, but you still want to call the **parent's version**, you use super.methodName().

class Parent {

void show() {

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

}

}

class Child extends Parent {

@Override

void show() {

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

super.show(); // 👉 Call the parent's version explicitly

}

public class Main {

public static void main(String[] args) {

Child c = new Child();

c.show();

}

OUTPUT:

Child show()

Parent show()

**🧠 What’s happening?**

* Child overrides show().
* But inside Child's method, you still want to run **Parent's implementation**, so you call super.show().

**Variable Hiding:**

* **Variables** cannot be overridden, but they can be **hidden**. This happens when a child class defines a variable with the same name as the parent.
* When you use super.variableName, you're accessing the **parent class’s variable** even if the child has a variable with the same name.
* The **variable resolution** happens at **compile-time**, based on the reference type.

|  |  |
| --- | --- |
| class Parent {  int num = 10;  void show() {  System.out.println("Parent show()");  }  }  class Child extends Parent {  int num = 20; // Hides the parent's variable  @Override  Voidshow(){ System.out.println("Child show()"); | super.show(); // Calls the parent’s show() method  System.out.println("Parent num using super: " + super.num); // Access parent’s variable  }  }  public class Main {  public static void main(String[] args) {  Child c = new Child();  c.show();  }  }out.println("Child show")}} |

OUTPUT:

Child show()

Parent show()

Parent num using super: 10

**What's Happening Here?**

1. **Method Resolution**:
   * c.show() calls the **overridden show() method** in the Child class. But inside this method, we use super.show() to call the parent class’s version of show().
2. **Variable Resolution**:
   * In the Child class, the num variable **hides** the num variable in the Parent class. So when you call super.num, it accesses the **parent class’s variable** (num), not the child’s.

TYPES OF INHERITANCE:

* Single Inheritance
* Multilevel Inheritance
* Hierarchical Inheritance
* Multiple Inheritance (using interfaces)
* Hybrid Inheritance

Single Inheritance

Single Inheritance is when **one child class inherits from one parent class**.

📌 It allows the child class to access the **fields and methods** of the parent class, promoting **code reuse**.

class Animal {

void eat() {

System.out.println("Animal eats");

}

}

class Dog extends Animal {

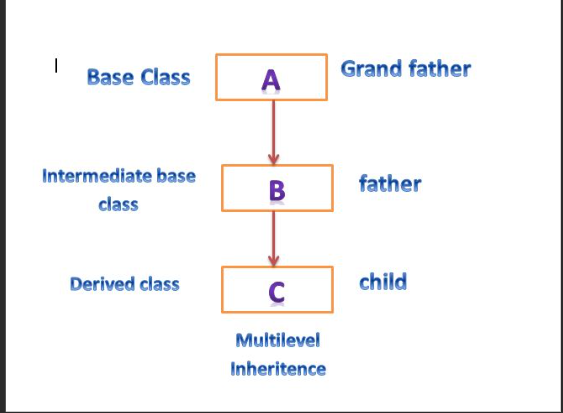
void bark() {

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

}

}// A Dog is an Animal. So Dog inherits common properties of Animal.

MULTILEVEL INHERITANCE:



Multi-level inheritance in Java is a type of inheritance where a class is derived from another class, which is also derived from another class. This forms a chain of inheritance, where each class inherits properties and methods from its parent class. For example, if class C extends class B and class B extends class A, then class C inherits from both class B and class A.

class Animal {

void breathe() {

System.out.println("Animal breathes");

}

}

class Mammal extends Animal {

void feedMilk() {

System.out.println("Mammal feeds milk");

}

}

class Human extends Mammal {

void speak() {

System.out.println("Human speaks");

}

}

▶️ Human inherits from Mammal, which inherits from Animal.  
🧠 **Human gets behaviors of both Mammal and Animal**.

Hierarchical Inheritance

**Definition**: One parent class with multiple child classes.

class Animal {

void sleep() {

System.out.println("Animal sleeps");

}

}

class Cat extends Animal {

void meow() {

System.out.println("Cat meows");

}

}

class Lion extends Animal {

void roar() {

System.out.println("Lion roars");

}

▶️ Both Cat and Lion share behavior from Animal.  
🧠 Common features like sleep() reused.

MULTIPLE INHERITANCE:

In Java, **multiple inheritance** refers to a class inheriting behaviors (methods and properties) from **more than one parent class**. However, Java does not support multiple inheritance directly through classes, due to **ambiguity** and **conflicts** that may arise when the same method is inherited from multiple classes.

**How Java Handles Multiple Inheritance:**

Java allows multiple inheritance **only through interfaces**. A class can implement multiple interfaces, thus inheriting the behaviors (abstract methods) of all those interfaces.so we know what dunctionality need we define what we want in the implementing class

**Hybrid Inheritance in Java**

**Hybrid Inheritance** is a **combination** of more than one type of inheritance. In Java, this typically means combining **multiple inheritance (via interfaces)** and

Single inheritance (or) multiple inheritance

Polymorphism:

**"Poly" means many, "morphism" means forms or ways to represent.**  
So, **polymorphism means "many ways to represent a single entity or item."**

In programming, it means one function, method, or object can take **many forms** depending on the context.

Types of polymorphisms:

Compile time polymorphisms / static polymorphisms -methond overloading

Runtime polymorphisms / run time polymorphisms -method overriding

Method overloading:

Same name but types, arguments,return types ,ordering cam be different:

Multiple constructor:

A a =new A();

A a1= new A(1,2);

The java determine which constructor to be called is determined

class Number {

// 1. Two int parameters

int sum(int a, int b) {

return a + b;

}

// 2. Three int parameters

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

return a + b + c; // <-- Fix: You missed +c

}

// 3. String and int

int sum(String a, int b) {

return Integer.parseInt(a) + b; // Parses string to int and adds

}

//4. Double sum(double a,double b)

{

Return a+b

}

}

public class Main {

public static void main(String[] args) {

Number obj = new Number();

System.out.println(obj.sum(2, 4)); // Calls 2-int version

System.out.println(obj.sum(2, 3, 4)); // Calls 3-int version

System.out.println(obj.sum("2", 3)); // Calls String-int version

// This will give \*\*error\*\*, because no method matches 5 int arguments:

// System.out.println(obj.sum(2, 3, 4, 5, 5));

}

}

**❓ If I call sum(2, 2.0) — what happens?**

Let’s break this down:

* 2 is an int
* 2.0 is a double

Now, your defined methods are:

* sum(int, int)
* sum(int, int, int)
* sum(String, int)

There is **no method** that accepts (int, double), so **Java will throw a compile-time error**, saying:

*"method sum in class Number cannot be applied to given types..."*

int sum(int a, int b) // Method 1

double sum(double a, double b) // Method 2

**Method Selection Rules (Overloading Resolution)**

When you call a method, Java uses the following rules **in order** to pick the correct one:

Rule 1: **Exact match first**

Java looks for a method where the parameter types **exactly match** the arguments passed:

sum(2, 3); // picks sum(int, int)

**Rule 2: Widening primitive conversion**

If no exact match is found, Java will **widen** a smaller data type to a larger one.

| **From** | **To** |
| --- | --- |
| Byte-> | short, int, long, float, double |
| Short-> | int, long, float, double |
| Int-> | long, float, double |
| Float-> | double |

sum(2, 3); // if no int version, will use sum(double, double)

**Rule 3: Autoboxing**

If no widening works, Java will try **autoboxing** — automatically converting a **primitive to its wrapper** class.

| **Primitive** | **Wrapper** |
| --- | --- |
| int | Integer |
| double | Double |

sum(Integer a, Integer b); // can be matched with sum(2, 3)

But this is **slower than widening**, so Java uses widening first if both are available.

**Rule 4: Varargs (Variable arguments)**

If no match works, Java tries varargs, e.g.:

int sum(int... numbers) {

// Called when no exact or widened match is found

}

sum(2, 3, 4, 5); // if no method with 4 arguments, varargs is used

**Rule 5: Compile-time Error**

If **none** of the above match, Java will throw a **compile-time error**.

sum(2, 3, 4, 5, 6); // No match found → Error!

**4. Why method overloading?**

* **Code clarity**: Same name for logically related operations
* **Flexibility**: Accept different types and counts of inputs
* **Avoid method duplication** with different names

**. Static Methods Can Be Overloaded** ✅

class Test {

static void display(int a) {

System.out.println("Static int: " + a);

}

static void display(String s) {

System.out.println("Static String: " + s);

}

public static void main(String[] args) {

display(10); // Calls int version

display("hello"); // Calls String version

}

}

Static method which is overloaded have separate memory:

| **Concept** | **Static Method** |
| --- | --- |
| Stored in | **Method Area** of JVM |
| Tied to | **Class**, not instances |
| Overloading? | Yes — **Separate memory for each overloaded method** |
| Memory Sharing? | No — Overloaded static methods have separate memory |
| Accessed by | **Class name** (or object, but it's still class-based) |

 St**atic methods** are **separate methods** in memory (even if overloaded) because they have distinct **method signatures**.

 They are stored in the **method area** of JVM memory and are **shared across all instances** of the class.

 Even though they can be called on an object, they are always resolved **at the class level**, not at the instance level.

Runtime polymorphishm:

When the child has same method as parent classs no different in argument return type but only change in method definition then it is called overriding.

class Shape {

void area() {

System.out.println("Area in Shape");

}

}

class Circle extends Shape {

void area() {

System.out.println("Area of Circle = π \* r \* r");

}

}

class Square extends Shape {

void area() {

System.out.println("Area of Square = side \* side");

}

}

public class Main {

public static void main(String[] args) {

Shape s = new Shape(); // Object of parent class

Circle c = new Circle(); // Object of child class (Circle)

Square sq = new Square(); // Object of child class (Square)

Shape square = new Square(); // Polymorphism: parent reference, child object

// Method calls

s.area(); // Output: Area in Shape

c.area(); // Output: Area of Circle = π \* r \* r

sq.area(); // Output: Area of Square = side \* side

square.area(); // Output: Area of Square = side \* side

}

}

Types of the reference variable is parent class but the object is of subclass in this occur method overriding:

**Upcasting:**

Upcasting happens when you assign an object of a **subclass** to a reference variable of its **parent class** type.

**Dynamic Method Dispatch** (also known as **Runtime Polymorphism**) is a core concept in **object-oriented programming** in Java. It refers to the **runtime decision-making process** where the method that gets invoked is determined by the **actual object type** (not the reference type) at runtime.

**✅ What is Dynamic Method Dispatch?**

Dynamic method dispatch is the mechanism by which Java determines which **overridden method** to call at **runtime**.

In Java, the **method call** is resolved at **runtime**, which means the **most specific version of the method** is called based on the **actual object** being referred to by the reference variable, even if the reference is of a **parent class** type.

**How Does Dynamic Method Dispatch Work?**

* In **dynamic method dispatch**, the method to be called is decided based on the **actual object type** that a reference variable points to, not the **type of the reference variable**.
* It allows a **child class** method to override a method in the **parent class** and ensures that **polymorphism** works by calling the overridden method of the actual object at runtime.

See in my example shapes has area methos if u define Shapes s= new Circle() means we know that what all things available in shape only can access but what method to run is deterimend by dynamic method dispatch.

Example:

Class shape{

}

Class Circle extends Shape{

Void area()

{  
System.out.println(“Area of circle”);

}

Shape s =new Circle();

It show error when you access s.area() as shape class doesnot have method area

As the method what we can access is determined based on reference variables but area is not in shape reference so it show error,

Class Shape{

Void area()

{

System.out.println(“shape”);

}

Class Circle{

Void area()

{

System.out.println(“Circle”)

}

Shape s= new Circle()

method to be called is decided based on the **actual object type** that a reference variable points to, not the **type of the reference variable**.

So child class object method area is called and the output is circle

A method declared with the final keyword **cannot be overridden** by subclasses. This is used to **prevent changing the implementation** of a method in any derived class.

class Parent {

final void show() {

System.out.println("Final method in Parent");

}

}

class Child extends Parent {

// ❌ This will cause a compile-time error

// void show() {

// System.out.println("Trying to override");

// }

}

**✅ 1. Early Binding (Static Binding)**

**Definition**:  
Early binding means that the **method call is resolved at compile-time**. The compiler knows exactly which method to call.

**✅ Characteristics:**

* Happens during **compile time**.
* Applies to:
  + **Static methods**
  + **Final methods**
  + **Private methods**
  + **Method overloading**
* **Faster**, because there’s no decision to make at runtime.
* Also called **compile-time binding** or **static dispatch**.

**2. Late Binding (Dynamic Binding)**

* **Definition**:  
  Late binding means that the **method call is resolved at runtime**, based on the actual object.

**✅ Characteristics:**

* Happens during **runtime**.
* Applies to:
  + **Overridden methods**
  + **Instance methods** (non-static, non-final, non-private)
* Also called **runtime binding** or **dynamic dispatch**.
* Enables **polymorphism**.

////important:

class Animal {

static void sound() {

System.out.println("Animal static sound");

}

}

class Dog extends Animal {

static void sound() { // This is NOT an override; it's method hiding

System.out.println("Dog static sound");

}

}

public class Main {

public static void main(String[] args) {

Animal a = new Dog(); // Upcasting

a.sound(); // Calls Animal.sound(), not Dog.sound()

}

}**❌ Static methods are not overridden**

* When you declare static methods in both parent and child classes **with the same signature**, it is **not overriding**.
* It is called **method hiding**.

**📌 Method resolution is based on reference type, not object**

* In the line Animal a = new Dog();, the **reference type is Animal**.
* So, a.sound() calls Animal.sound(), even though the actual object is Dog.
* This is **early binding**, not dynamic (late) binding.

Static is belonged to class and overrideing is possible in inheritance so for overriding we need object but static donot need object so method overriding is not possible

3.Encapsulation:solving implementation level issue(internal)

Wrapping up implementation of data members and method in class.

So we cann protected it from outside world

**🔹 Real-World Example:**

**ATM Machine**

* You insert your card and enter your PIN.
* Internally, the ATM **validates your PIN**, **checks your balance**, and **updates your account**.
* You are **not allowed to directly access the bank database**.
* The **data (account info)** is protected and can be accessed **only through methods**.

4.Abstraction:solving design level issue(external)

U give this method I donot want to know how u define the those method

Hiding unnecessary detail we have car we put key and car start but u donot need to know how engine run ,pertorl pump so we no need to know this we only need key

Easy example is system.out.println we only want this ,not how this is defined inside so this this called abstract

Abstraction means **hiding the internal details** and only showing the **essential features** to the user.

**Real-World Example:**

**Car Steering Wheel**

* When you drive a car, you **turn the steering wheel** to change direction.
* You don’t need to know **how the steering mechanism works inside** (gears, hydraulics, etc.).
* Only the **essential behavior (turn left/right)** is exposed to you.

| **Feature** | **Abstraction** | **Encapsulation** |
| --- | --- | --- |
| **What it hides** | Hides **implementation details** | Hides **data and internal state** |
| **Focus on** | **What** the object does | **How** the object secures its data |
| **Access** | Uses abstract classes / interfaces | Uses private, public, setters/getters |
| **Goal** | To reduce complexity | To protect data from unauthorized access |
| **Real-world** | Car dashboard, mobile app UI | ATM PIN system, medical records system |
| **Feature** | Encapsulation | Data Hiding |
| **Definition** | Wrapping data + methods into one unit (class) | Hiding internal data using access modifiers |
| **Goal** | To reduce complexity and improve modularity | To provide data security and protect integrity |
| **Achieved By** | Creating classes, grouping variables and methods | Using private variables + public getters/setters |
| **Focus** | On structure and design of the code | On restricting access to sensitive info |
| **Real-World Example** | A medical capsule that wraps many drugs | The recipe inside a locked vault — only approved people can access |
| **Relation** | Encapsulation includes data hiding | Data hiding is a part of encapsulation |

**🔒 Data Hiding = Security**

You restrict **direct access** to variables (like password) by marking them private. Users can’t access password directly — only via methods.

**🎯 Simple Analogy:**

* **Encapsulation** is like making a machine with **buttons**.
* **Data Hiding** is locking the internal parts so no one can open or mess with them.

**✅ Conclusion:**

* **Encapsulation** = Good design → makes code modular, manageable.
* **Data Hiding** = Good security → prevents misuse, protects data.