🔥 Constructor in Java

Definition

A **constructor** is a special block of code in a class that **runs automatically** when an object of the class is created.

```
class Student {
    String name;
    int age;

    Student() { // constructor
        name = "Unknown";
        age = 0;
    }
}
```

Purpose

- Initialize **object variables (fields)**
- Put the object in a **valid state**
- Optional: Execute **any setup logic** when object is created

---

🔥 Key Properties of Constructors

1.Constructor Name = Class Name

- **Rule:** The constructor **must have the same name as the class**.
- This is how Java identifies it as a constructor, not a normal method.

Example:

```
class Student {
    Student() { // same name as class
        System.out.println("Constructor called..!!");
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student(); // calls constructor automatically
    }
}
```

✅ Output:

```
Constructor called..!!
```

If you write a **different name**, it becomes a method, not a constructor.

2.No Return Type

- **Constructors cannot have return type**, not even `void`.
- If you add a return type → it becomes a **regular method**.

Example:

```
class Student {
    void Student() { // NOT a constructor, it's a method
        System.out.println("This is a method");
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student(); // default constructor used
    }
}
```

✅ Output:
(nothing prints because `Student()` method is NOT called automatically)

Key: **Constructors execute automatically**, methods don't.

3.Called Automatically

- The constructor is called **automatically** when an object is created using `new`.

Example:

```
class Student {
    Student() {
        System.out.println("Constructor executed");
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student(); // constructor called automatically
    }
}
```

✅ Output:

```
Constructor executed
```

- You **don't need to call it explicitly**.

4.Used for Initialization

- Constructors are mainly used to **initialize instance variables**.
- Can assign **default values** or **custom values** passed through parameters.

Example:

```
class Student {
    String name;
    int age;
    // Constructor with parameter
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student("Aniket", 29);
        System.out.println(s.name + " " + s.age); // Aniket 29
    }
}
```

- You can also have **default constructor:**

```
Student() {
    name = "Unknown";
    age = 0;
}
```

✅ **Benefit:** Ensures objects are **initialized properly** at creation.

Summary Table

Point	Detail	Example
Name = class	Constructor name must match class	Student()
No return type	Not even void	void Student() → method
Called automatically	Executes when object created	new Student()
Used for initialization	Assigns default/custom values	this.name = name

5.Constructor Overloading

Concept:

- **Overloading** means having **multiple constructors** in the same class **with different parameters.**
- Allows creating objects in **different ways** depending on available data.

Rules:

- Must have **same class name.**
- Must have **different parameter lists** (number or type).
- Return type is **not allowed** (constructor never has return type).

Example:

```
class Student {
    String name;
    int age;

    // Constructor 1
    Student() {
        name = "Unknown";
        age = 0;
    }

    // Constructor 2
    Student(String name) {
        this.name = name;
        age = 18;
    }

    // Constructor 3
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student("Aniket");
        Student s3 = new Student("Malika", 25);

        System.out.println(s1.name + " " + s1.age); // Unknown 0
        System.out.println(s2.name + " " + s2.age); // Aniket 18
        System.out.println(s3.name + " " + s3.age); // Malika 25
    }
}
```

✓ **Benefit:** Flexible object creation without writing separate initialization methods.

6.Constructors Cannot Be Inherited

Concept:

- A constructor **belongs only to its own class**.
- **Child class does NOT inherit** the parent class constructor.
- But a **child class can call parent constructor** using super().

Example:

```
class Animal {
    Animal(String type) {
        System.out.println("Animal type: " + type);
    }
}

class Dog extends Animal {
    Dog() {
        super("Mammal"); // calling parent constructor
        System.out.println("Dog created");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
    }
}
```

Output:

```
Animal type: Mammal
Dog created
```

✓ Key point:

- Child **cannot inherit parent constructor** directly.
- super() allows initializing parent part of object.

7. Constructor Can Be Private (Singleton Pattern)

Concept:

- Making a constructor **private** prevents other classes from creating objects.
- Usually used in **Singleton Pattern** → only **one instance** of a class exists.

Example (Singleton):

```
class Database {
    private static Database instance;

    private Database() { // private constructor
        System.out.println("Database created");
    }

    public static Database getInstance() {
        if (instance == null) {
            instance = new Database(); // only place to create object
        }
        return instance;
    }
}

public class Main {
    public static void main(String[] args) {
        Database db1 = Database.getInstance();
        Database db2 = Database.getInstance();

        System.out.println(db1 == db2); // true, same object
    }
}
```

✅ Benefit:

- Prevents **multiple object creation**.
- Ensures **single, global instance**.

Summary Table

Feature	Detail	Example/Use
Constructor Overloading	Multiple constructors with different parameters	Student() / Student(String) / Student(String,int)
Cannot be inherited	Child class cannot automatically use parent constructor	Use super() in child constructor
Can be private	Restrict object creation	Singleton pattern (Database.getInstance())

8. Default Constructor

Concept:

- If you **don't write any constructor** in your class, Java **automatically provides a default constructor**.
- It has:

    - **No arguments**
    - **Empty body**
- Purpose: To allow object creation even if you didn't write a constructor.

Example 1: No constructor written

```
class Student {
    String name;
    int age;
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student(); // default constructor called
        System.out.println(s.name); // null
        System.out.println(s.age);  // 0
    }
}
```

✓ Output:

```
null
0
```

Java automatically adds:

```
Student() { } // default constructor
```

Example 2: If you write any constructor

```
class Student {
    String name;

    Student(String name) {
        this.name = name;
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student(); // ERROR → no default constructor
    }
}
```

✗ Output: Compilation Error

Because **once you create a constructor**, Java **does NOT provide the default one**.

Key Points to Remember:

1. **Automatic only if you write no constructor.**
2. **Default constructor has no code**, so instance variables get **default values**:
    - int → 0
    - double → 0.0
    - boolean → false
    - Object → null
3. If you need **custom initialization**, you must write your **own constructor**.

⚙ Types of Constructors in Java

Java constructors are mainly classified into **3 types**:

1. **Default Constructor (Compiler-created)**
2. **No-Argument Constructor (User-created)**
3. **Parameterized Constructor**

---

1 Default Constructor (Compiler-created)

Concept

- If **no constructor** is written in a class, Java **automatically provides a default constructor**.
- It has:
  - **No arguments**
  - **Empty body**
- Purpose: Allows object creation even if the programmer didn't write a constructor.

Example

```
class A {
    int x;
    // No constructor written → Java adds default constructor automatically
}

public class Main {
    public static void main(String[] args) {
        A obj = new A(); // calls default constructor
        System.out.println(obj.x); // 0 → default value for int
    }
}
```

✓ Key Points:

- Instance variables get **default values** (int → 0, String → null, boolean → false)
- If you write **any constructor**, the default constructor is **not created** by Java.

---

2 No-Argument Constructor (User-created)

Concept

- A constructor **written by the programmer** with **no parameters**.
- Purpose: Initialize objects with **default values** or execute setup code.

Example

```
class Car {
    String brand;
    int price;
    Car() { // No-arg constructor
        brand = "Honda";
        price = 10000;
        System.out.println("Car object created");
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car(); // calls user-created no-arg constructor
        System.out.println(c.brand + " - " + c.price);
    }
}
```

✓ Output:

```
Car object created
Honda - 10000
```

Use Case:

- When you want to **initialize objects with default/custom values** at creation.

---

3 Parameterized Constructor

Concept

- Constructor with **parameters** to **pass values at the time of object creation**.
- Allows objects to be initialized with **custom values**.

Example

```
class Employee {
    String name;
    int age;

    Employee(String name, int age) { // parameterized constructor
        this.name = name;
        this.age = age;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e1 = new Employee("Aniket", 25);
        Employee e2 = new Employee("Malika", 22);

        e1.display(); // Name: Aniket, Age: 25
        e2.display(); // Name: Malika, Age: 22
    }
}
```

✓ Key Points:

- Initializes each object with **specific values**.
- Useful in **real-world projects** like Employee records, Bank accounts, Book objects, etc.

Comparison Table

Type	Parameters	Provided by	Purpose	Example
Default Constructor	None	Compiler	Allow object creation	class A { }
No-Argument Constructor	None	User	Initialize default values, run setup code	Car() { brand="Honda"; }
Parameterized Constructor	Yes	User	Initialize object with custom values	Employee(String name,int age)

💡 Tips:

- If you need **flexibility**, use **constructor overloading**: have **both no-arg and parameterized constructors**.
- Always use `this` **keyword** to distinguish between **instance variables and parameters**.

🔥 Constructor Overloading in Java

Concept

- **Overloading** = Multiple constructors **in the same class** with **different parameter lists**.
- Purpose: Allow objects to be created in **different ways**, depending on available information.
- **Same constructor name** (class name), **different parameters** (number or type).

Rules of Constructor Overloading

1. **Same class name** → all constructors must match class name.
2. **Different parameter lists** → either in **number of parameters** or **type/order of parameters**.
3. **Return type is not allowed** → constructors don't have return type.
4. **Can be called automatically** when object is created.

Example: Box Class

```
class Box {
    int w, h;

    // No-argument constructor
    Box() {
        w = 1;
        h = 1;
        System.out.println("No-arg constructor called");
    }

    // One-argument constructor
    Box(int size) {
        w = h = size;
        System.out.println("One-arg constructor called");
    }

    // Two-argument constructor
    Box(int w, int h) {
        this.w = w;
        this.h = h;
        System.out.println("Two-arg constructor called");
    }

    void display() {
        System.out.println("Width: " + w + ", Height: " + h);
    }
}

public class Main {
    public static void main(String[] args) {
        Box b1 = new Box();       // No-arg
        Box b2 = new Box(5);      // One-arg
        Box b3 = new Box(3, 7);   // Two-arg

        b1.display(); // Width:1, Height:1
        b2.display(); // Width:5, Height:5
        b3.display(); // Width:3, Height:7
    }
}
```

Output:

```
No-arg constructor called
One-arg constructor called
Two-arg constructor called
Width: 1, Height: 1
Width: 5, Height: 5
Width: 3, Height: 7
```

✅ Key Points

1. **Flexibility in Object Creation**
   - Users can create objects **without parameters**, **with one parameter**, or **with multiple parameters**.
2. **Avoids multiple initialization methods**
   - Instead of writing separate methods like `init1()`, `init2()`, use **constructor overloading**.
3. `this` **keyword**
   - Helps distinguish **instance variables** from **parameters**.
   - Can also be used for **constructor chaining**: calling one constructor from another using `this()`.

Example of Constructor Chaining

```
class Box {
    int w, h;

    Box() {
        this(1, 1); // calls two-arg constructor
        System.out.println("No-arg constructor called");
    }

    Box(int size) {
        this(size, size); // calls two-arg constructor
        System.out.println("One-arg constructor called");
    }

    Box(int w, int h) {
        this.w = w;
        this.h = h;
        System.out.println("Two-arg constructor called");
    }
}
```

Benefits:

- Reduces **duplicate code**
- Cleaner initialization

📌 Summary Table

Constructor	Parameters	Purpose
No-arg	None	Default object values
One-arg	int size	Square box with same width & height
Two-arg	int w, int h	Custom width & height

Constructor Overloading = Flexibility + Clean Code + Multiple Ways to Initialize Objects

☸ How Constructor Calls Work in Java (Inheritance)

Rule 1: Parent constructor is always called first

- When a **child object** is created, Java automatically calls the **parent class constructor before executing the child's constructor**.
- This ensures that **parent part of the object is initialized** before the child part.

Rule 2: Use of `super()`

- `super()` is used to explicitly call the **parent constructor**.
- If you don't write `super()`, Java **automatically inserts it** in the child constructor (only if parent has no-arg constructor).
- `super()` must be the **first statement** in the child constructor.

Example 1: Simple Parent-Child Constructor Call

```
class Parent {
    Parent() {
        System.out.println("Parent Constructor");
    }
}

class Child extends Parent {
    Child() {
        System.out.println("Child Constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
    }
}
```

Output:

```
Parent Constructor
Child Constructor
```

Explanation:

1. `new Child()` is called
2. Java automatically calls **Parent()** first
3. Then executes **Child()**

Example 2: Using `super()` with Parameterized Constructor

```
class Parent {
    Parent(String message) {
        System.out.println("Parent says: " + message);
    }
}

class Child extends Parent {
    Child() {
        super("Hello from Parent"); // explicitly call parent constructor
        System.out.println("Child Constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
    }
}
```

Output:

```
Parent says: Hello from Parent
Child Constructor
```

✓ Key Points:

- `super()` can pass **parameters** to parent constructor.
- Must be the **first line** in child constructor.

Rule 3: Multiple Levels of Inheritance

```
class GrandParent {
    GrandParent() { System.out.println("GrandParent"); }
}

class Parent extends GrandParent {
    Parent() { System.out.println("Parent"); }
}

class Child extends Parent {
    Child() { System.out.println("Child"); }
}

public class Main {
    public static void main(String[] args) {
        new Child();
    }
}
```

Output:

```
GrandParent
Parent
Child
```

Explanation:

- Constructors are called **top-down** from **grandparent** → **parent** → **child**.

Rule 4: Constructor Chaining in Inheritance

- Child constructor can **call different parent constructor** using `super(parameters)`.
- Helps in **custom initialization** in multi-level inheritance.

Summary Table: Constructor Call in Inheritance

Rule	Detail
Parent first	Parent constructor is always called before child
Automatic call	`super()` is automatically added if parent has no-arg constructor
Explicit call	Use `super(params)` to call a specific parent constructor
Multi-level	Constructors called **top-down** from highest parent

---

🎯 Why Do We Need Constructors?

Constructors are not just syntax—they **serve important purposes** in object-oriented programming.

---

1 To Initialize Object Data

- When an object is created, **instance variables need values**.
- Constructors let you **initialize variables automatically** at creation.

Example

```
class Student {
    String name;
    int age;

    Student(String name, int age) { // parameterized constructor
        this.name = name;
        this.age = age;
    }
}

Student s = new Student("Aniket", 25);
System.out.println(s.name + ", " + s.age); // Aniket, 25
```

✅ Without constructor, variables could remain uninitialized or need extra setter methods.

---

2 To Avoid Uninitialized Variables

- If variables are not initialized, they get **default values** (0, null, false), which may not be meaningful.
- Constructors ensure **meaningful default or custom values**.

```
class Car {
    String brand;
    int price;

    Car() { // no-arg constructor
        brand = "Unknown";
        price = 1000;
    }
}

Car c = new Car();
System.out.println(c.brand + ", " + c.price); // Unknown, 1000
```

3 To Force Object into a Valid State

- Constructors **ensure object cannot exist in invalid state**.
- Example: Prevent negative age, empty name, etc.

```
class Employee {
    String name;
    int age;

    Employee(String name, int age) {
        if(age < 0) throw new IllegalArgumentException("Age cannot be negative");
        this.name = name;
        this.age = age;
    }
}

Employee e = new Employee("Malika", 22); // valid
// Employee e2 = new Employee("John", -5); // ERROR
```

✓ Prevents invalid data at object creation.

4 To Perform Setup Tasks

- Sometimes creating an object requires **setup actions** beyond just storing data.
- Example: database connection, opening a file, initializing network resources.

```
class Database {
    Database() {
        System.out.println("Connecting to database...");
        // code to connect DB
    }
}

Database db = new Database(); // DB setup happens automatically
```

✓ Summary Table

Purpose	Example
Initialize object data	Student s = new Student("Aniket", 25)
Avoid uninitialized variables	Car() { brand="Unknown"; price=1000; }
Force valid state	Employee(String name,int age) validates age >= 0
Perform setup tasks	Database() { connect to DB }

💡 **Key Idea:**
Constructors **ensure every object is ready to use immediately** after creation, without requiring extra method calls.

🔥 Special Points About Constructors

Rule 1: Constructor cannot be `static`

Reason:

- Static members belong to the **class**, not the object.
- Constructor is used to **initialize an object**, so it **must be called on object creation**.
- If it were static, it would belong to the class, and JVM would **not know which object to initialize**.

Example (Illegal):

```
class Student {
    static Student() {  // ❌ Error
        System.out.println("Constructor cannot be static");
    }
}
```

Rule 2: Constructor cannot be `final`

Reason:

- `final` means it **cannot be overridden**.
- JVM internally **calls constructors during object creation**, even in inheritance.
- If constructor were final, **child class object creation could fail** because JVM cannot call parent constructor internally.

Example (Illegal):

```
class Student {
    final Student() {  // ❌ Error
        System.out.println("Constructor cannot be final");
    }
}
```

Rule 3: Constructor cannot be `abstract`

Reason:

- `abstract` means **no body** and must be implemented in a subclass.
- Constructor **must have a body** to initialize object fields.
- JVM needs to **run the constructor code** during object creation — abstract constructor would prevent that.

Example (Illegal):

```
abstract class Student {
    abstract Student();  // ❌ Error
}
```

✅ Summary Table

Restriction	Reason
static	Belongs to class, cannot initialize object
final	Cannot be overridden, JVM needs to call it during object creation
abstract	Must have body to initialize object

Key Takeaway for Interviews

- Always remember:

“Constructor is always instance-level and executable, so it cannot be static, final, or abstract.”

✅ Access Modifiers for Constructors in Java

Constructors can have **all four access modifiers**:

1. **public**
2. **protected**
3. **default** (package-private) → no keyword
4. **private**

Each modifier controls **where objects of the class can be created**.

1 Public Constructor

Rules

- Object can be created **from anywhere** (any class, any package).
- Most common type of constructor.

Example

```
public class Car {
    public Car() {
        System.out.println("Car created");
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car(); // accessible anywhere
    }
}
```

Use Case

- Your class should be **freely usable and instantiable everywhere**.
- Example: `String`, `ArrayList`, etc.

---

2 Protected Constructor

Rules

- Object can be created:
  - Inside the **same package**
  - In **subclasses** (even in different packages)
- Useful in **inheritance-based designs**.

Example

```
public class Vehicle {
    protected Vehicle() {
        System.out.println("Vehicle created");
    }
}

class Car extends Vehicle {
    Car() {
        super(); // can call protected constructor
        System.out.println("Car created");
    }
}
```

Use Case

- Restrict object creation to **subclasses or same package**.
- Example: Frameworks where only child classes should instantiate parent class.

---

3 Default (Package-Private) Constructor

Rules

- Object can be created **only inside the same package**.
- No access modifier is written.

Example

```
class Employee {
    Employee() { // default constructor
        System.out.println("Employee created");
    }
}
```

Use Case

- Restrict object creation **within the package**.

- Useful for **internal utilities, package-level frameworks**.

4 Private Constructor

Rules

- Object creation **only allowed inside the same class**.
- No one else can create objects outside.

Example: Singleton Pattern

```
class Singleton {
    private static Singleton instance;

    private Singleton() { // private constructor
        System.out.println("Singleton created");
    }

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Example: Utility Class

```
class Utils {
    private Utils() { } // prevent object creation
    public static int add(int a, int b) { return a+b; }
}
```

Use Case

- Prevent external object creation
- Common in: **Singleton, Factory, Utility classes**

✓ Quick Access Modifier Summary Table

Modifier	Object Creation	Use Case
public	Anywhere	Freely usable classes
protected	Same package + subclasses	Inheritance-based design
default	Same package only	Package-level restriction
private	Only within class	Singleton, Factory, Utility classes

💡 Key Tip:

- Use **private constructor + static method** for Singleton or utility classes.
- Use **protected/default** to **control visibility within package/subclasses**.
- Use **public** for general-purpose classes.

⚠ Important Rules About Constructors in Java

---

1 Constructors follow the same access rules as methods

- **Rule:** A constructor's access modifier (public, protected, default, private) works **exactly like methods**.
- There are **no special rules** for constructor access, except the context of object creation.

Example

```
class Demo {
   public Demo()
{
   System.out.println("Public constructor");
}
   protected Demo(int x)
{
  System.out.println("Protected constructor");
 }
   Demo(String s)
{
   System.out.println("Default constructor");
 }
   private Demo(double d)
{
   System.out.println("Private constructor");
}
}
```

- Objects can be created **only where allowed by the access modifier**.

---

2 All Constructors Private → No external instantiation

- If a class **only has private constructors, objects cannot be created from outside the class**.
- Commonly used in **Singletons, Factory classes, or utility classes**.

Example

```
class Singleton {
   private static Singleton instance;
   private Singleton() { System.out.println("Singleton created"); }

   public static Singleton getInstance() {
      if (instance == null) instance = new Singleton();
      return instance;
   }
}

public class Main {
   public static void main(String[] args) {
      // Singleton s = new Singleton(); // ✕ Error
      Singleton s = Singleton.getInstance(); // ✓ Allowed
   }
}
```

---

3 No Constructor Written → Java Creates Default Constructor

- If **no constructor is provided**, Java generates a **default constructor** automatically.
- **Access level:**
  - If class is **public**, default constructor is **public**
  - If class has **no modifier**, default constructor is **package-private**

Example

```
class Employee {
    // No constructor written
}

public class Main {
    public static void main(String[] args) {
        Employee e = new Employee(); // ✅ Default constructor called
    }
}
```

- **If you write any constructor**, default constructor is **not created**.

---

4 Protected Constructors are common in abstract classes

- Abstract classes **cannot be instantiated directly**.
- But **child classes** may need to call **parent constructor**.
- Using `protected` allows:
  - **Subclasses** (even in different packages) to call the constructor
  - Prevents **external object creation**

Example

```
abstract class Vehicle {
    protected Vehicle()
{
 System.out.println("Vehicle created");
}
}

class Car extends Vehicle {
    Car() {
     Super();
     System.out.println("Car created");
}
}

public class Main {
    public static void main(String[] args) {
        // Vehicle v = new Vehicle(); // ❌ Cannot instantiate abstract class
        Car c = new Car(); // ✅ Allowed
    }
}
```

✅ Summary Table

Rule	Explanation	Example / Use Case
1	Same access rules as methods	public, protected, default, private
2	All private → cannot instantiate externally	Singleton, Utility classes
3	No constructor → Java creates default	Default constructor (access depends on class type)

Rule	Explanation	Example / Use Case
4	Protected in abstract class	Allows subclasses to call constructor

---

💡 Interview Tip:

- Always mention:

“Constructor access works like methods, but private constructors are used to **control object creation**, and protected constructors are useful in **abstract class inheritance**.”

💡 Singleton Pattern — Using Private Constructor

Concept

- **Singleton** ensures a class has **only one object** throughout the application.
- Achieved by:
    1. Making the **constructor private** → prevents external object creation.
    2. Providing a **static method** to return the **single instance**.

Example

```
class AppConfig {
   // 1 Single instance of AppConfig
   private static AppConfig instance = new AppConfig();

   // 2 Private constructor → cannot create object from outside
   private AppConfig() {
      System.out.println("AppConfig created");
   }

   // 3 Public static method to provide the single instance
   public static AppConfig getInstance() {
      return instance;
   }
}

public class Main {
   public static void main(String[] args) {
      AppConfig a1 = AppConfig.getInstance(); // ✅ returns instance
      AppConfig a2 = AppConfig.getInstance(); // ✅ same instance

      System.out.println(a1 == a2); // true → only one object exists
   }
}
```

Output:

```
AppConfig created
true
```

Key Points

1. **Private Constructor**
    - Prevents external object creation.
    - Guarantees only **one instance** exists.

2. **Static Instance**
    - Stored inside class, accessible globally.
3. **Public Static Method**
    - Provides controlled access to the **single instance**.
4. **Use Cases**
    - Configuration settings (AppConfig, DatabaseConfig)
    - Logging classes
    - Thread pools

Why Constructor Matters Here

- Without **private constructor**, someone could do:

```
AppConfig obj = new AppConfig(); // ❌ breaks singleton
```

- Using private constructor enforces **controlled instantiation**.

✅ Difference Between Method and Constructor (Point-wise)

◆ 1. Purpose: Constructor vs Method

1 Constructor

Purpose:

- A constructor's main job is to **initialize an object** when it is created.
- It sets the **initial state** of the object by assigning values to instance variables or performing **setup tasks**.
- It **does not perform business logic** or actions beyond initialization.

Key Points:

- Automatically called **when an object is created**.
- Name is **same as the class**.
- No return type (not even void).

Example:

```
class Employee {
    String name;
    int age;

    // Constructor initializes object state
    Employee(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("Employee object created");
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e = new Employee("Aniket", 25);
        // Constructor has initialized name and age
    }
}
```

✅ Here, the constructor ensures that every `Employee` object has **name and age initialized** when created.

---

2 Method

Purpose:

- A method is used to **perform actions, business logic, or behavior** of an object.
- It **does not automatically run**; it must be called explicitly.
- Methods can **return values** and can have any name.

Key Points:

- Used to **define object behavior**.
- Can manipulate object data, perform calculations, or interact with other objects.
- Can be **called multiple times** on the same object.

Example:

```
class Employee {
    String name;
    int age;

    Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method performs an action
    void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e = new Employee("Aniket", 25);
        e.displayInfo(); // Method called explicitly to perform action
    }
}
```

✅ Here, `displayInfo()` **performs a task** (prints details) but does **not initialize the object**.

Summary Table: Constructor vs Method

Feature	Constructor	Method
Purpose	Initialize object/state	Perform actions, behavior, logic
Name	Same as class	Any valid name
Return type	None	Can have any return type
Called	Automatically during object creation	Explicitly by object
Frequency	Once per object creation	Can be called multiple times

---

Key Idea:

Constructor = "Sets up the object"
Method = "Makes the object do something"

◆ 2. Name: Constructor vs Method

---

1 Constructor Name

Rule:

- A constructor **must have the same name as the class**.
- This is how Java **identifies it as a constructor** and not a regular method.

Key Points:

- Name is **case-sensitive**, must match **exactly with the class name**.
- If the name does not match the class, Java treats it as a **normal method**.

Example:

```
class Employee {
    String name;

    // Correct constructor → name same as class
    Employee(String name) {
        this.name = name;
        System.out.println("Constructor called");
    }

    // Incorrect → becomes a method, not a constructor
    void Employee(String name) {
        System.out.println("This is a method, not a constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e = new Employee("Aniket"); // calls constructor
        e.Employee("Malika"); // calls method
    }
}
```

Output:

```
Constructor called
This is a method, not a constructor
```

✅ Notice how **the same name but with a return type** becomes a **method**, not a constructor.

---

2 Method Name

Rule:

- A method can have **any valid identifier** as its name.
- Naming is flexible and should follow **Java naming conventions**.
- Methods are identified by their **name + parameters** (signature).

Example:

```
class Employee {
    void displayInfo() {
        System.out.println("Displaying employee info");
    }

    int calculateBonus(int salary) {
        return salary / 10;
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e = new Employee();
        e.displayInfo();          // method call
        System.out.println(e.calculateBonus(50000)); // method call
    }
}
```

✅ Summary Table: Name

Feature	Constructor	Method
Name rule	Must match class name	Any valid identifier
Return type	None (not even void)	Required if method returns value
Identification	JVM recognizes by name + class	Compiler recognizes by name + parameters
Example	Employee(String name)	displayInfo(), calculateBonus(int salary)

Key Idea:

Constructor = "named exactly like class to initialize it"
Method = "flexibly named to perform actions"

◆ 3. Return Type: Constructor vs Method

1 Constructor Return Type

Rule:

- **Constructors cannot have a return type**, not even void.
- If you write a return type, Java treats it as a **normal method**, not a constructor.

Reason:

- A constructor's job is to **initialize the object**, not return a value.
- The object reference is **automatically returned** when you create it using new.

Example:

```
class Employee {
    String name;

    // Constructor → no return type
    Employee(String name) {
        this.name = name;
        System.out.println("Constructor called");
    }

    // Incorrect → has return type, becomes a method
    void Employee(String name) {
        System.out.println("This is a method, not constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e1 = new Employee("Aniket"); // calls constructor
        e1.Employee("Malika");                // calls method
    }
}
```

Output:

```
Constructor called
This is a method, not constructor
```

✅ JVM distinguishes constructor **by name + no return type**.

---

2 Method Return Type

Rule:

- A method **must have a return type**.
- Can be void (no return value) or any data type (int, String, etc.).

Example:

```
class Employee {
    // void method → no return
    void displayInfo() {
        System.out.println("Displaying employee info");
    }

    // non-void method → returns a value
    int calculateBonus(int salary) {
        return salary / 10;
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e = new Employee();
        e.displayInfo();                  // void method
        int bonus = e.calculateBonus(50000); // non-void method
        System.out.println(bonus);
    }
}
```

✅ Summary Table: Return Type

Feature	Constructor	Method
Return type	None (not even void)	Must have a return type (void/non-void)
Purpose	Initialize object	Perform actions/behavior
Example	Employee(String name)	void displayInfo(), int calculateBonus(int salary)

Key Idea:

Constructor = "initializes object, no return needed"
Method = "performs action, must return or not"

◆ 4. When It Is Called: Constructor vs Method

---

1 Constructor Call

Rule:

- A **constructor is called automatically** when an object is created using the `new` keyword.
- You **do not call it manually**.

Reason:

- Its purpose is to **initialize the object immediately** upon creation.

Example:

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

public class Main {
    public static void main(String[] args) {
        Car c = new Car();  // constructor called automatically
    }
}
```

Output:

```
Car object created
```

✅ Notice: The constructor **runs automatically** as soon as `new Car()` executes.

---

2 Method Call

Rule:

- A method must be **called explicitly** using the object reference (or class name if static).
- Methods **do not run automatically** when the object is created.

Example:

```
class Car {
    void start() {  // method
        System.out.println("Car started");
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car(); // object created, constructor runs
        c.start();         // method called manually
    }
}
```

Output:

```
Car object created
Car started
```

✅ Notice: Constructor ran automatically, but the `start()` method **only ran when called manually**.

✅ Summary Table: Call Time

Feature	Constructor	Method
Call type	Automatically during object creation	Explicitly by object (or class if static)
Frequency	Runs **once per object creation**	Can be called **multiple times**
Example	`Car c = new Car();`	`c.start();`

Key Idea:

Constructor = "runs automatically to set up the object"
Method = "runs manually to perform actions whenever needed"

◆ 5. Number of Times Called: Constructor vs Method

---

1 Constructor

Rule:

- A **constructor is called exactly once** for **each object** at the time of creation.
- It **cannot be called again** explicitly for the same object.

Example:

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

public class Main {
    public static void main(String[] args) {
        Car c1 = new Car(); // constructor called once
        Car c2 = new Car(); // constructor called again for new object
    }
}
```

Output:

```
Car object created
Car object created
```

✅ Each object creation triggers **its own constructor call**.

---

2 Method

Rule:

- A method can be **called any number of times** on the same object.
- Methods are **behavior/action oriented**, so you can run them whenever needed.

Example:

```
class Car {
    void start() {  // method
        System.out.println("Car started");
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car(); // constructor runs once
        c.start();         // method call 1
        c.start();         // method call 2
        c.start();         // method call 3
    }
}
```

Output:

```
Car started
Car started
Car started
```

✅ Methods can be executed **multiple times on the same object**.

✅ Summary Table: Number of Times Called

Feature	**Constructor**	**Method**
Called per object	Once	Any number of times
Example	`Car c = new Car();`	`c.start();` multiple times
Reason	Initializes object state once	Performs actions/behavior whenever needed

---

Key Idea:

Constructor = "called once per object to initialize it"
Method = "can be called repeatedly to perform actions"

◆ 6. Inheritance: Constructor vs Method

---

1 Constructor

Rule:

- **Constructors are not inherited** by child classes.
- Each class must **define its own constructor**.
- However, a **child class can call a parent constructor** using `super()`.

Reason:

- Constructor's main job is to **initialize the object of its own class**, so inheriting it doesn't make sense.

Example:

```
class Parent {
    Parent() {
        System.out.println("Parent constructor called");
    }
}

class Child extends Parent {
    Child() {
        super(); // calling parent constructor
        System.out.println("Child constructor called");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
    }
}
```

Output:

```
Parent constructor called
Child constructor called
```

✅ Notice: Parent constructor is **called explicitly using `super()`**, but **Child has its own constructor**.

---

2 Method

Rule:

- Methods **are inherited** by child classes (except **private methods**).
- Child can:
    - Use the method as-is
    - Override it with its own implementation

Example:

```
class Parent {
    void greet() {
        System.out.println("Hello from Parent");
    }
}

class Child extends Parent {
    void greet() {
        System.out.println("Hello from Child");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.greet();       // Calls overridden method in Child
        ((Parent)c).greet(); // Calls Parent version if needed via super in Child
    }
}
```

✅ Methods are **inherited and can be overridden**, unlike constructors.

✅ Summary Table: Inheritance

Feature	Constructor	Method
Inherited	✕ No	✓ Yes (except private)
Can call parent	Yes, using super()	Directly inherited or overridden
Example	Child() { super(); }	Child inherits greet()

Key Idea:

Constructor = "initializes its own class, not inherited"
Method = "defines behavior, inherited by child class"

◆ 7. Overloading: Constructor vs Method

1 Constructor Overloading

Rule:

- **Constructors can be overloaded** in the same class.
- Overloading means **multiple constructors with the same name (class name) but different parameter lists**.
- **Constructor cannot be overridden** because they are **not inherited**.

Example:

```
class Box {
    int w, h;

    // No-arg constructor
    Box() {
        w = h = 1;
        System.out.println("No-arg constructor called");
    }

    // One-arg constructor
    Box(int size) {
        w = h = size;
        System.out.println("One-arg constructor called");
    }

    // Two-arg constructor
    Box(int w, int h) {
        this.w = w;
        this.h = h;
        System.out.println("Two-arg constructor called");
    }
}

public class Main {
    public static void main(String[] args) {
        Box b1 = new Box();
        Box b2 = new Box(5);
        Box b3 = new Box(3, 7);
    }
}
```

Output:

```
No-arg constructor called
One-arg constructor called
Two-arg constructor called
```

✅ Overloading allows **flexible object creation** with different initial states.

---

2 Method Overloading and Overriding

Method Overloading:

- Same method name, **different parameters** (number, type, or order).
- Happens **within the same class**.

Method Overriding:

- Child class **redefines a method** from parent class with the **same name and parameters**.
- Happens **across inheritance**.

Example:

```
class Parent {
    void greet() {
        System.out.println("Hello from Parent");
    }

    void greet(String name) { // Overloaded
        System.out.println("Hello " + name);
    }
}

class Child extends Parent {
    @Override
    void greet() { // Overridden
        System.out.println("Hello from Child");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.greet();       // Overridden method → Hello from Child
        c.greet("Aniket"); // Overloaded method → Hello Aniket
    }
}
```

Output:

```
Hello from Child
Hello Aniket
```

✅ Key differences:

- **Constructors**: Can be **overloaded**, **cannot be overridden**
- **Methods**: Can be **overloaded** and **overridden**

✅ Summary Table: Overloading & Overriding

Feature	Constructor	Method
Overloading	✅ Yes	✅ Yes
Overriding	❌ No	✅ Yes (via inheritance)

Feature	Constructor	Method
Example	Box() / Box(int) / Box(int,int)	greet() / greet(String) / overridden in child

Key Idea:

Constructor overloading = different ways to create objects
Method overloading = same behavior, different inputs; overriding = change behavior in subclass

◆ 8. When Does Compiler Create One: Constructor vs Method

1 Constructor

Rule:

- If you **do not write any constructor** in your class, the **Java compiler automatically creates a default constructor**.
- This default constructor:
  - Has **no arguments**
  - Has an **empty body**
  - Sets up the object so it can be created without errors

Example:

```
class Employee {
    // No constructor written
}

public class Main {
    public static void main(String[] args) {
        Employee e = new Employee(); // default constructor created by compiler
        System.out.println("Object created successfully");
    }
}
```

Output:

```
Object created successfully
```

✅ Notice: The compiler silently provides a constructor so `new Employee()` works.

Important:

- If you write **any constructor** (even parameterized), **compiler does not create the default constructor**.

```
class Employee {
    Employee(String name) { // Parameterized constructor
        System.out.println("Employee created: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        // Employee e = new Employee(); // ❌ Error! No default constructor
        Employee e2 = new Employee("Aniket"); // ✅ Works
    }
}
```

2 Method

Rule:

- The compiler **never creates a method** automatically.
- **All methods must be explicitly written** by the programmer.
- JVM only provides a **default constructor**, not default methods.

Example:

```
class Employee {
    // No methods written
}

public class Main {
    public static void main(String[] args) {
        Employee e = new Employee();
        // e.displayInfo(); // ✗ Error! Method does not exist
    }
}
```

✓ Summary Table: Compiler Creation

Feature	Constructor	Method
Created automatically by compiler	✓ If no constructor is written	✗ Never
Type	Default (no-arg)	N/A
Example	`Employee e = new Employee();`	Must define explicitly

Key Idea:

Constructor = compiler provides a default if none exists
Method = programmer must define it explicitly

◆ 9. Object Creation: Constructor vs Method

1 Constructor

Rule:

- A constructor is **automatically executed when an object is created** using `new`.
- In fact, **constructors are an integral part of the object creation process**.
- They **initialize the object's state** and make it ready for use.

Example:

```
class Car {
    String model;
    Car(String model) { // constructor
        this.model = model;
        System.out.println(model + " Car object created");
    }
}

public class Main {
    public static void main(String[] args) {
        Car c1 = new Car("BMW"); // constructor called during object creation
        Car c2 = new Car("Audi"); // another object created
    }
}
```

Output:

```
BMW Car object created
Audi Car object created
```

✅ Notice: Each `new Car()` automatically calls the constructor to create and initialize the object.

---

2 Method

Rule:

- Methods **cannot create objects by themselves**.
- They **operate on existing objects** and perform actions or return values.
- Object creation must always use `new` (and call constructor), not a method.

Example:

```
class Car {
    String model;

    Car(String model) { // constructor
        this.model = model;
    }

    void display() { // method
        System.out.println("Car model: " + model);
    }
}

public class Main {
    public static void main(String[] args) {
        // Car c = display(); // ❌ Cannot create object via method
        Car c = new Car("BMW"); // ✅ Object created using constructor
        c.display();            // Method called on existing object
    }
}
```

✅ Methods **work on objects**, but cannot **instantiate objects themselves**.

✅ Summary Table: Object Creation

Feature	Constructor	Method
Object creation	✅ Part of creation process	❌ Cannot create object
Automatic call	✅ Yes, during `new`	❌ No, must be called explicitly
Example	`Car c = new Car("BMW");`	`c.display();`

Key Idea:

Constructor = "creates and initializes object"
Method = "performs actions using an existing object"

◆ 10. Abstract, Final, Static — Constructor vs Method

▨ Constructor

✕ Constructors cannot be:

- **abstract**
- **final**
- **static**

Let's understand **WHY**:

---

1 Why constructor cannot be abstract?

Abstract means:

- No body
- Must be overridden in subclass

But a constructor MUST:

- Have a body (to initialize object)
- Is never inherited, so overriding is impossible

☞ So abstract constructor makes no sense.

📌 Simple logic:

A constructor's job is to initialize the object → So it must have a body → So it cannot be abstract.

---

2 Why constructor cannot be final?

Final means:

- Cannot be overridden.

But:

- Constructors are **never overridden** in the first place.

- They are not inherited into child classes.

☞ So making it final is meaningless.

📌 **Logic:**

Something that cannot be overridden should not be marked final.

---

3 Why constructor cannot be static?

Static means:

- Belongs to class, not object
- Called without an object

But:

- Constructor’s job is to **create the object**
- Without object creation, calling constructor is pointless

☞ So static constructor would create a logical conflict.

📌 **Logic:**

Constructor creates the object → Static belongs to class → Class-level constructor is impossible.

▨ Methods

Methods **CAN** be:

- **abstract**
- **final**
- **static**

✔ Abstract Method

Used in abstract classes to force child class implementation.

✔ Final Method

Prevents method overriding.

✔ Static Method

Belongs to class, can be called without object.

▨ Summary Table

Feature	Constructor	Method
abstract	✕ Not allowed	✓ Allowed
final	✕ Not allowed	✓ Allowed
static	✕ Not allowed	✓ Allowed

◆ 11. Calling Another Constructor / Method

▨ Constructor Behavior

✓ A constructor *can* call another constructor

There are **two ways** a constructor can call another constructor:

1 Using `this()` → calls constructor of SAME class

```
class Car {
    Car() {
        this(100);   // calling parameterized constructor
        System.out.println("Default Constructor");
    }

    Car(int speed) {
        System.out.println("Speed: " + speed);
    }
}
```

Output:

```
Speed: 100
Default Constructor
```

Rule:

`this()` must always be the **first line** in the constructor.

2 Using `super()` → calls constructor of PARENT class

```
class Vehicle {
    Vehicle() {
        System.out.println("Vehicle Constructor");
    }
}

class Car extends Vehicle {
    Car() {
        super();   // calls parent constructor
        System.out.println("Car Constructor");
    }
}
```

Output:

```
Vehicle Constructor
Car Constructor
```

Rule:

`super()` must also be the **first line** in the constructor.

! Important Constraints

✓ You can use either `this()` OR `super()`, not both.

Because both must be FIRST statement.

✓ If you don’t write `super()`, Java inserts it automatically.

▨ Method Behavior

✕ A method CANNOT call a constructor directly.

You cannot do:

```
start();  // OK
Car();    // ✕ ERROR (cannot call constructor like method)
```

✓ Only one way to run a constructor inside a method:

Use the **new keyword** to create an object.

```
void start() {
   Car c = new Car();  // constructor runs here
}
```

This is the **only** valid way.

▨ Summary Table

Feature	Constructor	Method
Call another constructor	✓ Using `this()` or `super()`	✕ Not allowed
Call constructor directly	✓ Internally using this()/super()	✕ Cannot call constructor
Run constructor via object	✕ No need	✓ Using `new` keyword

◆ 12. Memory Allocation

▨ Constructor

✓ Constructor is tied to object creation + memory allocation

When you write:

```
Car c = new Car();
```

Steps happen in this order:

1️⃣ **Memory allocated in Heap** for object
2️⃣ **Instance variables get default values** (0, null, false)
3️⃣ **Constructor runs** to initialize custom values
4️⃣ Reference `c` is assigned to object

So:

✓ Memory allocation → constructor call → object becomes ready.

Example

```
class Car {
    int speed;

    Car() {
        speed = 50;
        System.out.println("Car Created");
    }
}

new Car();
```

During execution:

- Heap memory created
- `speed = 0` (default)
- Constructor runs → `speed = 50`

▨ Method

✓ Method does NOT allocate memory.

A method only **operates on an existing object**.

```
Car c = new Car();
c.start();  // start() runs only after object exists
```

Steps:

1. Object memory already created in heap

2. Constructor already executed
3. Now method is free to run ANY number of times

Key Difference

Operation	Constructor	Method
Triggered during object creation	✓ Yes	✕ No
Allocates heap memory	✓ Yes (part of new)	✕ Never
Runs after object exists	✕ No	✓ Yes
Can be called many times	✕ Once per object	✓ Any number

Simple Example

```
Car c = new Car();  // constructor runs → memory allocates
c.start();          // method runs → no memory allocation
c.start();          // method again → no memory allocation
```

◆ 13. Object Initialization

▨ Constructor

✓ Constructor is used for initial (first-time) object setup

When an object is created, its instance variables need values.
Constructor ensures the object starts in a **valid state**.

Example:

```
class User {
    String name;
    int age;

    User(String name, int age) {   // constructor
        this.name = name;          // initialization
        this.age = age;
    }
}
```

When you write:

```
User u = new User("Aniket", 25);
```

What happens?

- Object memory allocated
- Default values assigned (name=null, age=0)
- Constructor replaces them with **actual values**
- Object becomes ready to use

✓ Constructor = First-time initialization
✓ Guarantees object isn’t empty / invalid

▨ Method

✓ Methods are used to modify or update values AFTER the object is created.

Example:

```
u.setAge(30);   // method changes value later
u.updateName("Aniket Varpe");
```

Methods allow **runtime changes**.

⚡ Simple Example to Understand the Difference

```
class BankAccount {
    String owner;
    int balance;

    BankAccount(String owner) {     // constructor
        this.owner = owner;
        this.balance = 0;           // initial value
    }

    void deposit(int amount) {      // method
        balance += amount;          // updates later
    }
}
```

◆ When object is created:

```
BankAccount acc = new BankAccount("Aniket");
```

Constructor runs:

- owner = "Aniket"
- balance = 0

◆ Later:

```
acc.deposit(500);
acc.deposit(300);
```

Methods run:

- balance updated to 800

✓ Final Summary

Feature	Constructor	Method
Purpose	Initialize object state	Modify/change state later
When used?	On object creation	Anytime after creation
Runs automatically?	✓ Yes	✕ No
How many times?	Once per object	Many times

◆ 14. Usage in Design Patterns

Constructors and methods are used **very differently** in real-world Java design patterns.

Let's break it down clearly 👇

▨ Constructor — Special Uses in Patterns

Constructors are used **carefully** in patterns where object creation must be controlled.

1 Singleton Pattern

Goal: **Only one object** of a class should exist.

How constructor is used:

- Make constructor **private**
- Prevents creating new objects with `new`
- Force access through `getInstance()`

Example:

```
class AppConfig {
    private static AppConfig instance = new AppConfig();

    private AppConfig() { }  // private constructor

    public static AppConfig getInstance() {
        return instance;
    }
}
```

✓ Constructor controls the number of objects
✓ Prevents misuse
✓ Ensures only one instance

---

2 Factory Pattern

Goal: **Encapsulate object creation logic**.

Here:

- Constructor is **hidden** (private or default)
- Object creation is handled by a method (factory)

Example:

```
class Car {
    private Car() { }  // hide constructor

    public static Car createSportsCar() {
        Car c = new Car();
        // set properties
        return c;
    }
}
```

✓ Constructor hidden
✓ Factory method decides which object to create

3️⃣ Immutable Classes

Goal: Object cannot be changed once created.
(Like `String` class in Java)

How constructor is used:

- Constructor sets **all values**
- No setter methods
- Fields marked `private final`

Example:

```
final class User {
    private final String name;
    private final int age;

    User(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

✓ Constructor gives initial state
✓ Methods cannot modify state

▨ Methods — Used Everywhere in Behavior Patterns

Methods are used for:

- **Business logic**
- **Behavior**
- **Processing**
- **Handling events**
- **Executing algorithms**

Common patterns using **methods**:

1️⃣ Strategy Pattern

Different behaviors implemented using methods.

```
interface PaymentStrategy {
    void pay(int amount);   // method, not constructor
}
```

✓ Methods decide behavior
✓ Constructor only builds object

2️⃣ Observer Pattern

Methods notify listeners:

```
interface Observer {
    void update();
}
```

✓ Constructor not important
✓ Methods provide functionality

3️⃣ Template Method Pattern

Defines algorithm steps.

```
abstract class Meal {
    final void prepareMeal() {  // method
        cook();
        serve();
    }

    abstract void cook();
    abstract void serve();
}
```

✓ Heavy focus on methods
✓ Constructor rarely used

🔥 Final Summary

Feature	Constructor Usage	Method Usage
Purpose in patterns	Object creation control	Behavior, logic, actions
Common patterns	Singleton, Factory, Immutable	Strategy, Observer, Template
When used	During object creation	Throughout object lifecycle
Why important	Restricts or manages object creation	Implements application logic

🧩 Short Summary Table

Feature	Constructor	Method
Purpose	Initialize object	Perform action/logic
Name	Same as class	Any name
Return type	None	void / any type
Called when	Object created	When invoked
Auto-call	Yes	No
Overloading	Yes	Yes
Overriding	No	Yes
Inherited	No	Yes
Compiler creation	Yes (default)	No
Special keywords	this(), super()	this, super

Constructors with Inheritance (Java) —

all rules + examples + diagrams + interview points.

✅ 1. Basic Rule:

When a child object is created → Parent constructor runs first**

```
class Parent {
    Parent() {
        System.out.println("Parent constructor");
    }
}

class Child extends Parent {
    Child() {
        System.out.println("Child constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        new Child();
    }
}
```

Output

```
Parent constructor
Child constructor
```

✓ Parent constructor is always called **before child**.
✓ This happens automatically.

✅ 2. Why Parent Constructor Runs First?

Because the child object contains **parent part + child part**. So Java must **initialize the parent part first**.

✅ 3. How does Java call Parent constructor? → Using super()

Every child constructor has an **implicit super()** at the first line.

```
Child() {
    super();  // inserted by compiler
    System.out.println("Child");
}
```

Even if you don't write it → compiler adds it.

✅ 4. If Parent has ONLY parameterized constructor → Child MUST call it

❌ This will cause error:

```
class Parent {
    Parent(int x) { }
}

class Child extends Parent {
    Child() { }   // ERROR – compiler inserts super(), but no Parent()
}
```

✔ Correct way:

```
class Parent {
    Parent(int x) { }
}

class Child extends Parent {
    Child() {
        super(10);   // must call parent constructor manually
    }
}
```

✅ 5. You can call a specific parent constructor using super(arguments)

```
class Parent {
    Parent() { System.out.println("Parent default"); }
    Parent(int x) { System.out.println("Parent with value: " + x); }
}

class Child extends Parent {
    Child() {
        super(100);    // choose parent constructor
        System.out.println("Child constructor");
    }
}
```

✅ 6. super() must be FIRST line in constructor

```
Child() {
    System.out.println("Hello"); // ❌ error
    super();  // must be first
}
```

You cannot place anything before super().

✅ 7. this() and super() cannot be used together in the same constructor

Because both must be on the **first line**.

```
Child() {
    this(10);   // OK
    // super(); // ❌ Not allowed
}
```

✅ 8. Constructor Chaining

Parent → Child → GrandChild

```
class A {
    A() { System.out.println("A"); }
}

class B extends A {
    B() {
        super();     // calls A
        System.out.println("B");
    }
}

class C extends B {
    C() {
        super();     // calls B
        System.out.println("C");
    }
}

public class Main {
    public static void main(String[] args) {
        new C();
    }
}
```

Output

```
A
B
C
```

!9. Private Parent Constructor → Child cannot extend

```
class Parent {
    private Parent() { }
}
```

```
class Child extends Parent { }  // ❌ error
```

Because child cannot call super() (it's private).

🎯 10. Abstract Class and Constructors

Abstract class **can have constructors**.

Child must call them through super().

```
abstract class Animal {
    Animal() {
        System.out.println("Animal created");
    }
}

class Dog extends Animal {
    Dog() {
        System.out.println("Dog created");
    }
}
```

📌 Complete Real-Life Example:

```
class Vehicle {
    String type;

    Vehicle(String t) {
        this.type = t;
        System.out.println("Vehicle: " + type);
    }
}

class Car extends Vehicle {
    String model;

    Car(String t, String m) {
        super(t);        // calling parent constructor
        this.model = m;
        System.out.println("Car Model: " + model);
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car("4-Wheeler", "BMW");
    }
}
```

Output

```
Vehicle: 4-Wheeler
Car Model: BMW
```

🏁 Summary (Quick Memory Points)

- Child constructor always calls parent constructor first.
- super() is used for calling parent constructor.
- If parent has parameterized constructor → child must call it.
- super() is always the **first line**.
- this() and super() cannot be in the same constructor.
- Constructor chaining happens parent → child → grandchild.
- Abstract classes also have constructors.

✅ 1. Constructor Overloading (Allowed)

Constructor **overloading** means:

A class having multiple constructors with different parameter lists.

✓ Rules for Overloading

- Same class
- Same constructor name (class name)
- Different:
  - number of parameters
  - type of parameters
  - order of parameters

🔥 Example: Constructor Overloading

```
class Student {

    String name;
    int age;

    // Constructor 1 – No arguments
    Student() {
        System.out.println("Default Constructor");
    }

    // Constructor 2 – One argument
    Student(String name) {
        this.name = name;
        System.out.println("Name Constructor");
    }

    // Constructor 3 – Two arguments
    Student(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("Name + Age Constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        new Student();
        new Student("Aniket");
        new Student("Aniket", 25);
    }
}
```

Output

```
Default Constructor
Name Constructor
Name + Age Constructor
```

✓ Constructor Overloading **is allowed**.

✕ 2. Constructor Overriding (NOT Allowed)

Constructors **cannot be overridden** in Java.

Why constructor overriding is impossible?

Because:

1. **Constructors are not inherited**
   → Only inherited methods can be overridden.
2. **Constructor name = class name**
   → A child class has a different name, so it cannot define a constructor with the same name as parent's class.
3. **Constructors belong to class, not to object**
   → They initialize that class's objects only.

So overriding cannot happen.

✕ Example Showing Constructor Cannot Be Overridden

```
class Parent {
    Parent() {
        System.out.println("Parent Constructor");
    }
}

class Child extends Parent {

    // ✕ This is NOT overriding
    Child() {
        System.out.println("Child Constructor");
    }
}
```

Output when object is created:

```
new Child();
Parent Constructor
Child Constructor
```

✓ Child constructor **does NOT override** parent constructor.
✓ Child simply has **its own constructor**.
✓ Parent constructor still runs first through super().

🔥 Why this is NOT overriding?

Because overriding means:

Child class redefining **same method name, same signature, same return type**, inherited from parent. Constructors **do not meet any of these rules**. So they **cannot be overridden**.

🧠 Important Interview Points

✔ Allowed:

- Constructor Overloading
- Calling one constructor from another using this()
- Calling parent constructor using super()

✖ Not Allowed:

- Constructor Overriding
- Marking constructor as static, final, or abstract

✔ Parent constructor always runs before child constructor.

🏁 Short Summary (Very Easy)

Feature	Constructor Overloading	Constructor Overriding
Allowed?	✔ Yes	✖ No
Where?	Same class	Not possible
Why?	Different parameters	Constructors are not inherited
Purpose	Multiple ways to create object	Not applicable

⭐ What is Constructor Chaining?

Calling **one constructor from another constructor** in the *same class* using **this()**. This removes duplicate code and improves reusability.

⭐ Why use this()?

To avoid **repeated initialization code** inside all constructors.

⭐ RULES of this() — VERY IMPORTANT

1. **this() must be the FIRST statement in a constructor**
2. Used to call **another constructor of the same class**
3. Cannot use super() and this() together (both must be first)
4. Helps eliminate code duplication

⋆ Example 1: Simple Constructor Chaining using this()

```
class Student {

    Student() {
        System.out.println("Default Constructor");
    }

    Student(String name) {
        this();  // calls Student()
        System.out.println("Constructor with Name: " + name);
    }

    Student(String name, int age) {
        this(name);  // calls Student(String)
        System.out.println("Constructor with Name & Age: " + name + ", " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        new Student("Aniket", 25);
    }
}
```

OUTPUT

```
Default Constructor
Constructor with Name: Aniket
Constructor with Name & Age: Aniket, 25
```

✓ Student(String, int) → calls
✓ Student(String) → calls
✓ Student() → final

⋆ How Constructor Chaining works (Flow Diagram)

```
new Student("Aniket", 25)
       ↓
this(name, age)
       ↓
this(name)
       ↓
this()
```

⋆ Example 2: Removing Duplicate Code

✕ Without this() → duplicate initializations

```
class Car {
    String model;
    int year;

    Car(String model) {
        this.model = model;
        System.out.println("Model: " + model);
    }

    Car(String model, int year) {
        this.model = model;
        this.year = year;     // repeated code
        System.out.println("Model: " + model + ", Year: " + year);
    }
}
```

✓ With this() → clean & reusable

```
class Car {
    String model;
    int year;

    Car(String model) {
        this.model = model;
        System.out.println("Model: " + model);
    }

    Car(String model, int year) {
        this(model);  // reuse initialization logic
        this.year = year;
        System.out.println("Year: " + year);
    }
}
```

⋆ Example 3: Calling ALL constructors in chain

```
class Demo {

    Demo() {
        System.out.println("1. No-Arg Constructor");
    }

    Demo(int a) {
        this();  // calls no-arg
        System.out.println("2. One-Arg Constructor");
    }

    Demo(int a, int b) {
        this(a);  // calls one-arg
        System.out.println("3. Two-Arg Constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        new Demo(10, 20);
    }
}
```

Output

```
1. No-Arg Constructor
2. One-Arg Constructor
3. Two-Arg Constructor
```

⋆ Common Errors to Remember

✕ this() cannot be after any statement

```
Demo() {
    System.out.println("Hi");
    this(10); // ERROR
}
```

✕ Cannot use super() and this() together

```
Demo() {
    super();   // OK
    this(10);  // ✕ ERROR
}
```

⋆ When to use Constructor Chaining?

Use when:

✓ Many constructors have common logic
✓ You want clean initialization
✓ You want avoid duplicate code
✓ You want flexible object creation with multiple parameter options

---

✅ Use of super keyword with Constructor (in Java)

with rules + examples + outputs.

⋆ What is super() in constructor?

super() is used inside a child class constructor **to call the parent class constructor**. Whenever an object of the child class is created, the parent part of the object must be initialized first. This is done using **super()**.

⋆ Why is super() needed?

Because:

- Child class contains parent class properties
- Parent part must be created first
- Child constructor **automatically calls** parent constructor

---

⋆ Rule 1: super() is ALWAYS the first statement

```
Child() {
    super();  // must be first line
    System.out.println("Child Constructor");
}
```

If anything is above super(), it gives an error.

★ Rule 2: If you don't call super() explicitly → compiler adds it automatically

```
Child() {
    // compiler inserts super();
    System.out.println("Child");
}
```

★ Rule 3: You can call specific parent constructor using super(arguments)

If parent has **overloaded constructors**, child can choose which one to call.

★ Rule 4: If parent has NO default constructor → child MUST call super() manually

Very important interview question.

```
class Parent {
    Parent(int x) { }
}

class Child extends Parent {
    Child() {
        // super() inserted automatically → ERROR (because Parent() doesn't exist)
    }
}
```

✓ You must fix it like this:

```
Child() {
    super(10);
}
```

★ Rule 5: this() and super() CANNOT be used together in the same constructor

(Both must be first line)

```
Child() {
    this();    // OK
    // super(); // ✕ NOT allowed
}
```

★ Example 1: Basic use of super()

```
class Parent {
    Parent() {
        System.out.println("Parent Constructor");
    }
}

class Child extends Parent {
    Child() {
        super();  // calling Parent()
        System.out.println("Child Constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        new Child();
    }
}
```

Output

```
Parent Constructor
Child Constructor
```

★ Example 2: super() calling parameterized parent constructor

```
class A {
    A(int x) {
        System.out.println("Parent x: " + x);
    }
}

class B extends A {
    B() {
        super(100);
        System.out.println("Child Constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        new B();
    }
}
```

Output

```
Parent x: 100
Child Constructor
```

★ Example 3: Parent has multiple constructors → choose one

```
class Vehicle {
    Vehicle() {
        System.out.println("Vehicle default");
    }
    Vehicle(String name) {
        System.out.println("Vehicle: " + name);
    }
}

class Car extends Vehicle {
    Car() {
        super("BMW");   // choose parameterized constructor
        System.out.println("Car ready");
    }
}

public class Main {
    public static void main(String[] args) {
        new Car();
    }
}
```

Output

```
Vehicle: BMW
Car ready
```

⋆ Example 4: Constructor Chaining with super()

```
class A {
    A() { System.out.println("A"); }
}

class B extends A {
    B() {
        super();       // calls A
        System.out.println("B");
    }
}

class C extends B {
    C() {
        super();       // calls B
        System.out.println("C");
    }
}

public class Main {
    public static void main(String[] args) {
        new C();
    }
}
```

Output

```
A
B
C
```

⋆ Real-Life Example

```
class Person {
    Person(String name) {
        System.out.println("Person: " + name);
    }
}

class Employee extends Person {
    Employee(String name, int id) {
        super(name);       // parent constructor call
        System.out.println("Employee ID: " + id);
    }
}

public class Main {
    public static void main(String[] args) {
        new Employee("Aniket", 101);
    }
}
```

Output

```
Person: Aniket
Employee ID: 101
```

🧠 Interview Points to Remember

- super() calls the **parent constructor**
- Must be **first statement**
- If parent has no default constructor → child MUST use super(args)
- Cannot use super() & this() in same constructor
- Parent constructor always runs before child constructor

✅ 1) Constructors in Abstract Classes

👉 **Rule:**
Abstract classes *can have constructors*, even though they cannot be instantiated directly.

✔ Why abstract classes have constructors?

Because constructors are used to:

- initialize common fields
- perform setup tasks
- run before the child class constructor

Even though **we cannot create an object of an abstract class**,
its constructor **runs when a child class object is created**.

📌 Example: Abstract Class with Constructor

```
abstract class Animal {
    String name;

    Animal(String name) {
        System.out.println("Animal constructor called");
        this.name = name;
    }
}

class Dog extends Animal {

    Dog() {
        super("Tommy");  // MUST call abstract class constructor
        System.out.println("Dog constructor called");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
    }
}
```

▶ Output:

```
Animal constructor called
Dog constructor called
```

🎯 Key Points for Abstract Class Constructors

1. **Abstract class can contain constructor**
2. **Child class must call it** (explicitly or implicitly with super())
3. **Abstract constructor executes first**
4. **Abstract class cannot be instantiated directly**

⊘ Wrong:

```
Animal a = new Animal();  // ERROR: cannot instantiate abstract class
```

📌 Real Time Use

Abstract class constructor is used to initialize **common properties:**

```
abstract class DBConnection {
    DBConnection() {
        System.out.println("Connecting to database...");
    }
}
```

Child class automatically gets this feature.

✅ 2) Constructors in Interface

☞ Rule:

❌ Interfaces cannot have constructors

Why?

- Interfaces cannot store state (before Java 8)
- Interfaces cannot be instantiated
- Interfaces are purely abstract types

Therefore:

```
interface Test {
    Test() { }   // ❌ ERROR! Constructors not allowed
}
```

❓ Why NO constructors in interfaces?

Because:

1. Interface **does not have instance variables to initialize**
2. Interface **does not have object creation process**
3. Interface **only contains rules/contracts**, not implementation
4. Constructor is tied to **object creation**, but interfaces cannot create objects

---

✓ But Interfaces Can Have…

✓ static methods

✓ default methods

✓ static blocks (from Java 8)

but still **no constructors**.

🔥 Bonus: Interface Implementation with Constructor

Even though interface cannot have constructor,
the **class implementing the interface can have constructors**.

Example:

```
interface Vehicle {
    void start();
}

class Car implements Vehicle {

    Car() {
        System.out.println("Car constructor called");
    }

    public void start() {
        System.out.println("Car started");
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.start();
    }
}
```

🎯 Final Summary (Interview Ready)

Feature	Abstract Class	Interface
Constructor allowed?	**YES**	**NO**
Why?	To initialize common fields	Interfaces cannot have state or objects
When constructor runs?	When child class object is created	Never (no constructor)
Can we create object?	✕ No	✕ No

Feature	Abstract Class	Interface
Who calls the constructor?	Child class using super()	Not applicable

✅ Constructors With Exception Handling — Full Explanation

Constructors can involve exception handling just like methods.
You can:

- throw exceptions from a constructor
- handle exceptions inside a constructor (try–catch)
- force the calling code to handle exceptions

🔥 1. Constructors can throw checked exceptions

A constructor can declare **throws** just like a method.

📌 Example: Throwing a checked exception

```
class FileHandler {
    FileHandler(String fileName) throws IOException {
        if(fileName == null) {
            throw new IOException("File name cannot be null");
        }
        System.out.println("File opened: " + fileName);
    }
}

public class Main {
    public static void main(String[] args) {
        try {
            FileHandler fh = new FileHandler(null);
        } catch (IOException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

▶ Output:

```
Exception caught: File name cannot be null
```

🔥 2. Constructors can throw unchecked exceptions

Unchecked exceptions (RuntimeException) do not require throws.

```
class BankAccount {
    int balance;

    BankAccount(int balance) {
        if(balance < 0) {
            throw new IllegalArgumentException("Balance cannot be negative");
        }
        this.balance = balance;
    }
}
```

🔥 3. Handling exceptions INSIDE the constructor (try–catch)

Sometimes constructor wants to handle the problem internally.

```
class Connection {
    Connection() {
        try {
            int x = 10 / 0;  // exception
        } catch (Exception e) {
            System.out.println("Handled inside constructor: " + e);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Connection c = new Connection();
    }
}
```

▶ **Output:**

```
Handled inside constructor: java.lang.ArithmeticException: / by zero
```

🔥 4. When constructor fails → object is NOT created

If an exception escapes the constructor (not handled),
object creation is **aborted**.

```
class Test {
    Test() {
        System.out.println("Start");
        String s = null;
        System.out.println(s.length());  // exception
        System.out.println("End");       // NOT executed
    }
}

public class Main {
    public static void main(String[] args) {
        new Test();   // object not created
    }
}
```

🔥 5. Using try–catch when creating object

When constructor throws exception → caller must handle it:

```
try {
    Test t = new Test();
} catch(Exception e) {
    System.out.println("Constructor failed");
}
```

🔥 6. Exception handling in Constructor Chaining

If you use this() (same class) constructor chains:

```
class Demo {
    Demo() throws Exception {
        this(10); // calling other constructor
    }

    Demo(int x) throws Exception {
        if(x < 0) throw new Exception("Negative value");
    }
}
```

☞ If **any constructor in the chain** throws an exception,
the whole chain must declare **throws** or handle it.

🔥 7. Exception handling with super()

If parent constructor throws an exception → child must handle or declare.

```
class Parent {
    Parent() throws Exception {
        throw new Exception("Parent problem");
    }
}

class Child extends Parent {
    Child() throws Exception {
        super();  // must handle or declare
    }
}
```

🎯 Important Interview Points

1. ✓ Constructors can use **throws**
2. ✓ Constructors **can throw checked or unchecked exceptions**
3. ✓ If constructor throws an exception → **object is not created**
4. ✓ Exception rules are same as methods
5. ✓ this() and super() must be first line → try–catch cannot be before them
6. ✓ Exception in parent constructor must be handled by child

---

✅ What is a Copy Constructor in Java?

Java **does NOT have built-in copy constructor** (like C++),
but we **create our own constructor** that copies the values of one object into another.

☞ Definition:

A **copy constructor** is a constructor that takes **another object of the same class** as a parameter and copies all its fields.

🧩 Syntax

```
class ClassName {
    ClassName(ClassName obj) {
        // copy code
    }
}
```

🔥 Simple Example of Copy Constructor

```
class Student {
    String name;
    int age;

    // Normal constructor
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Copy constructor
    Student(Student s) {
        this.name = s.name;
        this.age = s.age;
    }
}

public class Main {
    public static void main(String[] args) {

        Student s1 = new Student("Amit", 22);

        // Using copy constructor
        Student s2 = new Student(s1);

        System.out.println(s2.name);  // Amit
        System.out.println(s2.age);   // 22
    }
}
```

🏆 Benefits of Copy Constructor

1. Creates a **separate object** with same data
2. Protects original object from change
3. Useful for **cloning / duplicating** objects
4. Easy to customize (deep copy, shallow copy)

🔥 Shallow Copy vs Deep Copy in Copy Constructor

1️⃣ Shallow Copy (default)

Copies only primitive fields and references.

```
class Person {
    StringBuilder name;

    Person(StringBuilder name) {
        this.name = name;
    }
    Person(Person p) {        // shallow copy
        this.name = p.name;
    }
}
```

Changing name in one object affects the other — because same reference.

2️⃣ Deep Copy

Creates a new object for reference fields.

```
class Person {
    StringBuilder name;

    Person(StringBuilder name) {
        this.name = new StringBuilder(name);
    }

    Person(Person p) {        // deep copy
        this.name = new StringBuilder(p.name.toString());
    }
}
```

Now changes in one object do **not** affect the other.

🔥 Copy Constructor vs Clone()

Feature	Copy Constructor	clone()
Type	User-defined	Inherited from Object
Easy to read?	✓ Yes	✗ No
Exception	No	throws CloneNotSupportedException
Customization	✓ Very easy	Hard
Recommended?	✓ Yes	✗ No (old style)

Interview Answer:
➡ Java developers prefer **copy constructors** over clone().

🔥 When to Use Copy Constructor?

- To duplicate an object safely
- To avoid problems of clone()
- When implementing deep copy
- When working with immutable objects
- For DTOs, entities, models

⭐ Final Simple Example (Deep Copy)

```
class Address {
    String city;

    Address(String city) {
        this.city = city;
    }
}

class Employee {
    String name;
    Address address;

    Employee(String name, Address address) {
        this.name = name;
        this.address = new Address(address.city); // deep copy
    }

    Employee(Employee e) {
        this.name = e.name;
        this.address = new Address(e.address.city); // deep copy
    }
}
```