

Constructor

1. What is a Constructor

Ans1. A **constructor** in Java is a special method that is used to initialize objects. It is called automatically when an object of a class is created. The constructor has the same name as the class and does not have a return type (not even **void**).

Types of Constructors in Java

1. Default Constructor (No-Arg Constructor)

- A constructor that does not take any arguments.
- If no constructor is defined, Java provides a default constructor.

```
class Example {
```

```
Example() { // Default constructor
```

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

```
}
```

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

```
Example obj = new Example(); // Constructor is called automatically
```

```
}
```

```
}
```

Output:

Default Constructor called!

2. Parameterized Constructor

- A constructor that takes arguments to initialize an object with specific values.

java

```
class Person {
```

```
    String name;
```

```
    int age;
```

```
    // Parameterized constructor
```

```
    Person(String n, int a) {
```

```
name = n;
```

```
age = a;
```

```
}
```

```
void display() {
```

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

```
}
```

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

```
    Person p1 = new Person("Alice", 25);
```

```
    p1.display();
```

```
}
```

```
}
```

Output:

Name: Alice, Age: 25

3. Copy Constructor (Java does not have a built-in copy constructor, but we can implement one manually.)

```
class Student {  
  
    String name;  
  
    int age;  
  
    // Parameterized constructor  
  
    Student(String n, int a) {  
  
        name = n;  
  
        age = a;  
  
    }  
}
```

```
// Copy constructor
```

```
Student(Student s) {
```

```
    name = s.name;
```

```
    age = s.age;
```

```
}
```

```
void display() {
```

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

```
}
```

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

```
    Student s1 = new Student("John", 22);
```

```
Student s2 = new Student(s1); // Copying s1 into s2
```

```
s2.display();
```

```
}
```

```
}
```

```
class Student {
```

```
String name;
```

```
int age;
```

```
// Parameterized constructor
```

```
Student(String n, int a) {
```

```
    name = n;
```

```
    age = a;
```

```
}
```

```
// Copy constructor
```

```
Student(Student s) {
```

```
    name = s.name;
```

```
    age = s.age;
```

```
}
```

```
void display() {
```

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

```
}
```

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

```
    Student s1 = new Student("John", 22);
```

```
Student s2 = new Student(s1); // Copying s1 into s2
```

```
s2.display();
```

```
}
```

```
}
```

Output:

Name: John, Age: 22

Key Points about Constructors in Java

- Constructors **do not have a return type**.
- Constructors **must have the same name** as the class.
- If no constructor is defined, Java **automatically provides a default constructor**.
- Constructors can be **overloaded** (i.e., multiple constructors with different parameters).
- A constructor can **call another constructor** using **this()**.
- The **super()** call in a constructor is used to call the parent class's constructor.

2. What is Constructor Chaining.

Ans2. **Constructor Chaining in Java**

Constructor Chaining is the process of calling one constructor from another within the same class or from a parent class using **this()** or **super()**.

Types of Constructor Chaining

1. **Within the Same Class** → Using `this()`
 2. **From Parent Class (Superclass)** → Using `super()`
-

1. Constructor Chaining Within the Same Class (Using `this()`)

When one constructor calls another constructor of the same class, it is called **constructor chaining within the same class**. This is done using `this()`.

Example:

```
class Person {  
  
    String name;  
  
    int age;  
  
  
    // Constructor 1: Default Constructor  
  
    Person() {  
  
        this("Unknown", 0); // Calls Constructor 2  
  
        System.out.println("Default constructor called");  
  
    }  
}
```

```
// Constructor 2: Parameterized Constructor
```

```
Person(String name, int age) {
```

```
    this.name = name;
```

```
    this.age = age;
```

```
    System.out.println("Parameterized constructor called");
```

```
}
```

```
void display() {
```

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

```
}
```

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

```
Person p1 = new Person(); // Calls default constructor
```

```
p1.display();
```

```
}
```

```
}
```

Output:

Parameterized constructor called

Default constructor called

Name: Unknown, Age: 0

◆ `this("Unknown", 0)` calls the parameterized constructor before executing the body of the default constructor.

2. Constructor Chaining from Parent Class (Using `super()`)

When a subclass constructor calls a constructor of its **superclass**, it is called **constructor chaining across inheritance**. This is done using `super()`.

Example:

```
class Animal {
```

```

String species;

// Constructor of superclass
Animal(String species) {
    this.species = species;
    System.out.println("Animal Constructor Called");
}
}

class Dog extends Animal {
    String breed;

    // Constructor of subclass
    Dog(String species, String breed) {
        super(species); // Calls the constructor of Animal
        this.breed = breed;
        System.out.println("Dog Constructor Called");
    }

    void display() {
        System.out.println("Species: " + species + ", Breed: " + breed);
    }

    public static void main(String[] args) {
        Dog d = new Dog("Mammal", "Labrador");
        d.display();
    }
}

```

Output:

```

Animal Constructor Called
Dog Constructor Called
Species: Mammal, Breed: Labrador

```

- ◆ `super(species)` calls the `Animal` class constructor **before** executing the `Dog` constructor.

Key Points About Constructor Chaining

- ✓ **this()** → Calls another constructor within the same class.
- ✓ **super()** → Calls a constructor of the parent class.
- ✓ **this()** and **super()** must be the first statement in a constructor.
- ✓ If a superclass has a parameterized constructor, the subclass must explicitly call it using **super()**.

3. Can we call a subclass constructor from a superclass constructor.

Ans3. No, a **superclass constructor cannot call a subclass constructor**. In Java, the constructor chaining always **moves upward in the inheritance hierarchy**, meaning that:

- A **subclass constructor** can call a **superclass constructor** using **super()**.
- But a **superclass constructor cannot call a subclass constructor** because the subclass does not yet exist when the superclass constructor is executing.

Why Can't a Superclass Call a Subclass Constructor?

1. **Object Creation in Java Follows a Top-Down Approach:**
 - When you create an object of a subclass, the **superclass constructor is always called first**.
 - The subclass does not yet exist at this point, so calling its constructor is impossible.
2. **Avoiding Infinite Recursion:**
 - If a superclass constructor tried to call a subclass constructor, and the subclass constructor called **super()**, it would create an **infinite loop**, causing a runtime error.

Example: What Happens If We Try?

```
class Parent {  
    Parent() {  
        System.out.println("Parent Constructor Called");  
        // Attempting to call a subclass constructor (NOT ALLOWED)  
        // new Child(); // This will cause logical issues  
    }  
}
```

```

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

    public static void main(String[] args) {
        Child obj = new Child();
    }
}

```

Output:

Parent Constructor Called
Child Constructor Called

Why does `new Child();` inside `Parent()` cause problems?

If you uncomment `new Child();` inside the `Parent` constructor, it would:

1. Call the `Parent` constructor.
2. Inside the `Parent` constructor, a new `Child` object is created.
3. This calls `Child()`, which again calls `super()` (Parent constructor).
4. This repeats indefinitely → **StackOverflowError**.

4. What happens if you keep a return type for a constructor.

Ans4. If you specify a **return type** for a constructor in Java, it will **no longer be treated as a constructor**, but instead as a regular method.

Example: What Happens if We Add a Return Type?

```

class Example {

    // Looks like a constructor, but it has a return type (int)

    int Example() {

        System.out.println("This is not a constructor, it's a
method.");

        return 0;

    }
}

```

```

public static void main(String[] args) {
    Example obj = new Example(); // Calls default
    constructor
    obj.Example(); // Calls the method
}
}

```

Output:

This is not a constructor, it's a method.

5. What is No-arg constructor .

Ans5. **No-Arg Constructor in Java**

A **no-arg constructor** is a constructor that **does not take any parameters**. It is used to initialize an object with default values.

Example of a No-Arg Constructor

```

class Example {
    // No-arg constructor
    Example() {
        System.out.println("No-arg constructor called!");
    }

    public static void main(String[] args) {
        Example obj = new Example(); // Calls the no-arg
        constructor
    }
}

```

Output:

No-arg constructor called!

Example: No-Arg Constructor vs. Parameterized Constructor

```
class Person {  
    String name;  
  
    // No-arg constructor  
    Person() {  
        name = "Unknown";  
        System.out.println("No-arg constructor called");  
    }  
  
    // Parameterized constructor  
    Person(String name) {  
        this.name = name;  
        System.out.println("Parameterized constructor called");  
    }  
  
    void display() {  
        System.out.println("Name: " + name);  
    }  
  
    public static void main(String[] args) {  
        Person p1 = new Person(); // Calls no-arg constructor  
        p1.display();  
    }  
}
```



```
    Person p2 = new Person("Alice"); // Calls  
parameterized constructor
```

```
    p2.display();  
}  
}
```

Output:

No-arg constructor called

Name: Unknown

Parameterized constructor called

Name: Alice

6. How is a No-argument constructor different from the default Constructor

Ans6. Example of a No-Argument Constructor

```
class Person {  
    String name;  
  
    // Explicit no-argument constructor  
    Person() {  
        name = "Default Name";  
        System.out.println("No-arg constructor called");  
    }  
  
    void display() {  
        System.out.println("Name: " + name);  
    }  
}
```

```
public static void main(String[] args) {  
    Person p = new Person(); // Calls the no-arg constructor  
    p.display();  
}  
}
```

Output:

No-arg constructor called

Name: Default Name

Example of a Default Constructor

```
class Student {  
    String name;  
  
    // No constructor defined → Java automatically provides a  
    // default constructor  
  
    void display() {  
        System.out.println("Name: " + name); // Will print  
        'null' (default value)  
    }  
  
    public static void main(String[] args) {  
        Student s = new Student(); // Calls default constructor  
        s.display();  
    }  
}
```

```
}
```

Output:

Name: null

Key Differences

1. **Customization** → A **no-arg constructor** can have custom logic, while a **default constructor** only initializes fields with default values.
2. **Explicit vs. Implicit** → A **no-arg constructor** must be explicitly written, whereas a **default constructor** is **automatically generated** if no other constructor exists.
3. **Overriding** → If a parameterized or no-arg constructor is present, **Java does not generate a default constructor**.

7. When do we need Constructor Overloading

Ans7. **When Do We Need Constructor Overloading in Java?**

Constructor overloading is needed when we want to create multiple ways to initialize an object **with different sets of data**. This allows flexibility in object creation.

Situations Where Constructor Overloading is Useful

1. **Providing Multiple Ways to Initialize an Object**
 - Some objects may need different levels of initialization.
2. **Reducing Code Duplication**
 - Instead of creating multiple methods for initialization, different constructors can be used.
3. **Improving Code Readability**
 - It is easier to understand how objects are created with different parameters.
4. **Handling Default Values**
 - Some constructors can assign default values while others take user input.

Example 1: Multiple Ways to Initialize an Object

```
class Person {  
    String name;  
    int age;  
  
    // No-argument constructor  
    Person() {  
        name = "Unknown";  
        age = 0;  
    }  
  
    // Constructor with one parameter  
    Person(String name) {  
        this.name = name;  
        this.age = 18; // Default age  
    }  
  
    // Constructor with two parameters  
    Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    void display() {  
        System.out.println("Name: " + name + ", Age: " + age);  
    }  
  
    public static void main(String[] args) {
```

```
Person p1 = new Person(); // Calls no-arg constructor
```

```
Person p2 = new Person("Alice"); // Calls constructor  
with 1 argument
```

```
Person p3 = new Person("Bob", 25); // Calls  
constructor with 2 arguments
```

```
p1.display();  
p2.display();  
p3.display();  
}  
}
```

Output:

Name: Unknown, Age: 0

Name: Alice, Age: 18

Name: Bob, Age: 25

Example 2: Avoiding Code Duplication Using **this()**

Instead of repeating code, we can call one constructor from another using **this()**.

```
class Car {  
    String brand;  
    int speed;  
  
    // No-argument constructor  
    Car() {  
        this("Unknown", 0); // Calls parameterized constructor
```

```
}

// Parameterized constructor
Car(String brand, int speed) {
    this.brand = brand;
    this.speed = speed;
}

void display() {
    System.out.println("Brand: " + brand + ", Speed: " +
speed);
}

public static void main(String[] args) {
    Car car1 = new Car(); // Calls no-arg constructor,
which calls the parameterized one

    Car car2 = new Car("Tesla", 120); // Calls
parameterized constructor

    car1.display();
    car2.display();
}
}
```

Output:

Brand: Unknown, Speed: 0

Brand: Tesla, Speed: 120

8. What is Default constructor Explain with an Example

Ans8. **What is a Default Constructor in Java?**

A **default constructor** is a constructor that **takes no arguments** and is **automatically provided by Java** if no constructor is explicitly defined in a class.

Key Characteristics of a Default Constructor

- ✓ **No Parameters:** It does not take any arguments.
 - ✓ **Automatically Provided by Java:** If no constructor is defined, Java generates a default constructor.
 - ✓ **Initializes Object with Default Values:**
 - Numeric fields (**int**, **double**, etc.) → **0**
 - Boolean fields → **false**
 - Object references (**String**, arrays, etc.) → **null**
-

Example 1: Default Constructor (Generated by Java)

If no constructor is defined, Java provides a **default constructor** automatically.

```
class Student {  
    String name;  
    int age;  
  
    // No constructor defined → Java generates a default  
    constructor  
  
    void display() {  
        System.out.println("Name: " + name); // Default is null  
        System.out.println("Age: " + age); // Default is 0  
    }  
}
```

```
}
```

```
public static void main(String[] args) {  
    Student s = new Student(); // Calls default constructor  
    s.display();  
}
```

```
}
```

Output:

Name: null

Age: 0

Example 2: Explicitly Defining a Default Constructor

You can **explicitly define a default constructor** if you want to **customize initialization**.

```
class Person {  
    String name;  
    int age;  
  
    // Explicitly defined default constructor  
    Person() {  
        name = "Unknown";  
        age = 18;  
        System.out.println("Default constructor called");  
    }  
  
    void display() {
```



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

    public static void main(String[] args) {
        Person p = new Person(); // Calls the default
        constructor
        p.display();
    }
}
```

Output:

Default constructor called

Name: Unknown, Age: 18

When is a Default Constructor Needed?

1. **If a class has no constructors, Java generates a default constructor automatically.**
2. **Required by frameworks like Hibernate, Spring, and JPA** for object creation via reflection.
3. **Useful in inheritance**, where a child class depends on the parent class's default constructor.