**🔹 Can 2 objects have the same hashCode?**

👉 **Yes, it is possible**. This situation is called a **hash collision**.

**1. JVM’s default hashCode() (from Object class)**

* By default, JVM gives each object a hashcode that is **derived from the memory address** of the object.
* Since two objects normally sit in different memory locations, their hashcodes will **usually** be different.
* But still, two different memory addresses *could* result in the same hashcode, because hashcodes are just integers and there are limited integers compared to memory addresses.  
  ⚡ So collisions are *rare* but **possible** even in JVM default.

**2. When we override hashCode() (like Lombok does)**

* Instead of memory addresses, we calculate hashCode based on **field values**.
* If two objects have the same values in their fields, their hashcodes will be **identical**.
* If two objects have different values, they may still (rarely) produce the same hashcode because of hash collisions in the hashing algorithm.

## 🔹 Can an object have 2 hashCodes?

👉 Yes, you can think of **two types of hashCodes** for the same object:

1. **Default JVM identity hashcode**
   * Generated from the object’s memory location.
   * Always available, even if we override hashCode().
   * Can be accessed with:
   * System.identityHashCode(obj);
2. **Overridden/custom hashCode**
   * If we override hashCode() (e.g., with Lombok), this one is based on the **field values (state)** of the object.
   * This is what gets used in collections like HashMap, HashSet.

✅ So technically, the same object can give you **two different numbers** depending on which method you call (hashCode() vs System.identityHashCode()).

**🔎 What is a Constructor**

In Java, a **constructor** is a special method used to **create objects**.

Example without Lombok:

public class Student {

private String name;

private int age;

// No-args constructor

public Student() { }

// All-args constructor

public Student(String name, int age) {

this.name = name;

this.age = age;

}

}

➡️ We had to **write these constructors manually**.

Now let’s see how **Lombok** makes this shorter.

**🟢 @NoArgsConstructor**

* Generates a **constructor with no parameters**.
* That means we can create an object **without passing any values**.
* Example:
* @NoArgsConstructor
* public class Student {
* private String name;
* private int age;
* }

👉 What Lombok secretly generates for us:

public Student() { }

➡️ **When is it useful?**

* Frameworks like **Hibernate, JPA, or Jackson** often require a default constructor to create objects by themselves.
* Even if we don’t write one, Lombok ensures it exists.

**🟢 @AllArgsConstructor**

* Generates a **constructor that takes one parameter for each field**.
* Example:
* @AllArgsConstructor
* public class Student {
* private String name;
* private int age;
* }

👉 What Lombok generates:

public Student(String name, int age) {

this.name = name;

this.age = age;

}

➡️ **When is it useful?**

* When we want to create an object and **immediately set all values** in one line.
* Student s = new Student("John", 20);

**🟢 Special Case → No Fields in Class**

If a class has **no fields at all**:

@NoArgsConstructor

@AllArgsConstructor

public class EmptyClass { }

👉 Both annotations generate the same thing:

public EmptyClass() { }

➡️ Because there are **no fields**, an "all-args constructor" has **no arguments**, which is the same as a no-args constructor.

**✅ Final Summary (easy + deep)**

* **@NoArgsConstructor** → Creates an **empty constructor** (no inputs).
* **@AllArgsConstructor** → Creates a **constructor with inputs for all fields**.
* **If no fields exist** → Both end up generating the **same constructo**

**🔎 Default Constructor vs No-Args Constructor**

* In Java, the **compiler itself** creates a **0-parameter constructor** **only when** you don’t write any constructor at all.
* class Student {
* int age;
* }

👉 Compiler secretly adds:

Student() { }

This is called the **default constructor**.

* If **we** write a **0-parameter constructor** ourselves, or Lombok generates one using @NoArgsConstructor,  
  👉 it is **not called a default constructor** anymore.  
  It’s simply a **no-argument constructor**.

⚠️ Difference is **who creates it**:

* Compiler → Default Constructor
* Developer or Lombok → No-Args Constructor

**🔎 Default Values vs Explicit Initialization**

* In Java, instance variables (fields in a class) get **default values** if not initialized.
  + int a; → default = 0
  + boolean b; → default = false
  + String s; → default = null
* But if you **assign a value yourself**, then it’s **explicit initialization**, not a default value.
* int a = 0; // here, 0 is not a "default value", it's explicitly assigned

**🔎 Overloading vs Overriding Constructor**

* **Overriding** means a child class provides its own version of a method from the parent class (same name, same parameters).  
  👉 But **constructors are not inherited**, so **constructors cannot be overridden**.
* **Overloading** means having **multiple constructors in the same class with different parameter lists**.  
  👉 Yes, constructors can be **overloaded**.

**🟢 @RequiredArgsConstructor (Lombok)**

* This annotation generates a **constructor only for specific fields**.
* Which fields?
  1. **All final fields**
  2. All fields annotated with **@NonNull**

So, it gives you a **"required fields" constructor**.

**✅ Example 1: With final**

import lombok.RequiredArgsConstructor;

@RequiredArgsConstructor

public class Student {

private final String name; // final → included in constructor

private int age; // not included

}

👉 Lombok generates:

public Student(String name) {

this.name = name;

}

**✅ Example 2: With @NonNull**

import lombok.RequiredArgsConstructor;

import lombok.NonNull;

@RequiredArgsConstructor

public class Student {

@NonNull

private String name; // included in constructor

private int age; // not included

}

👉 Lombok generates:

public Student(@NonNull String name) {

this.name = name;

}

⚠️ Also, Lombok will insert a **null-check** for @NonNull fields and throw a NullPointerException if you pass null.

**✅ Example 3: With both final + @NonNull**

@RequiredArgsConstructor

public class Student {

private final int rollNo; // included

@NonNull

private String name; // included

private int age; // not included

}

👉 Generated constructor:

public Student(int rollNo, @NonNull String name) {

this.rollNo = rollNo;

this.name = name;

}

**✅ Example 4: No final and no @NonNull**

@RequiredArgsConstructor

public class Student {

private String name;

private int age;

}

👉 Generated constructor:

public Student() { }

➡️ Because there are **no required fields**, Lombok just makes a **0-args constructor**.

## 🔎 1. Var-args Constructor with Lombok

* Lombok **cannot generate constructors with var-args** (... syntax).
* Example:
* public Employee(String... names) { } // ❌ Lombok cannot generate this
* If you need var-args, you must write it manually.

## 2. Multiple “Custom” Parameterized Constructors

* Lombok can generate:
  + **No-args** constructor (@NoArgsConstructor)
  + **All-args** constructor (@AllArgsConstructor)
  + **Required-args** constructor (@RequiredArgsConstructor)

But ⚠️ **you cannot tell Lombok to generate multiple custom versions like:**

* one constructor with 1 parameter,
* another with 2 parameters,
* another with 3 parameters, etc.

👉 If you need such **overloaded constructors**, you must **write them yourself**.

# 📝 Notes on **Constructor Access Levels** in Java (with Lombok support)

## 🔎 1. **Public Constructor**

* Most common case.
* Accessible everywhere (same class, same package, different package).
* Used for **regular POJOs, DTOs, model/entity classes**.

### Example:

@NoArgsConstructor(access = AccessLevel.PUBLIC)

@AllArgsConstructor(access = AccessLevel.PUBLIC)

public class Student {

private String name;

private int age;

}

✅ Usage:

Student s1 = new Student(); // no-args

Student s2 = new Student("John", 20); // all-args

## 🔎 2. **Protected Constructor**

* Accessible only in:
  + The **same package**
  + **Subclasses** (even if they are in a different package)
* Used when you **don’t want external classes** to create objects directly, but **inheritance** should still work.

### Example:

@RequiredArgsConstructor(access = AccessLevel.PROTECTED)

public abstract class Shape {

private final String type;

}

class Circle extends Shape {

public Circle() {

super("Circle"); // ✅ allowed

}

}

❌ Outside code cannot do:

new Shape("Triangle"); // Not allowed

✅ Only subclasses (like Circle) or same-package classes can.

## 🔎 3. **Private Constructor** 🔥 (Very Important)

* Accessible **only inside the same class**.
* ❌ Cannot be called from outside classes or subclasses.
* Used for **controlling object creation**.

### ✅ When to use:

1. **Singleton Pattern** → Only one instance allowed.
2. **Factory Pattern** → Control how objects are created.
3. **Utility Classes** → Prevent object creation (all static methods).

### Example 1: Singleton with Lombok

@AllArgsConstructor(access = AccessLevel.PRIVATE)

public class Config {

private String url;

private String username;

private static final Config INSTANCE = new Config("jdbc://", "admin");

public static Config getInstance() {

return INSTANCE;

}

}

✅ Usage:

Config c = Config.getInstance(); // Allowed

❌ Not allowed:

Config c = new Config("jdbc://", "user"); // ERROR, constructor is private

### Example 2: Utility Class

@NoArgsConstructor(access = AccessLevel.PRIVATE)

public class MathUtil {

public static int square(int x) {

return x \* x;

}

}

✅ Usage:

int result = MathUtil.square(5);

❌ Not allowed:

MathUtil m = new MathUtil(); // ERROR, constructor is private

This prevents accidental object creation.

## 🔎 4. **Package-Private (Default)** ⚡ (a bit tricky but powerful)

* If you don’t write any modifier, constructor is **package-private**.
* Usable **only within the same package**.
* Not accessible from **outside the package**, even in subclasses.
* Good for **frameworks or internal APIs** where you want to restrict usage.

### Example:

@NoArgsConstructor(access = AccessLevel.PACKAGE)

class Employee {

String name;

}

✅ Allowed (inside the same package):

Employee e = new Employee();

❌ Not allowed (outside the package):

import com.company.Employee;

Employee e = new Employee(); // ERROR, constructor is package-private

### ✅ When to use Package-Private:

* You want to **hide implementation** details from outside code.
* Example: JDK collections often use this to keep constructors package-scoped and force you to use **factory methods** (List.of(), Map.of(), etc.).

# 🔎 Final Comparison Table

| **Access Level** | **Keyword** | **Where Usable** | **Typical Use Case** |
| --- | --- | --- | --- |
| **Public** | public | Everywhere | Normal classes, POJOs |
| **Protected** | protected | Same package + subclasses | Abstract classes, inheritance |
| **Private** | private | Same class only | Singleton, Factory, Utility |
| **Package-Private** | (none) | Same package only | Internal APIs, framework classes |

# 🟢 @Data Annotation (Lombok)

### ✅ What it does

@Data is a **shortcut annotation** that combines several commonly used Lombok annotations in one.  
When you put @Data on a class, Lombok generates:

1. **@Getter** → generates getter methods for **all fields**.
2. **@Setter** → generates setter methods for **all non-final fields**.
3. **@EqualsAndHashCode** → generates equals() and hashCode() methods.
4. **@ToString** → generates toString() method.
5. **@RequiredArgsConstructor** → generates constructor for **all final fields** and fields annotated with @NonNull.

### ✅ Example

import lombok.Data;

import lombok.NonNull;

@Data

public class Employee {

private final int id; // included in RequiredArgsConstructor

@NonNull

private String name; // included in RequiredArgsConstructor

private String department; // has getter & setter

}

👉 Lombok will generate:

// RequiredArgsConstructor (for id and name)

public Employee(int id, @NonNull String name) {

this.id = id;

this.name = name;

}

// Getters

public int getId() { return id; }

public String getName() { return name; }

public String getDepartment() { return department; }

// Setters (not for final field "id")

public void setName(String name) { this.name = name; }

public void setDepartment(String department) { this.department = department; }

// toString()

@Override

public String toString() {

return "Employee(id=" + id + ", name=" + name + ", department=" + department + ")";

}

// equals() & hashCode()

@Override

public boolean equals(Object o) { ... }

@Override

public int hashCode() { ... }

### ✅ Why use @Data?

* Reduces boilerplate code in **model/DTO/entity classes**.
* Gives you a **ready-to-use JavaBean** style object.
* Ensures **equality checks**, **string representation**, and **constructors** are already handled.

### ⚠️ Caveats / Things to Keep in Mind

1. **Final fields don’t get setters** (because they can’t be changed).
2. **@Data uses @RequiredArgsConstructor, not @AllArgsConstructor**.
   * If you want **all fields in constructor**, add @AllArgsConstructor explicitly.
3. **Equals & hashCode**: By default, all non-static fields are included. Sometimes you may want to customize it with @EqualsAndHashCode.Exclude.
4. **ToString**: Includes all fields → may cause problems if object has **sensitive data** (like passwords) or **recursive references**.

### ✅ Example with Caveats

@Data

public class User {

private final int id;

private String username;

private String password; // ⚠️ sensitive field

}

👉 toString() will print the password too 😬

User u = new User(1, "admin");

u.setPassword("secret");

System.out.println(u);

// Output: User(id=1, username=admin, password=secret)

✅ Fix:

@ToString(exclude = "password")

@Data

public class User {

private final int id;

private String username;

private String password;

}

## What does final mean for fields?

In Java, a **final field** means:

* It must be **assigned exactly once**.
* After it is assigned, it **cannot be changed again**.

So:

* **Allowed**: assignment during declaration OR in a constructor.
* **Not allowed**: assignment in setters or later code.