**🔹 What is a Java Bean?**

A **Java Bean** is just a **normal Java class** but written with some specific **standards (rules)**.  
These rules make the class easy to **use, share, and maintain** in real-time projects.

**✅ Rules of a Java Bean**

1. **Class must be public**
   * This means the class should be visible to other classes in the project.
   * If it's not public, other layers/modules may not be able to use it.
2. **Recommended to implement Serializable interface**
   * Serializable means the object can be **converted into a byte stream** (for saving in a file, transferring over network, etc.).
   * Example: If you send an object to another microservice, serialization helps.
3. **Bean property variables should be private and non-static**
   * private → only the class can directly access it (encapsulation).
   * non-static → so that each object has its **own copy** of the data.
   * Example: Each student object should have its own name and age.
4. **Every property should have Setter and Getter methods**
   * Setter → used to set (update) the value.
   * Getter → used to get (read) the value.
   * This gives **controlled access** instead of exposing variables directly.
5. **It should have a zero-parameter constructor (default constructor)**
   * Helps frameworks (Spring, Hibernate, etc.) to easily create objects using reflection.
   * Example: new Student() should work without passing values.

# 📘 Notes – Java Beans in Layered Architecture

### ✅ Types of Java Beans

1. **VO (Value Object)**
   * Purpose: Holds **user inputs** coming from the UI (form data, JSON request body).
   * Example: LoginVO may contain username and password entered by the user.
2. **DTO (Data Transfer Object)**
   * Purpose: Used to **transfer data between layers** of the project.
   * Example: After fetching user details from DB, we create UserDTO with only safe, required fields → send it to the UI.
3. **BO (Business Object)**
   * Purpose: Represents **persistable business data** with business meaning.
   * Example: LoanBO may contain principal, rate, duration, and also methods like calculateEMI().
   * It is processed inside the **Business Layer**, sometimes later mapped to **Entity/Model objects** for DB persistence.

### ✅ Features of a Well-Designed Java Class

A proper Java Bean should contain:

1. **Overloaded Constructors**
   * Default constructor → required by frameworks (Hibernate, Spring).
   * Parameterized constructors → for convenience while creating objects.
2. **toString() Method**
   * Returns a string representation of the object.
   * Useful for logging, debugging, printing objects in readable form.
3. **equals() Method**
   * Used to compare **content/data** of two objects instead of memory reference.
   * Example: Two Student objects with same id should be considered equal.
4. **hashCode() Method**
   * Used with equals() for storing objects in collections like HashMap, HashSet.
   * Ensures that equal objects have the same hash code.
5. **Setters & Getters (optional with Lombok)**
   * Setters: to assign values to fields.
   * Getters: to retrieve field values.
   * Without Lombok → we write them manually.
   * With Lombok → we use annotations like @Getter, @Setter, @Data.

### ✅ Before Lombok API

* We had to **manually add all boilerplate code**:
  + Getters & Setters for each property.
  + toString(), equals(), hashCode().
  + Overloaded constructors.
* This made classes lengthy and repetitive, especially if fields increased or decreased.

# 📘 Notes – Lombok API (Project Lombok)

### ✅ What is Lombok?

* **Lombok (Project Lombok)** is an **open-source Java library**.
* It **automatically generates boilerplate code** (repetitive code) at compile time.
* It reduces the size of Java classes and makes code **cleaner and more readable**.

### ✅ What Lombok Generates Automatically

With Lombok, you don’t need to write:

* Constructors (default & parameterized)
* Setters and Getters
* toString()
* equals()
* hashCode()

👉 All of these are generated **at compile time** using **annotations**.

### ✅ Requirements

* **IDE support**:  
  You must configure/enable Lombok in your IDE (Eclipse, IntelliJ, STS, etc.), otherwise the auto-generated methods may not be recognized during development.
* **Dependency**:  
  In Maven/Gradle projects, add Lombok dependency. Example (Maven):
* <dependency>
* <groupId>org.projectlombok</groupId>
* <artifactId>lombok</artifactId>
* <version>1.18.30</version> <!-- Example version -->
* <scope>provided</scope>
* </dependency>

### ✅ Common Lombok Annotations

1. **@Getter** → generates getter methods for all fields.
2. **@Setter** → generates setter methods for all fields.
3. **@NoArgsConstructor** → generates a no-argument (default) constructor.
4. **@AllArgsConstructor** → generates a constructor with parameters for **all fields**.
5. **@RequiredArgsConstructor** → generates a constructor with only **final or @NonNull fields**.
6. **@ToString** → generates a toString() method including all fields.
7. **@EqualsAndHashCode** → generates both equals() and hashCode() methods.
8. **@Data** → combination of
   * @Getter + @Setter + @RequiredArgsConstructor + @ToString + @EqualsAndHashCode.
   * Commonly used in DTOs, VOs, and BOs.

### ✅ Example

#### 🔹 Without Lombok

public class EmployeeDTO {

private Long id;

private String name;

private double salary;

// Constructor

public EmployeeDTO(Long id, String name, double salary) {

this.id = id;

this.name = name;

this.salary = salary;

}

// Getters and Setters

public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

public String getName() { return name; }

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

public double getSalary() { return salary; }

public void setSalary(double salary) { this.salary = salary; }

// toString

@Override

public String toString() {

return "EmployeeDTO [id=" + id + ", name=" + name + ", salary=" + salary + "]";

}

// equals & hashCode

@Override

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

@Override

public int hashCode() { ... }

}

#### 🔹 With Lombok

import lombok.Data;

@Data

public class EmployeeDTO {

private Long id;

private String name;

private double salary;

}

👉 That’s it. Lombok automatically generates:

* Getters & Setters
* toString()
* equals() & hashCode()
* Required constructor

# 📌 Lombok API – Notes

### 🔹 How Lombok Works

* **Lombok API annotations** instruct the **Java compiler (javac)** to **generate boilerplate code dynamically** in the .class file.
* Even if the .java file does not explicitly contain methods like **getters, setters, constructors, toString(), equals(), hashCode()**, the compiler adds them during compilation.
* Example: When you don’t write a constructor, the compiler **automatically adds a default constructor** and makes java.lang.Object the default superclass. Lombok uses the same mechanism for boilerplate methods.

### 🔹 Retention Level of Lombok

* Lombok annotations have **Retention = SOURCE**.
* This means:
  + Annotations are **only available in source code**, not stored in .class files.
  + However, the **generated methods (getters, setters, etc.) are stored in .class files** by javac.
  + So at runtime, your application sees the generated methods, but **not Lombok annotations**.

### 🔹 Lombok Annotations in Action

#### 1. @Getter and @Setter

* Applied at **class level** → generates **getters and setters** for all fields.
* Applied at **field level** → generates **getter and setter** only for that specific field.

**Example 1 – Class level**

import lombok.Getter;

import lombok.Setter;

@Getter

@Setter

class Student {

private Integer sid;

private String sname;

private Integer sage;

}

👉 Compiler generates:

public Integer getSid() { return sid; }

public void setSid(Integer sid) { this.sid = sid; }

public String getSname() { return sname; }

public void setSname(String sname) { this.sname = sname; }

public Integer getSage() { return sage; }

public void setSage(Integer sage) { this.sage = sage; }

**Example 2 – Field level**

import lombok.Getter;

import lombok.Setter;

class Student {

@Getter @Setter

private Integer sid;

private String sname; // no getter/setter

private Integer sage; // no getter/setter

}

👉 Compiler generates **only getter/setter for sid**.

# 📌 toString() in Java & Lombok

### 🔹 Purpose of toString()

* The toString() method is used to **represent an object’s data in a readable String format**.
* Very useful for:
  + **Logging** object values
  + **Debugging** (checking values at runtime)
  + Printing objects directly in System.out.println(object)

### 🔹 Default Behavior (Without Override)

If you don’t override toString(), the default implementation in java.lang.Object executes:

public String toString() {

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

👉 Example output:

in.orcas.Student@1a2b3c

This is **not human-readable**, hence we usually override toString().

### 🔹 Lombok @ToString

* Lombok’s @ToString **automatically generates** a toString() method for all fields.
* Example:

import lombok.ToString;

@ToString

class Student {

private Integer sid;

private String sname;

private Integer sage;

}

👉 Compiler generates:

public String toString() {

return "Student(sid=" + sid + ", sname=" + sname + ", sage=" + sage + ")";

}

### 🔹 Can we customize Lombok-generated methods?

* ❌ Direct customization of Lombok-generated code is **not possible**.
* Lombok only generates **default/common logic**.
* For **custom logic**, you must **manually override** the method in your class.

### 🔹 What happens if we override manually?

* If you explicitly define your own toString():
  + Lombok’s @ToString will **not generate** the method.
  + The compiler will use your custom implementation.
  + Lombok will show a **warning**: "Not generating toString() because method already exists."
* Same behavior applies to @Getter and @Setter → if you manually write getters/setters, Lombok won’t generate them.

✅ This ensures you always have control: Lombok generates defaults, but your code takes priority.

**@EqualsAndHashCode**

* It is a Lombok annotation.
* When you write it on a class, it **automatically creates** equals() and hashCode() methods for that class.
* You don’t need to write those methods yourself.
* It works at the **class level** (means you place it above the class definition).

**What is hashCode()?**

* Every object in Java has a number called **hash code**.
* This number is like the **ID** for the object.
* JVM (Java Virtual Machine) gives this number when the object is created.
* You can get this number by calling the method hashCode().
* System.out.println(obj.hashCode());
* **JVM’s equals() (from Object class)**  
  → It checks whether **two references point to the same object in memory**.  
  → If both references are pointing to the exact same memory location → returns true.  
  → Otherwise → returns false (even if the object values are same).
* **JVM’s hashCode() (from Object class)**  
  → It generates a number based on the **memory address (object’s identity hash)**.  
  → So two different objects → usually different hash codes.  
  → But it doesn’t guarantee uniqueness (though collisions are rare).
* equals() → **reference check**
* hashCode() → **memory address based number**

**Jvm Default Behaviour**

class Student {

int id;

String name;

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student();

s1.id = 101; s1.name = "Pavan";

Student s2 = new Student();

s2.id = 101; s2.name = "Pavan";

// JVM default equals()

// ✅ Checks only memory address, not values

System.out.println(s1.equals(s2)); // false ❌ because different objects in memory

// JVM default hashCode()

// ✅ Generated from memory address (object identity hash)

System.out.println(s1.hashCode()); // Example: 1234567

System.out.println(s2.hashCode()); // Example: 9876543

// ❌ Different numbers even if values are same

}

}

**🟢 Lombok’s @EqualsAndHashCode**

* When you use this annotation on a class, Lombok **generates equals() and hashCode() methods for you**.
* But the generated equals() is **NOT like JVM’s default**.
* Instead, Lombok compares the **fields (data)** of the object.

### 2️⃣ Lombok @EqualsAndHashCode Behavior

import lombok.EqualsAndHashCode;

@EqualsAndHashCode

class Student {

int id;

String name;

// Lombok generates equals() and hashCode() automatically

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student();

s1.id = 101; s1.name = "Pavan";

Student s2 = new Student();

s2.id = 101; s2.name = "Pavan";

// Lombok generated equals()

// ✅ Compares values of id and name

System.out.println(s1.equals(s2)); // true ✅ because values match

// Lombok generated hashCode()

// ✅ Generated based on values (id and name)

System.out.println(s1.hashCode()); // Example: 123456

System.out.println(s2.hashCode()); // Example: 123456

// ✅ Same number since values are same

}

}

**✅ Key Difference**

**equals():**

* Lombok **compares field values**, not just field names.
* In your example:
* StudentLombok l1 = new StudentLombok(1, "John");
* StudentLombok l2 = new StudentLombok(1, "John");
  + Lombok checks:
    - Is l1.id == l2.id? ✅ (1 == 1)
    - Is l1.name.equals(l2.name)? ✅ ("John" == "John")
  + Since all fields match → equals() returns **true**.

hashcode() :

Student s1 = new Student(1, "John");

Student s2 = new Student(1, "John");

System.out.println(s1.hashCode()); // e.g. 705927765

System.out.println(s2.hashCode()); // e.g. 705927765 (same as s1)

System.out.println(s1.equals(s2)); // true

**✅ Step 1: hashCode()**

* When you call s1.hashCode(), Lombok generates a **number** from s1’s field values.
* When you call s2.hashCode(), Lombok generates another number from s2’s field values.
* If both have the same values → the numbers will be **the same**.

👉 So **hashCode is only about producing a number from field values**.

**✅ Step 2: equals()**

* equals() **does not compare hashcodes**.
* Instead, it **directly compares field values** (like id and name).
* If all fields match → equals() returns **true**.

👉 Hashcode is **not used by equals()**

Neither equals() nor hashCode() care about **field names**.  
They only care about **field values** (the data stored inside the fields).