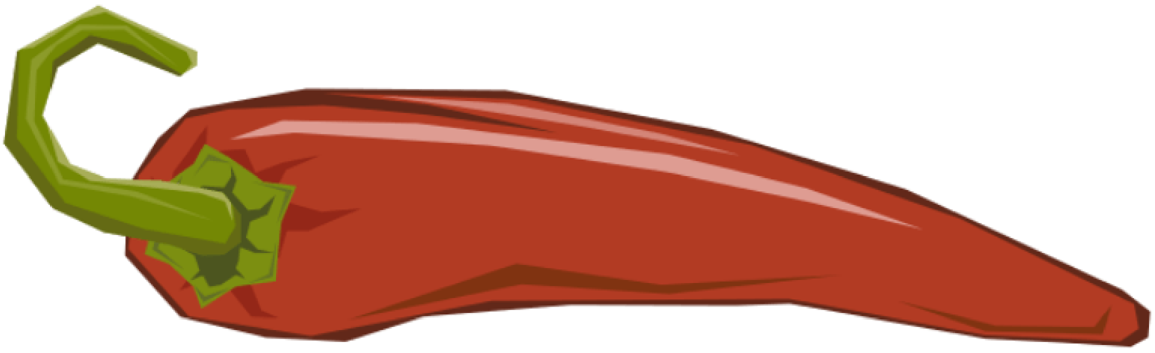


INDEX

Lombok API

Introduction

Java bean

- Java bean is a java class that is developed by following some standards
    a. Class must be public
    b. Recommended to implement Serializable
    c. Bean properties (member variables) should be private and non-static
    d. Every bean property should have 1 set of public setter & getter methods
    e. Should 0-param constructor directly or indirectly

3 types of Java Beans

- VO class (Value Object class) - To hold inputs or outputs
- DTO class (Data Transfer Object class) - To carry data from one layer to another layer or from one project to another project
- BO class/ Entity class/ Model class - To hold persistable/ persistent data

Well-Designed Java class should contain

a. overloaded constructors
b. toString()
c. equals(-) method
d. hashCode() method
e. getters & setters (optional)

Note:

- ✓ if equals() and hashCode() methods are not there in java class keeping objects of that java class becomes very problematic as the elements of collections.
- ✓ If toString() method is not there we can not print object data in single shot as Stringusing S.o.p(-) statement.
- ✓ Without setters & getters setting data and reading to/ from properties is very complex.

Before Lombok API

- We should manually make java classes as well-designed classes by adding the above said methods and we should manually increase or decrease getter and setter methods based on number of properties we are adding and removing.

With Lombok API

- All the above things will be taken care and will be generated internally.
- Lombok API also called as Project Lombok, generates the following boilerplate code of java class
  - Constructors
  - getters & setters
  - toString
  - equals
  - hashCode and etc.

- It is an open-source API.
- It must be configured with IDEs to make IDEs using Lombok API to generate the common boilerplate code.
- It supplies bunch of annotations for generating this common code
  - @Setter
  - @Getter
  - @AllArgsConstructor
  - @NoArgsConstructor
  - @RequiredArgsConstructor
  - @ToString
  - @EqualsAndHashCode
  - @Data (It is mix of multiple annotations)

    and etc.

Steps to configure Lombok with Eclipse [STS IDE]

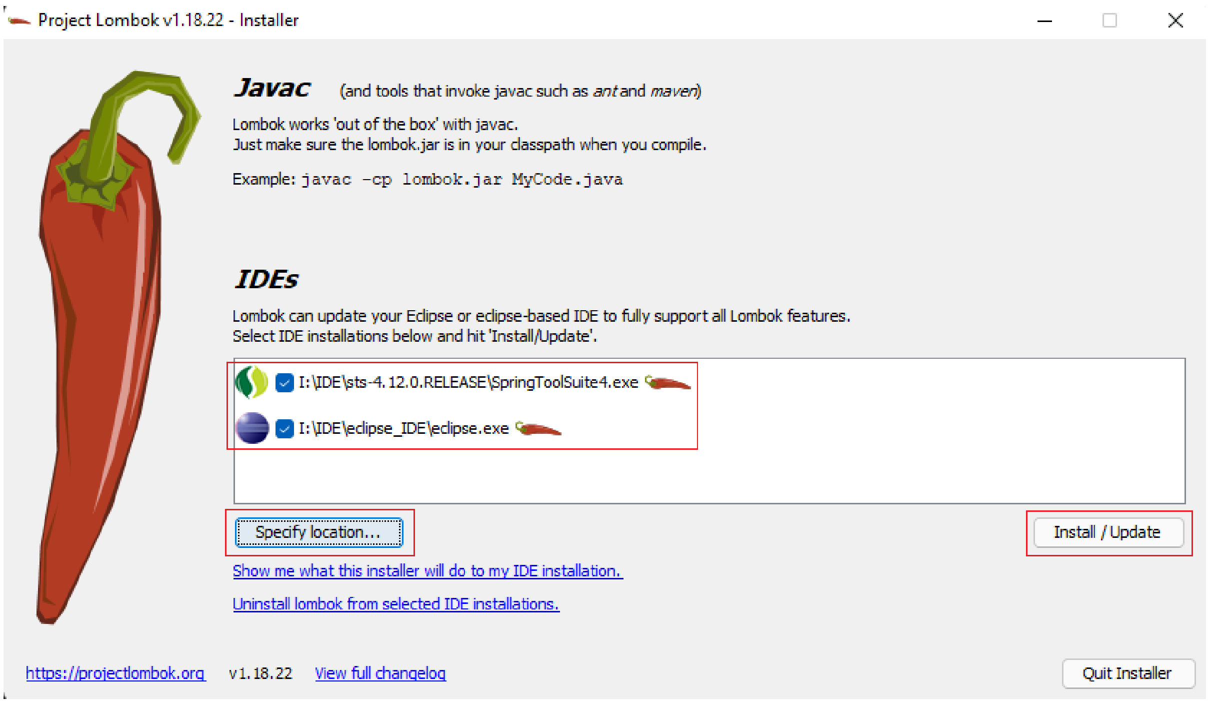
Step 1: Download Lombok [Link] or you can download from mvnrepository.com also [Link].

Step 2: Make sure Eclipse/ STS IDE installed.

Step 3: Create project in Eclipse or STS IDE by adding lombok-<ver>.jar file to the build path.

Note: If you will use the lombok annotation now then you will get error because lombok is not configured with your IDE.

Step 4: To configure launch Lombok API by double clicking on lombok-<ver>.jar then it will scan and detect your IDEs exe files other wise you can select your IDEs path by click on the Specify location… button then choose the IDEs and click on Install/ Update.


Project Lombok v1.18.22 - Installer

Javac (and tools that invoke javac such as *ant* and *maven*)

Lombok works 'out of the box' with javac.
Just make sure the lombok.jar is in your classpath when you compile.

Example: `javac -cp lombok.jar MyCode.java`

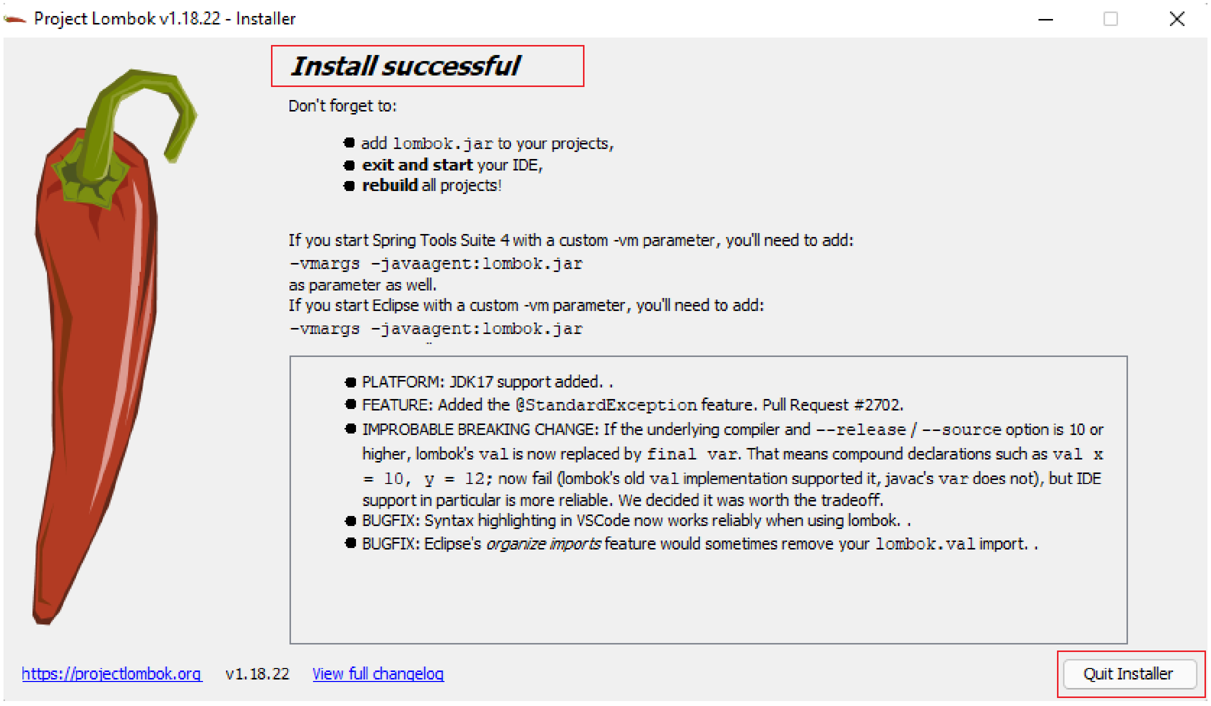
IDEs

Lombok can update your Eclipse or eclipse-based IDE to fully support all Lombok features.
Select IDE installations below and hit 'Install/Update'.

☑ I:\IDE\sts-4.12.0.RELEASE\SpringToolSuite4.exe
☑ I:\IDE\eclipse_IDE\eclipse.exe

Specify location...
Install / Update
Show me what this installer will do to my IDE installation.
Uninstall lombok from selected IDE installations.
https://projectlombok.org v1.18.22 View full changelog
Quit Installer

Step 5: Now you will get Install successful message then click on the Quit Installer button.


Project Lombok v1.18.22 - Installer

Install successful

Don't forget to:

- add `lombok.jar` to your projects,
- **exit and start** your IDE,
- **rebuild** all projects!

If you start Spring Tools Suite 4 with a custom -vm parameter, you'll need to add:
`-vmargs -javaagent:lombok.jar`
as parameter as well.
If you start Eclipse with a custom -vm parameter, you'll need to add:
`-vmargs -javaagent:lombok.jar`

- PLATFORM: JDK17 support added. .
- FEATURE: Added the `@StandardException` feature. Pull Request #2702.
- IMPROBABLE BREAKING CHANGE: If the underlying compiler and `--release` / `--source` option is 10 or higher, lombok's `val` is now replaced by `final var`. That means compound declarations such as `val x = 10, y = 12;` now fail (lombok's old `val` implementation supported it, javac's `var` does not), but IDE support in particular is more reliable. We decided it was worth the tradeoff.
- BUGFIX: Syntax highlighting in VSCode now works reliably when using lombok. .
- BUGFIX: Eclipse's *organize imports* feature would sometimes remove your `lombok.val` import. .

Step 6: Restart Eclipse /STS IDE.

Step 7: Add one Java bean class to project of Eclipse/ STS IDE by having lombok

API annotation and observe whether code is generated or not?

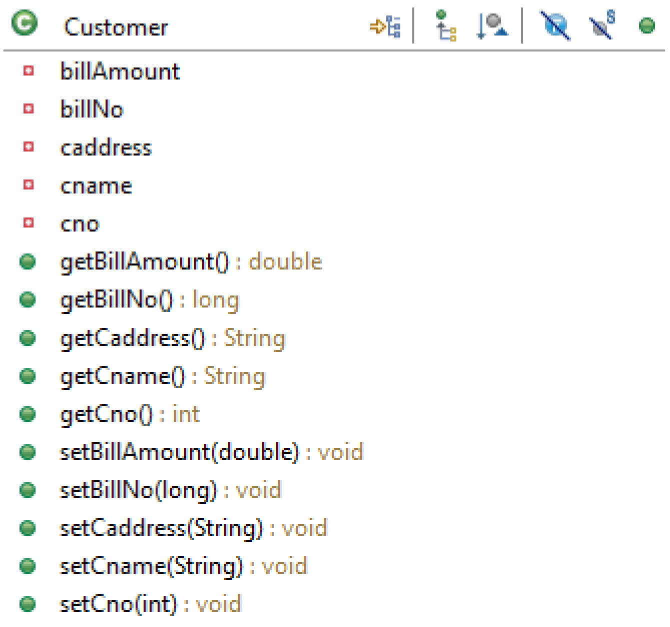
Customer.java

```
package com.sahu.model;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class Customer {
	private int cno;
	private String cname;
	private String caddress;
	private double billAmount;
	private long billNo;
}
```

If you got to Type hierarchy (FN + F4/ F4) you can see the methods



```
Customer
billAmount
billNo
caddress
cname
cno
getBillAmount() : double
getBillNo() : long
getCaddress() : String
getCname() : String
getCno() : int
setBillAmount(double) : void
setBillNo(long) : void
setCaddress(String) : void
setCname(String) : void
setCno(int) : void
```

Note:

- ✓ Lombok API annotations makes java compiler to generate certain code dynamically in the .class file.
- ✓ Java compiler is having ability to add code dynamically to .class file though instructions are not there in .java file like default constructor

generation, making java.lang.Object as default super class and etc.

- ✓ Lombok API annotations Retention level is Source i.e., these annotations will not be recorded to .class files but because of lombok annotations instructions the code generated by javac compiler like setters, getters, toString and etc. will be recorded into .class file.

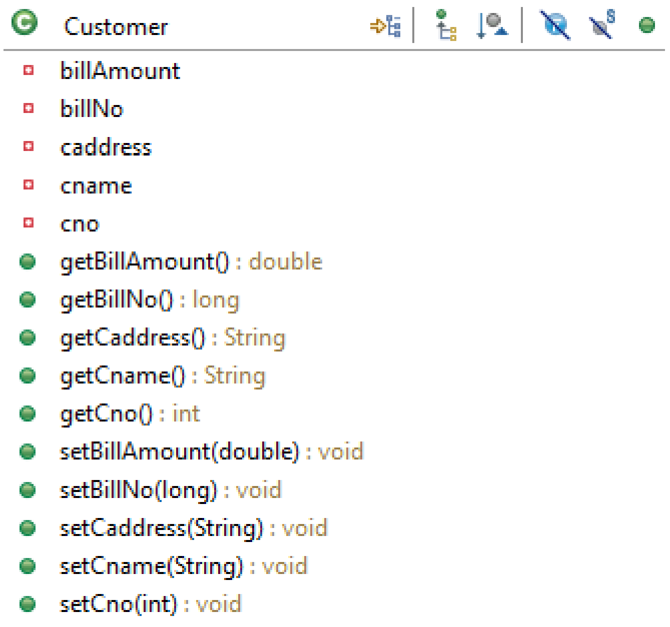
Annotations

@Getter, @Setter

- If these are applied at class level generate setters and getters for all fields/ properties.
- If these are applied at fields level generate setters and getters for specific fields/ properties.

Customer.java

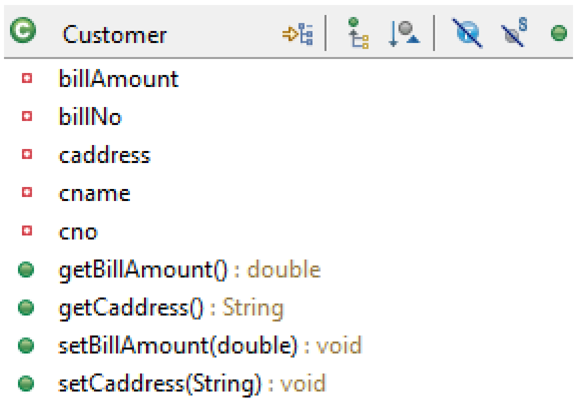
```
@Setter
@Getter
public class Customer {
        private int cno;
        private String cname;
        private String caddress;
        private double billAmount;
        private long billNo;
}
```



```
Customer
  billAmount
  billNo
  caddress
  cname
  cno
  getBillAmount() : double
  getBillNo() : long
  getCaddress() : String
  getCname() : String
  getCno() : int
  setBillAmount(double) : void
  setBillNo(long) : void
  setCaddress(String) : void
  setCname(String) : void
  setCno(int) : void
```

Customer.java

```
public class Customer {
        private int cno;
        private String cname;
        @Setter
        @Getter
        private String caddress;
        @Setter
        @Getter
        private double billAmount;
        private long billNo;
}
```



```
Customer
billAmount
billNo
caddress
cname
cno
getBillAmount() : double
getCaddress() : String
setBillAmount(double) : void
setCaddress(String) : void
```

@ToString

- Makes the javac compiler to generate/ override toString() having logic to display object data. It is applicable only top of class.

Q. What is the use toString()?

Ans.

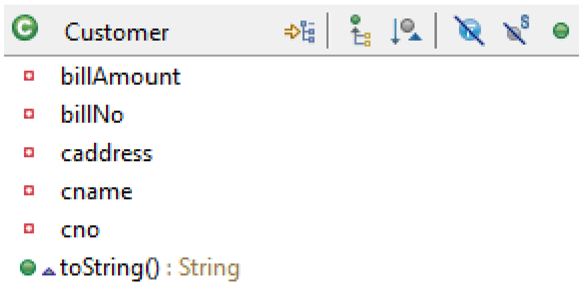
- ➢ It is useful to display object data in the form of single String format.
- ➢ If we do not override this method in our class then java.lang.Object class toString() method executes and this method gives <fully qualified class name>@<hexa-decimal notation of hashCode>

```
public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

If we call S.o.p() with any reference variable on that toString() will be called automatically.

Customer.java

```
@ToString
public class Customer {
        private int cno;
        private String cname;
        private String caddress;
        private double billAmount;
        private long billNo;
}
```



```
Customer
  billAmount
  billNo
  caddress
  cname
  cno
  toString() : String
```

Q. Can we customize the logics generated by Lombok API?
Ans. Not possible

Q. What happens if we place toString() explicitly in our class along with Lombok API @ToString?
Ans. @ToString of Lombok API will not give instruction to javac compiler to generate toString() because same method is already available in .java file. (warning will come for @ToString annotation like "Not generating toString(): A method with that name already exists").

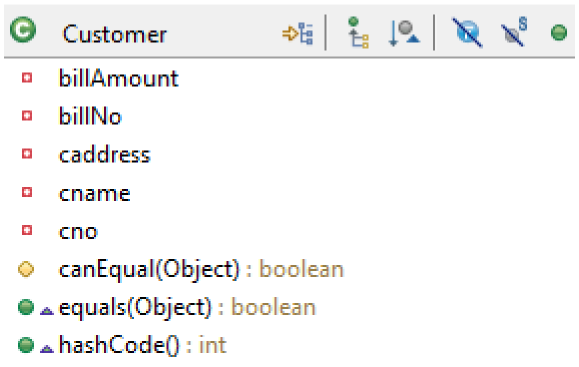
Note: Same is applicable for @Getter and @Setter annotations.

@EqualsAndHashCode

- Generates equals() method and hashCode() method in the .class file dynamically by giving instruction to java compiler (javac).
- Applicable at class level (type).

Customer.java

```
@EqualsAndHashCode
public class Customer {
        private int cno;
        private String cname;
        private String caddress;
        private double billAmount;
        private long billNo;
}
```


Customer
billAmount
billNo
caddress
cname
cno
canEqual(Object) : boolean
equals(Object) : boolean
hashCode() : int

Q. What is hashCode()?

Ans. HashCode is unique identity number given by JVM by every object and we can get it by calling hashCode() method.

E.g.,

```
System.out.println(c1.hashCode()+" "+c2.hashCode());
```

Q. Can two objects have same hashCode()?

Ans.

- If hashCode() method java.lang.Object class is called then not possible because it generates hashCode based hashing algorithm as unique number.
- If we override hashCode() method our code generally it generates the hashCode based on state of the object of two objects are having same state then we get same hashCode for both objects.
- E.g.,

```
Customer c1 = new Customer(101,"raja","hyd",677.88);
Customer c2 = new Customer(101,"raja","hyd",677.88);
System.out.println(c1.hashCode()+" "+c2.hashCode()); //prints
                                                        same hashCode
```

Q. Can one object have two hashCode?

Ans. If you override hashCode() method then based on state of the object one hashCode will be generated but internally JVM maintains another hashCode based on hashing algorithm.

E.g.,

```
Customer c1 = new Customer(101,"raja","hyd",677.88);
Customer c2 = new Customer(101,"raja","hyd",677.88);
System.out.println(c1.hashCode()+" "+c2.hashCode()); //our hashCode

System.out.println(System.identityHashCode(c1)); //JVM hashCode
System.out.println(System.identityHashCode(c2));
```

Q. What is the use of equals(-) method?

Ans.

- It is given to compare that state of two objects.
- It internally uses hashCode() support also.
- If equals() is overridden in our class then it will compare the state of the two objects otherwise Object class equals() method executes which always compares references by internally using operator equals() method of java.lang.Object class

```
public boolean equals(Object obj) {
	return (this == obj);
}
```

Q. what is the difference between == and equals() method.

Ans.

- Both will compare references if equals() method is not overridden the our class.
- If overridden equals(-) method compares state of objects and == operator checks the references.

```
Customer c1 = new Customer(101,"raja","hyd",677.88);
Customer c2 = new Customer(101,"raja","hyd",677.88);
System.out.println(c1.equals(c2));
System.out.println(c1==c2);
```

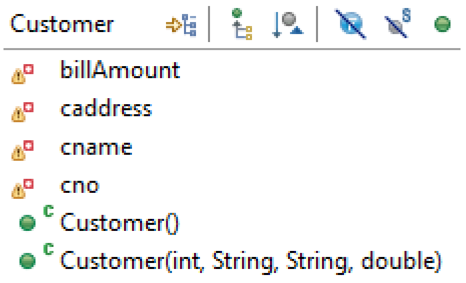
Answers are #1. false & #2. False if equals(-) method is not overridden in Customer class otherwise answers are #1. true & #2. false.

@NoArgsConstructor, @AllArgsConstructor

- @NoArgsConstructor Generates 0-param constructor.
- @AllArgsConstructor Generates parameterized having all properties/ fields as params, if class is not having any properties/ fields then it generates 0-param constructor.
- Both are applicable at class level (type).

Customer.java

```
@AllArgsConstructor
@NoArgsConstructor
public class Customer {
        private int cno;
        private String cname;
        private String caddress;
        private double billAmount;
}
```



```
Customer
  billAmount
  caddress
  cname
  cno
  Customer()
  Customer(int, String, String, double)
```

Note:

- ✓ Only compiler generated 0-param constructor is called default constructor.
- ✓ If we place 0-param constructor explicitly or if Lombok generates the 0-param constructor then that should not be called default constructor

Q. Can we perform constructor overloading and constructor overriding?
Ans. Constructor overloading is possible, but constructor overriding is not possible because in sub class constructor names changes to sub class name.

F&Q:

```
public class Test {
        public Test (int x) {this (x, y)}
        public Test (int x, int y) {this (x)}
        public Test () {this (x)}
}
```

Gives error because constructor chaining is leading infinite loop. To break it write in any constructor call super().

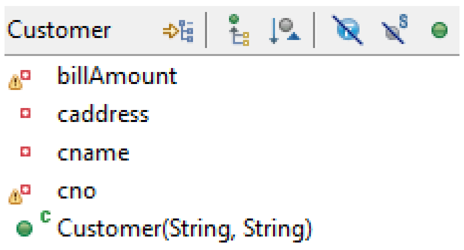
@RequiredArgsCosntructor

- Allows to generates parameterized constrictor involving our choice number of properties/ fields.

- The properties that you want to involve should be annotated with @NonNull annotation.
- If no properties are annotated with @NonNull then it will give 0-param constructor.

Customer.java

```
@RequiredArgsConstructor
public class Customer {
        private int cno;
        @NonNull
        private String cname;
        @NonNull
        private String caddress;
        private double billAmount;
}
```

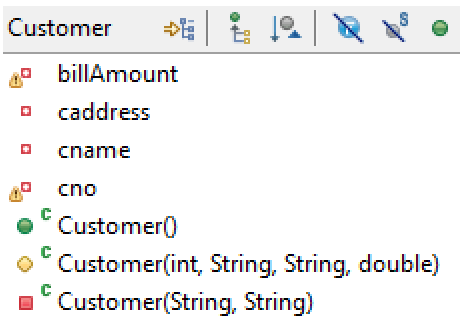


```
Customer
    billAmount
    caddress
    cname
    cno
    Customer(String, String)
```

- We can take constructor as private, protected and public to get them through Lombok API we can use "access" attribute of @XxxArgsCosntrctor annotations as shown below

Customer.java

```
@RequiredArgsConstructor(access = AccessLevel.PRIVATE)
@AllArgsConstructor(access = AccessLevel.PROTECTED)
@NoArgsConstructor(access = AccessLevel.PUBLIC)
public class Customer {
        private int cno;
        @NonNull
        private String cname;
        @NonNull
        private String caddress;
        private double billAmount;
}
```



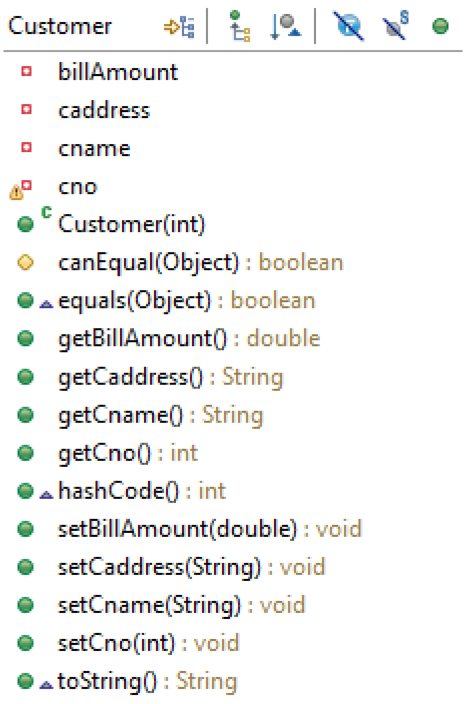
```
Customer
billAmount
caddress
cname
cno
Customer()
Customer(int, String, String, double)
Customer(String, String)
```

@Data

- It is combination of @Getter + @Setter + @EqualsAndHashCode + @ToString + @RequiredArgsConstructor

Customer.java

```java
@Data
public class Customer {
	@NonNull
	private int cno;
	private String cname;
	private String caddress;
	private double billAmount;
}
```



```
Customer
billAmount
caddress
cname
cno
Customer(int)
canEqual(Object) : boolean
equals(Object) : boolean
getBillAmount() : double
getCaddress() : String
getCname() : String
getCno() : int
hashCode() : int
setBillAmount(double) : void
setCaddress(String) : void
setCname(String) : void
setCno(int) : void
toString() : String
```

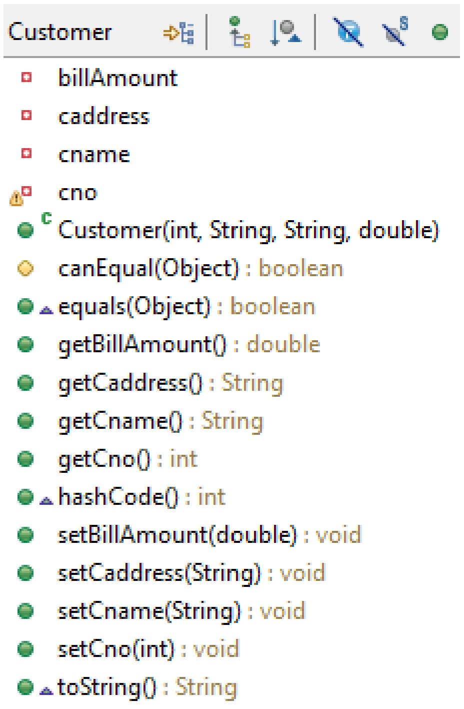
Note: Using Lombok API

- ✓ We cannot generate constrictor with var args.
- ✓ We cannot generate multiple our choice parameterized constructors at time like with 1 param, with 2 params, with 3 params at a time.

- @RequiredArgsConstructor of @Data works only when @AllArgsConstructor, @NoArgsConstructor is placed on the top of the class if you still need the effect of @RequiredArgsConstructor then place it explicitly.

Customer.java

```
@Data
@AllArgsConstructor
public class Customer {
        @NonNull
        private int cno;
        private String cname;
        private String caddress;
        private double billAmount;
}
```



```
Customer
billAmount
caddress
cname
cno
Customer(int, String, String, double)
canEqual(Object) : boolean
equals(Object) : boolean
getBillAmount() : double
getCaddress() : String
getCname() : String
getCno() : int
hashCode() : int
setBillAmount(double) : void
setCaddress(String) : void
setCname(String) : void
setCno(int) : void
toString() : String
```

Category wise Annotations

1. Getter and Setter Annotations

- **@Getter**
  - **Description:** Generates getter methods for all non-static fields.
  - **Use Case:** When you need getter methods but don't want to write them manually.

- **@Setter**
  - **Description:** Generates setter methods for all non-static fields.
  - **Use Case:** When you need setter methods but don't want to write them manually.

- **@Accessors**
  - **Description:** Allows customization of getter and setter naming conventions.
  - **Use Case:** Used when you want to modify the default Lombok behavior for fluent API (chaining methods) or prefix handling.

2. Constructors

- **@NoArgsConstructor**
  - **Description:** Generates a no-argument constructor.
  - **Use Case:** Useful when frameworks like Hibernate require a default constructor.

- **@AllArgsConstructor**
  - **Description:** Generates a constructor with all fields as parameters.
  - **Use Case:** Useful when you need a constructor to initialize all fields.

- **@RequiredArgsConstructor**
  - **Description:** Generates a constructor with only final and @NonNull fields.
  - **Use Case:** Used when only some fields must be initialized at object creation.

3. Data Handling

- **@Data**
  - **Description:** A shortcut that combines @Getter, @Setter, @ToString, @EqualsAndHashCode, and

@RequiredArgsConstructor.
    - **Use Case:** Ideal for simple DTOs (Data Transfer Objects).

- **@Value**
    - **Description:** Similar to @Data, but makes all fields private final and removes setters.
    - **Use Case:** For immutable objects.

4. Equals, HashCode, and ToString

- **@EqualsAndHashCode**
    - **Description:** Generates equals() and hashCode() methods.
    - **Use Case:** Needed when objects should be compared by values rather than references.

- **@ToString**
    - **Description:** Generates a toString() method.
    - **Use Case:** Used to create readable string representations of objects.

5. Logging Annotations

- **@Slf4j**
    - **Description:** Creates a private static final org.slf4j.Logger log.
    - **Use Case:** When using SLF4J for logging.

- **Other logging alternatives:**
    - @Log → Uses java.util.logging.Logger
    - @CommonsLog → Uses Apache Commons Logging
    - @Log4j → Uses Log4j
    - @Log4j2 → Uses Log4j2
    - @XSlf4j → Uses SLF4J with Log4j

6. Synchronized and Exception Handling

- **@Synchronized**
    - **Description:** Makes a method thread-safe.
    - **Use Case:** When synchronization is required.

- **@SneakyThrows**
    - **Description:** Allows throwing checked exceptions without try-catch.
    - **Use Case:** Useful when you want to avoid explicit exception

handling.

- **@StandardException**
  - **Description:** Generates a custom exception class with constructors for various use cases.
  - **Use Case:** Used when you need a custom exception without manually writing constructors.

7. Builder Pattern

- **@Builder**
  - **Description:** Implements the Builder pattern.
  - **Use Case:** Used when constructing objects step by step.

- **@SuperBuilder**
  - **Description:** Extends @Builder to support inheritance.
  - **Use Case:** Needed when using @Builder in subclasses.

- **@Singular**
  - **Description:** Used with @Builder to handle collections properly.
  - **Use Case:** Needed when working with lists or sets in builder pattern.

8. Utility and Miscellaneous Annotations

- **@UtilityClass**
  - **Description:** Converts a class into a singleton utility class with: private constructor and static methods only
  - **Use Case:** Used when creating helper classes (e.g., MathUtils, StringUtils).

- **@NonNull**
  - **Description:** Generates a NullPointerException if a parameter is null.
  - **Use Case:** Used to enforce non-null parameters.

- **@Cleanup**
  - **Description:** Ensures close() is called on a resource.
  - **Use Case:** Used to automatically close resources.

- **@With**
  - **Description:** Generates a copy method for immutable objects,

allowing modification of a specific field while keeping other fields unchanged.
    - **Use Case:** When working with immutable objects, where you need to create modified copies instead of modifying the original instance.

- **@Experimental**
    - **Description:** Marks a feature as experimental, indicating that it may change or be removed in future Lombok versions.
    - **Use Case:** Used when trying out unstable Lombok features that may not be fully supported yet.

- **@FieldDefaults**
    - **Description:** Sets default field modifiers (private, final, etc.) at the class level.
    - **Use Case:** Used when you want to enforce consistent field access control throughout the class.

- **@Delegate**
    - **Description:** Enables delegation of method calls to another class.
    - **Use Case:** Used when you want to forward method calls to another class without writing wrapper methods manually.

--- The END ---