39-Lombok

Author: Vincent Lau

Learning Objectives

- Understand the purpose of using Lombok
- As a beginner, understand the actual codes generated by Lombok.
- Able to code the conventional approach for each annotation.

Introduction

`Lombok` is a popular Java library that helps **reduce boilerplate code by automatically generating common methods during compile time, such as getters, setters, constructors, etc**. It allows you to annotate your classes and fields to generate these methods, which can lead to more concise and readable code.

Background

Before this chapter, we assume that you have full understanding of constructors, getters, setters, builders, toString, hashCode, equals in OOP.

Did you ever think that all the above methods in a class are boilerplate code?

In which case should there be a way to generate the code automatically?

And most importantly, developers should be able to control the auto-generation one by one, because it implies different meanings when we include different combinations of those methods.

Now, Lombok tries to solve this problem and we will investigate this library in this chapter.

Notes: As a developer, you should be able to write all the methods manually, even if you like the way that lombok generates for you.

Conventional Approach

```
1 import java.util.Objects;
2
3 public class Person {
4   private String firstName;
5   private String lastName;
6   private int age;
```

```
7
8   // Constructors
9   public Person() {}
10
11  public Person(String firstName, String lastName, int age) {
12    this.firstName = firstName;
13    this.lastName = lastName;
14    this.age = age;
15  }
16
17  // Private Constructor, for builder class use
18  private Person(PersonBuilder builder) {
19    this.firstName = builder.firstName;
20    this.lastName = builder.lastName;
21    this.age = builder.age;
22  }
23
24  // Getters and Setters
25  public String getFirstName() {
26    return firstName;
27  }
28
29  public void setFirstName(String firstName) {
30    this.firstName = firstName;
31  }
32
33  public String getLastName() {
34    return lastName;
35  }
36
37  public void setLastName(String lastName) {
38    this.lastName = lastName;
39  }
40
41  public int getAge() {
42    return age;
43  }
44
45  public void setAge(int age) {
46    this.age = age;
47  }
48
49  // toString
50  @Override
51  public String toString() {
52    return "Person [firstName=" + this.firstName //
53        + ", lastName=" + this.lastName //
```

```
54          + ", age=" + this.age + "]";
55    }
56
57    // hashCode
58    @Override
59    public int hashCode() {
60      return Objects.hash(this.firstName, this.lastName, this.age);
61    }
62
63    // equals
64    @Override
65    public boolean equals(Object obj) {
66      if (this == obj)
67        return true;
68      if (!(obj instanceof Person))
69        return false;
70      Person person = (Person) obj;
71      return Objects.equals(this.firstName, person.firstName)
72          && Objects.equals(this.lastName, person.lastName)
73          && Objects.equals(this.age, person.age);
74    }
75
76    // Builder static inner class
77    public static class PersonBuilder {
78      private String firstName;
79      private String lastName;
80      private int age;
81
82      public PersonBuilder firstName(String firstName) {
83        this.firstName = firstName;
84        return this;
85      }
86
87      public PersonBuilder lastName(String lastName) {
88        this.lastName = lastName;
89        return this;
90      }
91
92      public PersonBuilder age(int age) {
93        this.age = age;
94        return this;
95      }
96
97      public Person build() {
98        return new Person(this);
99      }
100   }
```

```
}

```

Add Lombok to Your Project

To use Lombok in your Java project, you need to add the Lombok dependency to your build configuration. If you're using a build tool like Maven or Gradle, you can add the following dependency:

Maven

```xml
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.30</version> <!-- Use the latest version -->
    <scope>provided</scope>
</dependency>
```

Gradle

```
dependencies {
    compileOnly 'org.projectlombok:lombok:1.18.22' // Use the latest version
    annotationProcessor 'org.projectlombok:lombok:1.18.22'
}
```

Using Lombok Annotations

Once you've added Lombok to your project, you can start using its annotations.

Here are some common Lombok annotations and their purposes:

1. Getters and Setters

Reference: https://projectlombok.org/features/GetterSetter

When you use Lombok's `@Getter` and `@Setter` annotations, Lombok generates the getter and setter methods automatically for you. The Lombok approach eliminates the need to write the getter and setter methods explicitly.

Conventional Approach

```java
public class Person {
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Lombok Approach

```java
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class Person {
    private String firstName;
    private String lastName;
}
```

2. No-Args & All-Args Constructors

Reference: https://projectlombok.org/features/constructor

With Lombok's `@NoArgsConstructor` and `@AllArgsConstructor` annotations, you can generate constructors automatically. Here's the comparison:

- `@NoArgsConstructor` : Generates a no-args constructor that initializes all fields to their default values. This constructor is useful when you need to create instances without providing initial values.

- `@AllArgsConstructor` : Generates an all-args constructor that accepts parameters for all fields. This constructor is useful when you want to create instances with specified values for all fields.

Conventional Approach

```
1  public class Person {
2      private String firstName;
3      private String lastName;
4      // NoArgsConstructor
5      public Person() {
6      }
7      // AllArgsConstructor
8      public Person(String firstName, String lastName) {
9          this.firstName = firstName;
10         this.lastName = lastName;
11     }
12 }
```

Lombok Approach

```
1  import lombok.NoArgsConstructor;
2  import lombok.AllArgsConstructor;
3
4  @NoArgsConstructor
5  @AllArgsConstructor
6  public class Person {
7      private String firstName;
8      private String lastName;
9  }
```

3. Required-Args Constructor

Reference: https://projectlombok.org/features/constructor

The `@RequiredArgsConstructor` annotation generates a constructor that initializes final OR non-null fields. This means that any fields marked as `final` or annotated with `@NonNull` (a Lombok annotation) will be automatically included in the constructor, and the constructor will expect values for these fields to be provided when creating an object.

Conventional Approach

```java
public class Person {
    private final String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName) {
        if (lastName == null)
            throw new NullPointerException("lastName");
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

Lombok Approach

```java
import lombok.RequiredArgsConstructor;
import lombok.NonNull;

@RequiredArgsConstructor
public class Person {
    private final String firstName;
    @NonNull
    private String lastName;
    private int age; // Not final and not marked as @NonNull
}
```

In the above example, the `@RequiredArgsConstructor` annotation generates a constructor that expects `firstName` and `lastName` to be provided during object creation because they are marked as `final` or annotated with `@NonNull`. The `age` field is not included in the constructor because it is not `final` and not marked as `@NonNull`.

For those fields marked with `@NonNull`, an explicit null check is also generated. The constructor will throw a `NullPointerException` if any of the parameters intended for the fields marked with `@NonNull` contain `null`.

4. Equals & HashCode

Reference: https://projectlombok.org/features/EqualsAndHashCode

The `@EqualsAndHashCode` annotation in Lombok generates `equals()` and `hashCode()` methods based on the fields you specify. Let's compare how to implement `equals()` and `hashCode()` methods using both the conventional approach and Lombok's `@EqualsAndHashCode` annotation.

Conventional Approach

```java
import java.util.Objects;

public class Person {
    private String firstName;
    private String lastName;

    // Constructors, getters, setters

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (!(o instanceof Person))
            return false;
        Person person = (Person) o;
        return Objects.equals(firstName, person.firstName) &&
               Objects.equals(lastName, person.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }
}
```

Lombok Approach

```java
import lombok.EqualsAndHashCode;

@EqualsAndHashCode
public class Person {
    private String firstName;
    private String lastName;

    // Constructors, getters, setters
}
```

With Lombok's `@EqualsAndHashCode` annotation, you can automatically generate `equals()` and `hashCode()` methods based on the fields you specify. Here's the comparison:

- Conventional Approach: In the conventional approach, you need to manually implement `equals()` and `hashCode()` methods using the `Objects.equals()` and `Objects.hash()` methods. This requires careful implementation to ensure consistency and correctness.
- Lombok Approach: By using the `@EqualsAndHashCode` annotation, Lombok automatically generates `equals()` and `hashCode()` methods for you based on the fields you specify. This eliminates the need for manual implementation and ensures consistent and correct equality and hash code behavior.

5. Equals & HashCode (Super Class)

Let's explore a complicated one.

Conventional Approach

```java
import java.util.Objects;

public class Person {
  private String firstName;
  private String lastName;

  // Constructors, getters, setters ...

  public Person(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o)
      return true;
    if (!(o instanceof Person))
      return false;
    Person person = (Person) o;
    return Objects.equals(this.firstName, person.firstName) &&
        Objects.equals(this.lastName, person.lastName);
  }

  @Override
  public int hashCode() {
    return Objects.hash(this.firstName, this.lastName);
  }
}
```

```java

public class Employee extends Person {
  private int employeeId;

  public Employee(int employeeId, String firstName, String lastName) {
    super(firstName, lastName);
    this.employeeId = employeeId;
  }

  // getters, setters ....

  @Override
  public boolean equals(Object o) {
    if (this == o)
      return true;
    if (!(o instanceof Employee))
      return false;
    if (!super.equals(o))
      return false; // Check superclass equality
    Employee employee = (Employee) o;
    return employeeId == employee.employeeId;
  }

  @Override
  public int hashCode() {
    return Objects.hash(super.hashCode(), employeeId);
  }
}
```

Lombok Approach

```java
@EqualsAndHashCode(callSuper = true)
public class Employee extends Person {
    private int employeeId;

    // Constructors, getters, setters
}

@EqualsAndHashCode
public class Person {
    protected String firstName;
    protected String lastName;

    // Constructors, getters, setters
}
```

In both approaches, you need to consider the superclass fields when overriding `equals()` and `hashCode()` in subclasses. In the conventional approach, you explicitly call the superclass methods and include superclass fields in the hash code calculation. In Lombok, you achieve the same by setting the `callSuper` attribute of `@EqualsAndHashCode` to `true`.

When using Lombok's `@EqualsAndHashCode`, ensure you understand how the `callSuper` attribute works and whether it's appropriate to include superclass fields and behavior in the generated methods for your specific use case.

6. toString (Super Class)

Reference: https://projectlombok.org/features/ToString

The `@ToString` annotation in Lombok generates a `toString()` method that includes a human-readable representation of the object's fields. Let's compare how to implement the `toString()` method using both the conventional approach and Lombok's `@ToString` annotation.

Conventional Approach

```
1  public class Person {
2      private String firstName;
3      private String lastName;
4
5      // Constructors, getters, setters
6      @Override
7      public String toString() {
8        return "Person(firstName=" + this.firstName + ", lastName=" +
   this.lastName + ")";
9      }
10 }
11
12 public class Employee extends Person {
13     private int employeeId;
14
15     // Constructors, getters, setters
16     @Override
17     public String toString() {
18       return "Employee(super=" + super.toString() + ", employeeId=" +
   this.employeeId + ")";
19     }
20 }
21
22
```

Lombok Approach

```java
import lombok.ToString;

@ToString
public class Person {
    private String firstName;
    private String lastName;

    // Constructors, getters, setters
}

@ToString(callSuper = true)
public class Employee extends Person {
    private int employeeId;

    // Constructors, getters, setters
}
```

With Lombok's `@ToString` annotation, you can automatically generate a `toString()` method that includes a string representation of the object's fields. Here's the comparison:

- Conventional Approach: In the conventional approach, you need to manually implement the `toString()` method, concatenating the field names and values into a formatted string.
- Lombok Approach: By using the `@ToString` annotation, Lombok automatically generates a `toString()` method for you. It includes a formatted string representation of the object's fields, making debugging and logging easier.

7. Builder Pattern

Reference: https://projectlombok.org/features/Builder

The `@Builder` annotation in Lombok generates a builder pattern for constructing objects with a fluent API. Let's compare how to implement the builder pattern using both the conventional approach and Lombok's `@Builder` annotation.

Conventional Approach

```java
public class Person {
    private String firstName;
    private String lastName;
```

```
    private int age;

    private Person(Builder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.age = builder.age;
    }

    public static class Builder {
        private String firstName;
        private String lastName;
        private int age;

        public Builder firstName(String firstName) {
            this.firstName = firstName;
            return this;
        }

        public Builder lastName(String lastName) {
            this.lastName = lastName;
            return this;
        }

        public Builder age(int age) {
            this.age = age;
            return this;
        }

        public Person build() {
            return new Person(this);
        }
    }

    // Getters and other methods
}
```

Lombok Approach

```
import lombok.Builder;

@Builder
public class Person {
    private String firstName;
    private String lastName;
    private int age;
```

```
8
9       // Getters and other methods
10  }
```

With Lombok's `@Builder` annotation, you can automatically generate a builder pattern for constructing objects. Here's the comparison:

- Conventional Approach: In the conventional approach, you need to manually define a nested `Builder` class with methods for setting each field, and you also need to define a constructor that takes a `Builder` instance as a parameter.
- Lombok Approach: By using the `@Builder` annotation, Lombok automatically generates the builder pattern for you. It generates a builder class with fluent methods for setting fields and a constructor that accepts the builder instance.

8. Data

Reference: https://projectlombok.org/features/Data

The `@Data` annotation in Lombok is a convenient way to automatically generate `getter` and `setter` methods, `equals()`, `hashCode()`, `toString()` methods and `RequiredArgsConstructor` for your class fields. Let's compare how to achieve this using both the conventional approach and Lombok's `@Data` annotation.

Conventional Approach

```
1  import java.util.Objects;
2
3  public class Person {
4      private String firstName;
5      private String lastName;
6      private int age;
7
8      public Person() {
9
10     }
11
12     public String getFirstName() {
13         return firstName;
14     }
15
16     public void setFirstName(String firstName) {
17         this.firstName = firstName;
18     }
19
```

```
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (!(o instanceof Person))
            return false;
        Person person = (Person) o;
        return this.age == person.age &&
               Objects.equals(this.firstName, person.firstName) &&
               Objects.equals(this.lastName, person.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.firstName, this.lastName, this.age);
    }

    @Override
    public String toString() {
      return "Person(firstName=" + this.firstName
          + ", lastName=" + this.lastName
          + ", age=" + this.age ")";
    }
}

```

Lombok Approach

```
import lombok.Data;
```

```
2
3 @Data // @Getter @Setter @RequiredArgsConstructor @ToString @EqualsAndHashCode
4 public class Person {
5     private String firstName;
6     private String lastName;
7     private int age;
8 }
```

With Lombok's `@Data` annotation, you can automatically generate getter and setter methods, `equals()`, `hashCode()`, and `toString()` methods for your class fields. Here's the comparison:

- Conventional Approach: In the conventional approach, you need to manually implement getter and setter methods, as well as `equals()`, `hashCode()`, and `toString()` methods. This can lead to a lot of boilerplate code.
- Lombok Approach: By using the `@Data` annotation, Lombok automatically generates getter and setter methods for fields, as well as the `equals()`, `hashCode()`, and `toString()` methods. This significantly reduces the amount of code you need to write and maintain.