Making our daily job easy

by reducing boilerplate code using Project Lombok

Doychin Bondzhev

Who am I

- I work in my own company - **dSoft-Bulgaria**
- Member of Bulgarian Java User Group – **just like all other Java Developers in Bulgaria**
- Organizer of Java Beer events in **Plovdiv** – http://www.java.beer
- member of JSR-377 expert group
  - Desktop|Embedded Application API

BULGARIAN Java User Group

J2D

How to make our job easier

- Problems
  - We need to write lots of code.
  - We also need to read/review other people's code
  - We have lots of code that does not add value. It is there just because of the requirements of the Java language

How to make our job easier

- Solutions
  - Use IDE to generate that code
  - Use another language that provides the necessary features

But we are Java developers

We want to use our favorite language

BULGARIAN Java User Group

J2D

How to do it in Java?

Use Project Lombok

https://projectlombok.org/

What is Project Lombok?

It is a library that gives us a set of annotations to help us get rid of all boilerplate code

How to use Lombok

- Download lombok.jar from https://projectlombok.org/ and add it to your build class path.
- Use it with maven

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.16.18</version>
  <scope>provided</scope>
</dependency>
```

- Gradle

```
provided group: 'org.projectlombok', name: 'lombok', version: '1.16.18'
```

- Ant

```
<javac srcdir="src" destdir="build" source="1.8">
        <classpath location="lib/lombok.jar" />
</javac>
```

@Getter and @Setter

- Can be used on **class** or **field**
- When used on **class** Lombok generates get/set methods to all non-static fields
- When used on **field** Lombok generates get/set methods for that field
- Has one parameter that tells what access level to use for get/set methods
- By default access level is public

Next are equals(), hashCode() and toString()

- **@EqualsAndHashCode** tells Lombok to generate for us equals() and hashCode()
- **@ToString** tells Lombok to generate toString()
- **callSuper** – This parameter indicates that call to the corresponding super method must be included during the evaluation of the corresponding method
- **of** – comma separated list of field names to include in the evaluation of **hashCode()**, **equals()** and **toString()**
- **exclude** – comma separated list of field names to exclude from the evaluation of **hashCode()**, **equals()** and **toString()**
- Last two can't be used at the same time

Constructors

- **@AllArgsConsutructor** – generates constructor that will have all non-static fields as parameters
- **@NoArgsConstructor** – generates default empty constructor
- **@RequiredArgsConstructor** – generates constructor that will have as parameters all final and **@NonNull** fields

- Common parameters
  - **access** – parameter to set the access level for the generated constructor. By default it is public
  - **staticName** – if set the generated constructor will be private and Lombok will add static method with the same set of parameters that will call the corresponding private constructor

@Data - one annotation to rule them all

- Used on class as shortcut for @ToString, @EqualsAndHashCode, @Getter, @Setter, @RequiredArgsConstructor
- You can customize parameters for any of the above annotations by adding it and providing the necessary values
- **staticConstructor** – a parameter that tells Lombok to generate static constructor method and constructor for the class to be private. The name of the method is the value of staticConstructor parameter.

Where we can use these annotations?

- Any class that conforms to Java Bean specification
  - DTO classes
  - JPA entity beans

One thing to note

NOTE: Be careful with relational fields that build bi-directional relations between classes.

- Must use **of** or **exclude** parameters for @EqualsAndHashCode and @ToString in order to avoid **StackOverflowError**

Creating immutable classes

- **@Value** – equivalent to **@Data** but for immutable classes
- Converts all non-final fields to final and provides only get methods to access them.
- **@Wither** annotation placed on a field adds **“set”** method that returns new instance of the class with changed value in that field.

Builders made easy

- **@Builder** – annotation that generates builder API for your data class
- This annotation has parameters to customize Builder class name, build and builder method names
- **@Singular** – annotation used on fields that are collections from **java.util** or **com.google.common.collect**

Note: This annotation requires from you to provide values for all fields including for those having initial value

More for @Builder

- @Builder.DEFAULTS – a new annotation that is supposed to “help” with initial values
  - **Better don’t use it**. It breaks your code when you try to directly create instance of your class.

@Synchronized

- Can be used on methods but does not synchronize on **this** like **synchronized** method modifier
- Synchronizes on private field **$lock** which is automatically created if it is missing
- You can specify another field to use for synchronization

try-with-resource without **java.io.Closeable**

Resource allocator class

```
class RAllocator {
    void doSomethingWithResource() {}
    void dispose() {}
}
```

try-with-resource without **java.io.Closeable**

Code before compilation:

```
public class CleanupExample {
  public static void main(String[] args) {
    @Cleanup("dispose")
    RAllocator allocator = new RAllocator();
    allocator.doSomethingWithResource();
  }
}
```

try-with-resource without **java.io.Closeable**

Code after compilation:

```
public class CleanupExample {
      public CleanupExample() {}
      public static void main(String[] args) {
        RAllocator allocator = new RAllocator ();
        try {
           allocator.doSomethingWithResource();
        } finally {
          if (Collections.singletonList(allocator).get(0) != null) {
              allocator.dispose();
          }
        }
      }
   }
```

Other useful annotations

- @Log, @Log4j, @CommonsLog, @JBossLog, @Log4j2, @Slf4j, @XSlf4j – add and initialize **private static final log** field to your code for the corresponding logging framework

Other useful annotations

@NonNull – automatic null check

- when used on method parameter it adds check in the code to generate **NullPointerException** in case parameter is **null**. The message of the exception contains the name of the parameter.

- when used on field it indicates that this field should not be null and all methods that set value to this field **will contain null check**

Other useful annotations

- **@SneakyThrows** - throw checked exceptions without **throws**

Source code:

```
public class SneakyThrowExample {
    public static void main(String[] args) {
        new SneakyThrowExample();
    }
    @SneakyThrows
    public SneakyThrowExample() {
        throw new Exception();
    }
}
```

Other useful annotations

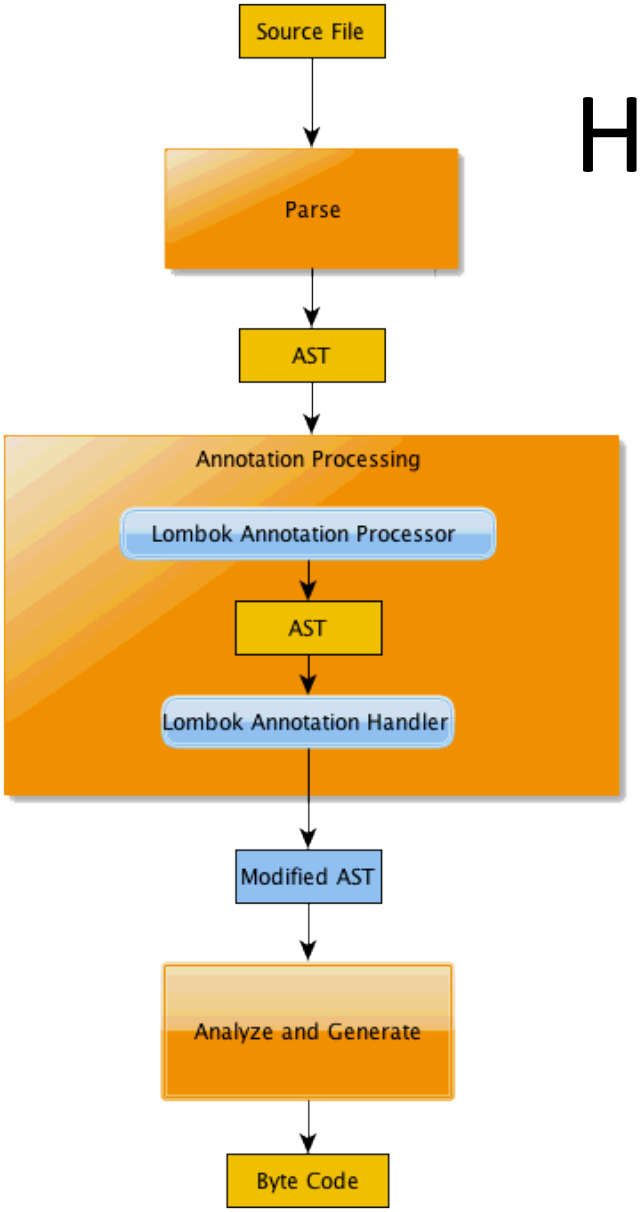
- **@SneakyThrows** - throw checked exceptions without **throws**

Compiled code from class file

```
public class SneakyThrowExample {
    public static void main(String[] args) {
        new SneakyThrowExample();
    }
    public SneakyThrowExample() {
        try {
            throw new Exception();
        } catch (Throwable ex) {
            throw ex;
        }
    }
}
```

@Accessors

- **chain –** set this parameter to **true** to have your setter methods return **this.**
- **fluent** – Set this parameter to **true** if you prefer fluent API instead of **get/set** naming schema. This sets **chain** to **true**.
- **prefix** – comma separated list of prefixes used in the naming of your fields. Lombok will remove the prefix from the field name when evaluating the name for the corresponding get/set method



How it works

Lombok works at annotation processing level and provides access to Abstract Syntax Tree to it's annotation handlers.

The annotation handlers do their magic by altering that AST – adding classes, fields, methods and expressions to generate the necessary code.

BULGARIAN Java User Group

J2D

How to add your own extensions

- Create a fork from Lombok source code on **github**
- Add your code in the corresponding packages
- Build distribution and use it in your projects
- Contribute back your code if you think it can be used by others

Integration with your IDE

- Supported IDE's are
  - **Eclipse** and other eclipse based IDE– supported by lombok installer
  - **NetBeans** – just add **lombok.jar** to your project libraries and **enable** annotation processing in Editor
  - **IntelliJ** – requires third-party plug-in that is available in IntelliJ repositories

Resources

- https://projectlombok.org
- How to create extensions (old but gives general idea what to do) https://notatube.blogspot.bg/2010/12/project-lombok-creating-custom.html
- YouTube – lots of tutorials and presentations from other conferences

Q & A

Thank you