Hello everyone,

Who am I

First let me introduce myself. My name is Doychin Bondzhev and I have my own product company. Just like all of you who are here I'm also member of Bulgarian Java User group. I live in Plovdiv and am I'm the organizer of Java Beer in Plovdiv. I try to do such event every 3 months. Next one will be in December. And finally I'm also expert group member for JSR-377.

That is enough for me.

How to make our job easier

In our work we have to read and write lots of code. As Java developers we have to work in the boundaries set by the Java language. Sometimes we wish there was better way of expressing our ideas. Some new languages in the last few years are trying to offer us solutions.

But we are java developers

But we are Java developers and we want to write Java code. As you know Java is verbose language and sometimes most of the code we write does not bring any useful information. It just needs to be there in order to make things to work.

Let me start with one simple example:

NOTE: START FROM CLASS THAT HAS ONLY FIELD DEFINITINS AND USE IDE TO GENERATE ALL GET/SET, equals , hashCode and toString METHODS.

```
public class Person implements Serializable {

    private int key;

    private String firstName;

    private String secondName;

    private String thirdName;

    private int age;

    private int yearOfBirth;

    private double salary;
}
```

As you can see there is a lot of code that needs to be there. All of that code does not bring any useful information. It is there only to provide needed functionality to allow access to the fields and cover all usage scenarios.

So today I'll talk about how to make our lives easier by reducing the amount of code we have to write and maintain. I will also talk how to make the job of our colleagues easier by shrinking the code they have to read and understand.

What if we can use some tools that will auto-generate all that boilerplate code for us?

How to do it in Java?

This is where Project Lombok comes to help.

How many of you know what is project Lombok?

Lombok provides us with some helpful annotations that can save us from all that extra code.

How to use Lombok

You have to download it from https://projectlombok.org and add it to your build classpath. If you use maven or gradle you have to add it as provided dependency in your project.

Now let's start with some of the annotations provided by Lombok.

@Getter and @Setter

First I will start with two annotations that can help us get rid of all these set and get methods.

For that we have two annotations @Getter and @Setter. These two annotations can be used on field or type. In the first case they will tell Lombok to generate get/set method for the field where annotation was used. In case we use them on a type Lombok will generate get/set methods for all non-static fields that don't have get/set method. There are annotation parameters that you can use to control the generated code. First parameter controls generated method access level, which by default is public. The other parametr you can use to list the annotations that need to be placed on the generated method.

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

Next we need to add **equals()** and **hashCode()** methods. For that purpose we have annotation @EqualsAndHashCode

Also we will use @ToString to add **toString()** method.

Both of these annotations have common set of parameters. First is **callSuper** which tell Lombok to include call to corresponding super method. This is useful when you are building complex hierarchies of classes. Other two parameters are **of** and **exclude.** They are used to control the set of fields included in the evaluation of the result for the corresponding method. These two can't be used at the same time. The result will be compilation error.

Constructors

Finally we need to add constructor to our class. Lombok gives us three annotations for constructor generation:

- @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

They also have some common parameters like **access** that controls the access level of the constructor and **staticName** if you want your constructor to be private and to have static constructor method.

As you can see our class is now much smaller in size then what was in the original code.

@Data

But we can also make it even smaller. For that purpose there is the @Data annotation which tells Lombok that we want this class to be data class and have getXXX/setXXX, equals(), hashCode(), toString() methods generated for us using default parameters and constructor that accepts as parameters all required fields.

@Data annotation has one parameter **`staticConstructor`**. If this parameter is used our class will have its constructor defined as private and there will be static method that will have the name of the value of that parameter. This method will return instance of our class created with the private constructor.

If we need to change something in the default parameters for code generation, we can always add corresponding annotation and provide the necessary parameters.

For example if I want to avoid age to be used in hashCode() and equals() we have to add

```
@EqualsAndHashCode(exclude = "age")
```

The good thing is that we can always write our own get/set, equals(), hashCode() or toString() and Lombok will generate only the missing methods.

Where we can use these annotations?

DTO, JPA entity classes – all classes that are Java bean classes.

NOTE: EXAMPLE with JPA Class and bi-directional relations

One thing to note

When you have complex data models that have relations between classes you have to be careful with bi-directional relations. You should use **of** or **exclude** parameters for @EqualsAndHashCode and @ToString in order to avoid **StackOverflowError**

Immutable classes

Now let's look at another case. You want to create an immutable class. For that you will use @Value annotation. It is the immutable version of @Data annotation.

As result you will have class with all non-final fields now final. Class constructor that accepts all non-final fields, getter methods and corresponding **hashCode(), equals()** and **toString().**

With @Wither annotation placed on any of the fields you will have "set" method that will return new instance of your class but with changed value in that field.

Builder

Another interesting feature offered by Lombok is @Builder annotation. This annotation will generate for us builder class that we can use to build instances of our own class. It can also be used on static constructor methods or class constructors.

This annotation has parameters which can be used to customize the name of the builder method, the name of the build method and the name of the Builder class.

Note: Be careful with field initial values.

When there is initial value for a field and we use Builder to build an instance we have to set that value to. Otherwise builder class will set the default value for type of the field.

To help with collection fields there is **@Singular** annotation. It is used on a field or parameter that is of collection type. It will add two adder methods that will add elements to the collection instead of one set method. These methods will return the type of the builder class so you can chain calls and fill the collection by adding elements one by one.

@Builder.DEFAULTS

In order to solve the problem with initial values the annotation @Builder.DEFAULTS was added. This annotation has to be placed on all fields that have initial value to tell Lombok to use that value during the build process in case that field was skipped.

This feature has one nasty problem. It breaks the default behavior when you try to directly create instance of your class!!!

Better don't use it until problem is fixed in later version.

@Synchronized

Another useful annotation is @Synchronized. You can use it to annotate methods. The difference is that Lombok will add $lock field in your class if it is missing and will synchronize on the instance in that field. You can also specify the name of an existing field and generated code will use that field instead of the $lock.

```
public class Synchronized {
```

```
    Object lock = new Object();

    @lombok.Synchronized
    public void doSomething() {
        System.out.println();
    }

    @lombok.Synchronized("lock")
    public void doSomethingElse() {
        System.out.println();
    }
}
```

try-with-resources without java.io.Closeable

In Lombok you can use **@Cleanup** to mark a resource that needs to be closed after it is used. The benefit of that annotation is that you can use it on resources that do not implement **java.io.Closeable** interface

EXAMPLE with @Cleanup

```
public class CleanupExample {

    public static void main(String[] args) {
        @Cleanup("dispose")
        ResourceAllocator allocator = new ResourceAllocator();
        allocator.doSomethingWithResource();

    }
}

class ResourceAllocator {

    void doSomethingWithResource() {
    }

    void dispose() {
    }
}
```

Other useful annotations

@Log, @Log4j, @CommonsLog, @JBossLog, @Log4j2, @Slf4j, @XSlf4j

Next set of annotations is used to create **private static final log** field in your class. You have to use the annotation that corresponds to the logging library you use in your project.

By default logger initialization will use your class name but you can customize that part by specifying value for topic parameter.

@NonNull

Checking method parameters for null value is something we have to do when we want to be sure that all parameters contain valid values. But this can be done by just adding @NonNull annotation in

front of our parameters. Lombok will generate for us checks and will throw **NullPointerException** with message that will tell us which parameter is null.

@SneakyThrows

This annotation allows us to have checked exceptions thrown in our code but without having to add them to throws clause of your method or constructor. This feature must be used very carefully.

One usage scenarios is when you call some API method that throws checked exception but you are sure that checked exception will never happen. Then you don't want to wrap all the code in try{} catch{} block just to catch that exception and swallow it or throw it in a wrapper unchecked exception. SneakyThrows will generate for you that try {} catch {} block, but instead of wrapping the exception it will throw the original checked exception.

@Accessors

This annotation is used to adjust code generation for getter and setter methods. It has three parameters

How it works

With just a few words Lombok is annotation processor that modifies the Abstract Syntax Tree generated by Java parser. Modifications happen in the corresponding handlers for each annotation. These handlers alter the AST by adding or altering classes, method, fields and expressions.

From there the compiler uses that modified AST to generate the necessary byte code.

Extending Lombok with custom functionality

For the moment there is only one way to add your own custom annotations and code that will process them. You have to fork Lombok code from github and you have to add your own code inside existing Lombok packages. The reason is that Lombok uses custom class loader that is used to hide the internals of Lombok

IDE integration

Out of the box Lombok provides integration with Eclipse and NetBeans. For IntelliJ Lombok plug-in is required in order to use Code completion. Also plug-in provides Lombok and Delombok refactoring.