(/)

Introduction to Project Lombok

Last modified: February 10, 2019

by baeldung (/author/baeldung/)

Programming (/category/programming/)

Lombok (/tag/lombok/)

Lombok (https://projectlombok.org) is one of the tools I literally always drop into my projects builds the first. I couldn't imagine myself programming Java without it these days. I really hope you find its power reading this article!

1. Avoid Repetitive Code

Java is a great language but it sometimes gets too verbose for things you have to do in your code for common tasks or compliancy with some framework practices. These do very often bring no real value to the

business side of your programs – and this is where Lombok is here to make your life happier and yourself more productive.

The way it works is by plugging into your build process and autogenerating Java bytecode into your *.class* files as per a number of project annotations you introduce in your code.

Including it in your builds, whichever system you are using, is very straight forward. Their project page (https://projectlombok.org/features/index.html) has detailed instructions on the specifics. Most of my projects are maven based, so I just typically drop their dependency in the *provided* scope and I'm good to go:

```
<dependencies>
    ...
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.16.8</version>
        <scope>provided</scope>
    </dependency>
    ...
</dependencies>
```

Check for the most recent available version here (https://projectlombok.org/changelog.html).

Note that depending on Lombok won't make users of your *.jar*s depend on it as well, as it is a pure build dependency, not runtime.

2. Getters/Setters, Constructors – So Repetitive

Encapsulating object properties via public getter and setter methods is such a common practice in the Java world, and lots of frameworks rely on this "Java Bean" pattern extensively: a class with an empty constructor and get/set methods for "properties".

This is so common that most IDE's support autogenerating code for these patterns (and more). This code however needs to live in your sources and also be maintained when, say, a new property is added or a field renamed.

Let's consider this class we want to use as a JPA entity as an example:

```
@Entity
public class User implements Serializable {

    private @Id Long id; // will be set when persisting

    private String firstName;
    private String lastName;
    private int age;

    public User() {
    }

    public User(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    // getters and setters: ~30 extra lines of code
}
```

This is a rather simple class, but still consider if we added the extra code for getters and setters we'd end up with a definition where we would have more boilerplate zero-value code than the relevant business information: "a User has first and last names, and age."

Let us now *Lombok-ize* this class:

```
@Entity
@Getter @Setter @NoArgsConstructor // <--- THIS is it
public class User implements Serializable {

    private @Id Long id; // will be set when persisting

    private String firstName;
    private String lastName;
    private int age;

    public User(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
}
```

By adding the *@Getter* and *@Setter* annotations we told Lombok to, well, generate these for all the fields of the class. *@NoArgsConstructor* will lead to an empty constructor generation.

Note this is the **whole** class code, I am not omitting anything as opposed to the version above with the *// getters and setters* comment. For a three relevant attributes class, this is a significant saving in code!

If you further add attributes (properties) to your *User* class, the same will happen: you applied the annotations to the type itself so they will mind all fields by default.

What if you wanted to refine the visibility of some properties? For example, I like to keep my entities' *id* field modifiers *package* or *protected* visible because they are expected to be read but not explicitly set by application code. Just use a finer grained *@Setter* for this particular field:

```
1 private @Id @Setter(AccessLevel.PROTECTED) Long id;
```

3. Lazy Getter

Often, applications need to perform some expensive operation and save the results for subsequent use.

For instance, let's say we need to read static data from a file or a database. It's generally a good practice to retrieve this data once and then cache it to allow in-memory reads within the application. This saves the application from repeating the expensive operation.

Another common pattern is to **retrieve this data only when it's first needed**. In other words, **only get the data when the corresponding getter is called the first time. This is called *lazy-loading*.**

Suppose that this data is cached as a field inside a class. The class must now make sure that any access to this field returns the cached data. One possible way to implement such a class is to make the getter method retrieve the data only if the field is *null*. For this reason, **we call this a *lazy* getter**.

Lombok makes this possible with the ***lazy* parameter in the @*Getter* annotation** we saw above.

For example, consider this simple class:

```
public class GetterLazy {

    @Getter(lazy = true)
    private final Map<String, Long> transactions = readTxnsFromFile()

    private Map<String, Long> readTxnsFromFile() {

        final Map<String, Long> cache = new HashMap<>();
        List<String> txnRows = readTxnListFromFile();

        txnRows.forEach(s -> {
            String[] txnIdValueTuple = s.split(DELIMETER);
            cache.put(txnIdValueTuple[0], Long.parseLong(txnIdValueTu
        });

        return cache;
    }
}
```

This reads some transactions from a file into a *Map*. Since the data in the file doesn't change, we'll cache it once and allow access via a getter.

If we now look at the compiled code of this class, we'll see a **getter method which updates the cache if it was *null* and then returns the cached data**:

```
public class GetterLazy {

    private final AtomicReference<Object> transactions = new AtomicRe

    public GetterLazy() {
    }

    //other methods

    public Map<String, Long> getTransactions() {
        Object value = this.transactions.get();
        if (value == null) {
            synchronized(this.transactions) {
                value = this.transactions.get();
                if (value == null) {
                    Map<String, Long> actualValue = this.readTxnsFrom
                    value = actualValue == null ? this.transactions :
                    this.transactions.set(value);
                }
            }
        }

        return (Map)((Map)(value == this.transactions ? null : value)
    }
}
```

It's interesting to point out that **Lombok wrapped the data field in an *AtomicReference (https://www.baeldung.com/java-atomic-variables).*** This ensures atomic updates to the *transaction* field. The *getTransactions()* method also makes sure to read the file if *transactions* is *null.*

The use of the *AtomicReference transactions* field directly from within the class is discouraged. **It's recommended to use the *getTransactions()* method for accessing the field.**

For this reason, if we use another Lombok annotation like *ToString* in the same class, it will use *getTransactions()* instead of directly accessing the field.

4. Value Classes/DTO's

There are many situations in which we want to define a data type with the sole purpose of representing complex "values" or as "Data Transfer Objects", most of the time in the form of immutable data structures we build once and never want to change.

We design a class to represent a successful login operation. We want all fields to be non-null and objects be immutable so that we can thread-safely access its properties:

```
public class LoginResult {

    private final Instant loginTs;

    private final String authToken;
    private final Duration tokenValidity;

    private final URL tokenRefreshUrl;

    // constructor taking every field and checking nulls

    // read-only accessor, not necessarily as get*() form
}
```

Again, the amount of code we'd have to write for the commented sections would be of a much larger volume that the information we want to encapsulate and that has real value for us. We can use Lombok again to improve this:

```
@RequiredArgsConstructor
@Accessors(fluent = true) @Getter
public class LoginResult {

    private final @NonNull Instant loginTs;

    private final @NonNull String authToken;
    private final @NonNull Duration tokenValidity;

    private final @NonNull URL tokenRefreshUrl;

}
```

Just add the *@RequiredArgsConstructor* annotation and you'd get a constructor for all the final fields int the class, just as you declared them. Adding *@NonNull* to attributes makes our constructor check for nullability and throw *NullPointerExceptions* accordingly. This would also happen if the fields were non-final and we added *@Setter* for them.

Don't you want boring old *get*()* form for your properties? Because we added *@Accessors(fluent=true)* in this example "getters" would have the same method name as the properties: *getAuthToken()* simply becomes *authToken()*.

This "fluent" form would apply to non-final fields for attribute setters and as well allow for chained calls:

```
// Imagine fields were no longer final now
return new LoginResult()
  .loginTs(Instant.now())
  .authToken("asdasd")
  . // and so on
```

5. Core Java Boilerplate

Another situation in which we end up writing code we need to maintain is when generating *toString()*, *equals()* and *hashCode()* methods. IDEs try to help with templates for autogenerating these in terms of our class attributes.

We can automate this by means of other Lombok class-level annotations:

- ***@ToString* (https://projectlombok.org/features/ToString.html)**: will generate a *toString()* method including all class attributes. No need to write one ourselves and maintain it as we enrich our data model.

- ***@EqualsAndHashCode* (https://projectlombok.org/features/EqualsAndHashCode.html)**: will generate both *equals()* and *hashCode()* methods by default considering all relevant fields, and according to **very well though semantics (http://www.artima.com/lejava/articles/equality.html)**.

These generators ship very handy configuration options. For example, if your annotated classes take part of a hierarchy you can just use the *callSuper=true* parameter and parent results will be considered when generating the method's code.

More on this: say we had our *User* JPA entity example include a reference to events associated to this user:

```
1 @OneToMany(mappedBy = "user")
2 private List<UserEvent> events;
```

We wouldn't like to have the hole list of events dumped whenever we call the *toString()* method of our User, just because we used the *@ToString* annotation. No problem: just parameterize it like this: *@ToString(exclude = {"events"})*, and that won't happen. This is also helpful to avoid circular references if, for example, *UserEvent*s had a reference to a *User*.

For the *LoginResult* example, we may want to define equality and hash code calculation just in terms of the token itself and not the other final attributes in our class. Then, simply write something like *@EqualsAndHashCode(of = {"authToken"})*.

Bonus: if you liked the features from the annotations we've reviewed so far you may want to examine *@Data* (https://projectlombok.org/features/Data.html) and *@Value* (https://projectlombok.org/features/Value.html) annotations as they behave as if a set of them had been applied to our classes. After all, these discussed usages are very commonly put together in many cases.

6. The Builder Pattern

The following could make for a sample configuration class for a REST API client:

```
public class ApiClientConfiguration {

    private String host;
    private int port;
    private boolean useHttps;

    private long connectTimeout;
    private long readTimeout;

    private String username;
    private String password;

    // Whatever other options you may thing.

    // Empty constructor? All combinations?

    // getters... and setters?
}
```

We could have an initial approach based on using the class default empty constructor and providing setter methods for every field. However, we'd ideally want configurations no to be *set*-ed once they have been built (instantiated), effectively making them immutable. We therefore want to avoid setters, but writing such a potentially long args constructor is an anti-pattern.

Instead, we can tell the tool to generate a *builder* pattern, preventing us to write an extra *Builder* class and associated fluent setter-like methods by simply adding the @Builder annotation to our *ApiClientConfiguration*.

```
@Builder
public class ApiClientConfiguration {

    // ... everything else remains the same

}
```

Leaving the class definition above as such (no declare constructors nor setters + *@Builder*) we can end up using it as:

```
ApiClientConfiguration config =
    new ApiClientConfigurationBuilder()
        .host("api.server.com")
        .port(443)
        .useHttps(true)
        .connectTimeout(15_000L)
        .readTimeout(5_000L)
        .username("myusername")
        .password("secret")
     .build();
```

7. Checked Exceptions Burden

Lots of Java APIs are designed so that they can throw a number of checked exceptions client code is forced to either *catch* or declare to *throws*. How many times have you turned these exceptions you know won't happen into something like this?

```
public String resourceAsString() {
    try (InputStream is = this.getClass().getResourceAsStream("sure_ir
        BufferedReader br = new BufferedReader(new InputStreamReader(
        return br.lines().collect(Collectors.joining("\n"));
    } catch (IOException | UnsupportedCharsetException ex) {
        // If this ever happens, then its a bug.
        throw new RuntimeException(ex); <--- encapsulate into a Runtir
    }
}
```

If you want to avoid this code patterns because the compiler won't be otherwise happy (and, after all, you **know** the checked errors cannot happen), use the aptly named *@SneakyThrows* (https://projectlombok.org/features/SneakyThrows.html):

```
@SneakyThrows
public String resourceAsString() {
    try (InputStream is = this.getClass().getResourceAsStream("sure_ir
        BufferedReader br = new BufferedReader(new InputStreamReader(
        return br.lines().collect(Collectors.joining("\n"));
    }
}
```

8. Ensure Your Resources are Released

Java 7 introduced the try-with-resources block to ensure your resources held by instances of anything implementing *java.lang.AutoCloseable* are released when exiting.

Lombok provides an alternative way of achieving this, and more flexibly via @Cleanup (https://projectlombok.org/features/Cleanup.html). Use it for any local variable whose resources you want to make sure are released. No need for them to implement any particular interface, you'll just get its *close()* method called.

```
1  @Cleanup InputStream is = this.getClass().getResourceAsStream("res.tx
```

Your releasing method has a different name? No problem, just customize the annotation:

```
1  @Cleanup("dispose") JFrame mainFrame = new JFrame("Main Window");
```

9. Annotate Your Class To Get a Logger

Many of us add logging statements to our code sparingly by creating an instance of a *Logger* from our framework of choice. Say, SLF4J:

```
1  public class ApiClientConfiguration {
2
3      private static Logger LOG = LoggerFactory.getLogger(ApiClientConf
4
5      // LOG.debug(), LOG.info(), ...
6
7  }
```

This is such a common pattern that Lombok developers have cared to simplify it for us:

```
1  @Slf4j // or: @Log @CommonsLog @Log4j @Log4j2 @XSlf4j
2  public class ApiClientConfiguration {
3
4      // log.debug(), log.info(), ...
5
6  }
```

Many logging frameworks (https://projectlombok.org/features/Log.html) are supported and of course you can customize the instance name, topic, etc.

10. Write Thread-Safer Methods

In Java you can use the *synchronized* keyword to implement critical sections. However, this is not a 100% safe approach: other client code can eventually also synchronize on your instance, potentially leading to unexpected deadlocks.

This is where *@Synchronized* (https://projectlombok.org/features/Synchronized.html)comes in: annotate your methods (both instance and static) with it and you'll get an autogenerated private, unexposed field your implementation will use for locking:

```
@Synchronized
public /* better than: synchronized */ void putValueInCache(String key
    // whatever here will be thread-safe code
}
```

11. Automate Objects Composition

Java does not have language level constructs to smooth out a "favor composition inheritance" approach. Other languages have built-in concepts such as *Traits* or *Mixins* to achieve this.

Lombok's @Delegate (https://projectlombok.org/features/experimental/Delegate.html) comes in very handy when you want to use this programming pattern. Let's consider an example:

- We want *User*s and *Customer*s to share some common attributes for naming and phone number
- We define both an interface and an adapter class for these fields
- We'll have our models implement the interface and *@Delegate* to their adapter, effectively *composing* them with our contact information

First, let's define an interface:

```
public interface HasContactInformation {

    String getFirstName();
    void setFirstName(String firstName);

    String getFullName();

    String getLastName();
    void setLastName(String lastName);

    String getPhoneNr();
    void setPhoneNr(String phoneNr);

}
```

And now an adapter as a *support* class:

```
@Data
public class ContactInformationSupport implements HasContactInformat

    private String firstName;
    private String lastName;
    private String phoneNr;

    @Override
    public String getFullName() {
        return getFirstName() + " " + getLastName();
    }
}
```

The interesting part comes now, see how easy it is to now compose contact information into both model classes:

```
public class User implements HasContactInformation {

    // Whichever other User-specific attributes

    @Delegate(types = {HasContactInformation.class})
    private final ContactInformationSupport contactInformation =
            new ContactInformationSupport();

    // User itself will implement all contact information by delegat

}
```

The case for *Customer* would be so similar we'd omit the sample for brevity.

12. Rolling Lombok Back?

Short answer: Not at all really.

You may be worried there is a chance that you use Lombok in one of your projects, but later want to rollback that decision. You'd then have a maybe large number of classes annotated for it… what could you do?

I have never really regretted this, but who knows for you, your team or your organization. For these cases you're covered thanks to the *delombok* tool from the same project.

By *delombok-ing* your code you'd get autogenerated Java source code with exactly the same features from the bytecode Lombok built. So then you may simply replace your original annotated code with these new *delomboked* files and no longer depend on it.

This is something you can integrate in your build (https://projectlombok.org/features/delombok.html) and I have done this in the past to just study the generated code or to integrate Lombok with some other Java source code based tool.

13. Conclusion

There are some other features we have not presented in this article, I'd encourage you to take a deeper dive into the feature overview (https://projectlombok.org/features/index.html) for more details and use cases.

Also most functions we've shown have a number of customization options you may find handy to get the tool generate things the most compliant with your team practices for naming etc. The available built-in configuration system (https://projectlombok.org/features/configuration.html) could also help you with that.

I hope you have found the motivation to give Lombok a chance to get into your Java development toolset. Give it a try and boost your productivity!

The example code can be found in the GitHub project (https://github.com/eugenp/tutorials/tree/master/lombok).

CATEGORIES

SPRING (/CATEGORY/SPRING/)
REST (/CATEGORY/REST/)
JAVA (/CATEGORY/JAVA/)
SECURITY (/CATEGORY/SECURITY-2/)
PERSISTENCE (/CATEGORY/PERSISTENCE/)
JACKSON (/CATEGORY/JSON/JACKSON/)
HTTP CLIENT (/CATEGORY/HTTP/)
KOTLIN (/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)