TOOLS

Project Lombok: Clean, Concise Code

Try adding Lombok to an application and see how many lines of code you can cut.

by Josh Juneau

May 1, 2017

Imagine that you are coding a Java application and creating a plain old Java object (POJO), a Java class with several private fields that will require getter and setter methods to provide access. How many lines of code will be needed to generate getters and setters for each of the fields? Moreover, adding a constructor and a `toString()` method will cause even more lines of code and clutter. That is a lot of boilerplate code. How about when you are utilizing Java objects that need to be closed after use, so you need to code a `finally` block or use try-with-resources to ensure that the object closing occurs? Adding `finally` block boilerplate to close objects can add a significant amount of clutter to your code.

Project Lombok is a mature library that reduces boilerplate code. The cases mentioned above cover just a few of those where Project Lombok can be a great benefit. The library replaces boilerplate code with easy-to-use annotations. In this article, I examine several useful features that Project Lombok provides—making code easier to read and less error-prone and making developers more productive. Best of all, the library works well with popular IDEs and provides a utility to "delombok" your code by reverting—that is, adding back all the boilerplate that was removed when the annotations were added.

Check for Nulls

Let's start with one of the most basic utilities that Lombok has to offer. The `@NonNull` annotation, which should not be confused with the Bean Validation annotation, can be used to generate a null check on a setter field. The check throws a

`NullPointerException` if the annotated class field contains a null value. Simply apply it to a field to enforce the rule:

```
@NonNull @Setter
private String employeeId;
```

This code generates the following code:

```
public id setEmployeeId(@NonNull final String
   if(employeeId == null) throw
      new java.lang.NullPointerException("emp
   this.employeeId = employeeId;
```

Primitive parameters cannot be annotated with `@NonNull`. If they are, a warning is issued and no null check is generated.

Concise Data Objects

Writing a POJO can be laborious, especially if there are many fields. If you are developing a POJO, you should always provide private access directly to the class fields, while creating accessor methods—getters and setters—to read from and write to those fields. Although developing accessor methods is easy, they generally are just boilerplate code. Lombok can take care of generating these methods if a field is annotated with `@Getter` and `@Setter`. Therefore, the following two code listings provide the exact same functionality.

Without Project Lombok:

```
private String columnName;
public String getColumnName(){
   return this.columnName;
} public void setColumnName(String columnName
   this.columnName = columnName;
}
```

Using Project Lombok:

```
@Getter @Setter private String columnName;
```

As you can see, Lombok not only makes the code more concise, but it also makes the code easier to read and less error-prone. These annotations also accept an optional parameter to designate the access level if needed. More good news: `@Getter` and `@Setter` respect the proper naming conventions, so generated code for a Boolean field results in accessor methods beginning with *is* rather than *get*. If they are applied at the class level, getters and setters are generated for each nonstatic field within the class.

In many cases, data objects also should contain the `equals()`, `hashCode()`, and `toString()` methods. This boilerplate can be taken care of by annotating a class with the `@EqualsAndHashCode` and `@ToString` annotations, respectively. These annotations cause Lombok to generate the respective methods, and they are customizable so that you can specify field exclusions and other factors. By default, any nonstatic or nontransient fields are included in the logic that is used to compose these methods. These annotations use the attribute `exclude` to specify methods that should not be included in the logic. The `callSuper` attribute accepts a `true` or `false`, and it indicates whether to use the `equals()` method of the superclass to verify equality. The following code demonstrates the use of these annotations.

```
@EqualsAndHashCode
@ToString(exclude={" columnLabel"})
public class ColumnBean {
   private BigDecimal id;
   private String columnName;
   private String columnLabel;
}
```

The `@Data` annotation can be used to apply functionality behind all the annotations discussed thus far in this section. That is, simply annotating a class with `@Data` causes Lombok to generate getters and setters for each of the nonstatic class fields and a class constructor, as well as the `toString()`, `equals()`, and `hashCode()` methods. It also creates a constructor that accepts any final fields or those annotated with `@NonNull` as arguments. Finally, it generates default `toString()`, `equals()`, and `hashCode()` methods that take all class fields and methods into consideration. This makes the coding of a POJO very easy, and it is much the same as some alternative languages, such as Groovy, that offer similar features. **Listing 1** (all listings for this article can be downloaded here) shows the full Java code for the POJO that is generated by the following code:

```
@Data
public class ColumnBean {
   @NonNull
   private BigDecimal id;
   @NonNull
   private String columnName;
   @NonNull
   private String columnLabel;
}
```

Note that if you create your own getters or setters, Lombok does not generate the code even if the annotations are present. This can be handy if you wish to develop a custom getter or setter for one or more of the class fields.

If you are merely interested in having constructors generated automatically, `@AllArgsConstructor` and `@NoArgsConstructor` might be of use. `@AllArgsConstructor` creates a constructor for the class using all the fields that have been declared. If a field is added or removed from the class, the generated constructor is revised to accommodate this change. This behavior can be convenient for ensuring that a class constructor always accepts values for each of the class fields. The disadvantage of using this annotation is that reordering the class fields causes the constructor arguments to be reordered as well, which could introduce bugs if there is code that depends upon the position of arguments when generating the object. `@NoArgsConstructor` simply generates a no-argument constructor.

The `@Value` annotation is similar to the `@Data` annotation, but it generates an immutable class. The annotation is placed at the class level, and it invokes the automatic generation of getters for all private final fields. No setters are generated, and the class is marked as `final`. Lastly, the `toString()`, `equals()`, and `hashCode()` methods are generated, and a constructor is generated that contains arguments for each of the fields.

Can't My IDE Do That?

You might be asking yourself, "Can't my IDE already do that sort of refactoring?" Most modern IDEs—such as NetBeans, Eclipse, and IntelliJ—offer features such as encapsulation of fields and auto-generation of code. These abilities are great because they can significantly increase productivity. However, these capabilities do not reduce code clutter, so they can lead to refactoring down the road. Let's say your Java object has 10 fields. To conform to a JavaBean, it will contain 20 accessor methods (one getter and setter pair per field). That's a lot of clutter. Also, what happens when you decide to change one of your field names? You'll have to do some refactoring in order to change it cleanly. If you're using Lombok, you simply change the field name and move on with your life.

Builder Objects

Sometimes it is useful to have the ability to develop a builder object, which allows objects to be constructed using a step-by-step pattern with controlled construction. For example, in some cases large objects require several fields to be populated, which can be problematic when such an object is implemented via a constructor.

Lombok makes it simple to create builder objects in much the same way that it enables easy POJO creation. Annotating a class with `@Builder` produces a class that adheres to the builder pattern—that is, an inner builder class is produced that contains each of the class fields. ("Builder" is preceded by the name of the class. So a class named `Foo` has a `FooBuilder`

class generated.) The generated builder class contains a “setter” method for each of the class fields, but the names of the methods do not include the usual “set” prefix. The methods themselves set the value that is passed into the methods, and then they return the builder object. **Listing 2** in the downloadable code demonstrates a class that contains a builder, and **Listing 3** demonstrates the same object annotated with `@Builder`.

Several variations can be used with `@Builder`. For example, the annotation can be placed on the class, on a constructor, or on a method. Placing the annotation on a constructor produces the same builder object shown in **Listing 2**, but it generates methods for each of the constructor’s arguments in the builder. This means that you can omit a class field from the constructor, or you can choose to include a superclass field in the constructor. The only way to include superclass fields in a builder is for an object to contain a superclass.

The `toBuilder` attribute of the `@Builder` annotation accepts `true` or `false`, and it can be used to designate whether a `toBuilder()` method is included in the generated builder object. This method copies the contents of an existing object of the same type.

It is possible to treat one of the fields as a builder collection by annotating it with `@Singular`. This causes two adder methods to be generated—one to add a single element and another to add all elements. This annotation also causes a `clear()` method to be generated, which clears the contents of the collection.

Easy Cleanup

Lombok makes it easy to clean up resources as well. How often have you either forgotten to close a resource or written lots of boilerplate `try-catch` blocks to accommodate resource closing? Thanks to the `@Cleanup` annotation, you no longer need to worry about forgetting to release a resource.

Although the Java language now contains the try-with-resources statement to help close resources, `@Cleanup` can be a useful alternative in some cases, because it causes a `try-finally` block to be generated around the subsequent code, and then it calls the annotated resource’s `close()` method. If the cleanup method for a given resource is not named `close()`, the cleanup method name can be specified with the annotation’s value attribute. **Listing 4** in the downloadable code demonstrates a block of code that contains some lines to manually close the resource. **Listing 5** demonstrates the same block of code using `@Cleanup`.

It is important to note that in a case where code throws an exception and then subsequent code invoked via `@Cleanup`

also throws an exception, the original exception will be hidden by the subsequently thrown exception.

Locking Safely

To ensure safety by having only one thread that can access a specified method at a time, the method should be marked as `synchronized`. Lombok supplies an even safer way to ensure that only one thread can access a method at a time: the `@Synchronized` annotation. This annotation can be used only on static and instance methods, just like the `synchronized` keyword. However, rather than locking on this, the annotation locks on a private field named `$lock` for nonstatic methods and on `$LOCK` for static methods. This field is autogenerated if it does not already exist, or you can create it yourself. You can also specify a different lock field by specifying it as a parameter to `@Synchronized`. The following code illustrates the use of `@Synchronized`:

```
@Synchronized
public static void helloLombok() {
   System.out.println("Lombok");
}
```

This solution can be a safer alternative to using the `synchronized` keyword, because it allows you to lock on an instance field rather than on `this`.

Effortless Logging

Most logging requires some declaration to set up a logger within each class. This code is definitely repetitive boilerplate code. Lombok can take care of the logger declaration if you place the `@Log` annotation (or an annotation pertaining to your choice of logging API) on any class that requires logging capability.

For instance, if you wish to use a logging API—say, Log4j 2—each class that uses the logger must declare something similar to the following:

```
public class ClassName() {
   private static final org.apache.log4j.Logg
      org.apache.log4j.Logger.getLogger(Class
. . .

// Use log variable as needed }
```

Lombok makes it possible to do the following instead:

```
@Log4j2
public class ClassName() {
. . .
```

```
// Use log variable as needed
}
```

Listing 6 in the downloadable code shows an example using Log4j 2. The name of the logger will automatically be the same as its containing class's name. However, this can be customized by specifying the topic attribute of the respective logging annotation. For a complete listing of supported logging APIs, refer to the Lombok documentation and the Lombok Javadoc.

Other Useful Items

There are several other useful features Lombok offers that I haven't yet covered. Let's go through a couple of the most highly used.

Informal declaration. The `val` keyword can be used in place of an object type when you declare a local final variable, much like the `val` keyword that you have seen in alternative languages such as Groovy or Jython. Take the following code, for instance:

```
final ArrayList<Job> myJobs = new ArrayList<J
```

Using the `val` keyword, you can change the code to the following:

```
val myJobs = new ArrayList<Job>();
```

There are some considerations for using the `val` keyword. First, as mentioned previously, it marks the method declaration as `final`. Therefore, if you later need to change the value of the variable, using the `val` keyword is not possible. It also does not work correctly in some IDEs, so if you are trying to mark local variables as `final` in those IDEs, they are flagged as errors.

Be sneaky with exceptions. There are occasions where exception handling can become a burden, and I'd argue that this is typically the case when you are working with boilerplate exceptions. Most of the time, Java allows you to easily see where problems exist via the use of checked exceptions. However, in those cases where checked exceptions are burdensome, you can easily hide them using Lombok.

The `SneakyThrows` annotation can be placed on a method to essentially "swallow" the exceptions, allowing you to omit the `try-catch` block completely. The annotation allows a method to handle all exceptions quietly, or you can specify exactly which exceptions to ignore by passing the exception classes to the annotation as attributes. **Listing 7** in the downloadable code demonstrates the use of `@SneakyThrows` specifying which exceptions to swallow.

I want to reiterate that this Lombok feature should be used with caution, because it can become a real issue if too many exceptions are ignored.

Lazy getters. It is possible to indicate that a field should have a getter created once, and then the result should be cached for subsequent invocations. This can be useful if your getter method is expensive as far as performance goes. For instance, if you need to populate a list from a database query, or you need to access a web service to obtain the data for your field on the first access, it might make sense to cache the result for subsequent calls. To use this feature, a private final variable must be generated and initialized with the expensive expression. You can then annotate the field with `@Getter(lazy=true)` to implement this functionality.

IDE compatibility. Lombok plays well with the major IDEs, so simply including Lombok in your project and annotating accordingly typically does not generate errors in code or cause errors when the generated methods are called. In fact, in NetBeans the class `Navigator` is populated with the generated methods after annotations are placed and the code is saved, even though the methods do not appear in the code. Auto-completion works just as if the methods were typed into the class, even when generated properties are accessed from a web view in expression language.

Even more-concise Java EE. Over the past few years, Java EE has been making good headway on becoming a very productive and concise platform. Those of you who recall the laborious J2EE platform can certainly attest to the great number of improvements that have been made. I was very happy to learn that Lombok plays nicely with some Java EE APIs, such as Java Persistence API (JPA). This means it is very easy to develop constructs such as entity classes without writing all the boilerplate, which makes the classes much more concise and less error-prone. I've developed entire Java EE applications without any getters or setters in my entity classes, just by annotating them with `@Data`. I suggest you play around with it and see what works best for you.

Use caution and roll back. As with the use of any library, there are some caveats to keep in mind. This is especially true when you are thinking about future maintenance or modifications to the codebase. Lombok generates code for you, but that might cause a problem when it comes to refactoring. It is difficult to refactor code that does not exist until compile time, so be cautious with refactoring code that uses Lombok. You also need to think about readability. Lombok annotations might make troubleshooting a mystery for someone who is not familiar with the library—and even for those who are—if something such as `@SneakyThrows` is hiding an exception.

Fortunately, Lombok makes it easy to roll back if you need to. The delombok utility can be applied to your code to convert code

that uses Lombok back to vanilla Java. This utility can be used via Ant or the command line.

Conclusion

The Lombok library was created to make Java an easier language in which to code. It takes some of the most common boilerplate headaches out of the developer's task list. It can be useful for making your code more concise, reducing the chance for bugs, and speeding up development time. Try adding Lombok to one of your applications and see how many lines of code you can cut out.

Josh Juneau

Josh Juneau (@javajuneau) works as an application developer, system analyst, and database administrator. He primarily develops using Java and other JVM languages. He is a frequent contributor to Oracle Technology Network and *Java Magazine* and has written several books for Apress about Java and Java EE. Juneau was a JCP Expert Group member for JSR 372 and JSR 378. He is a member of the NetBeans Dream Team, a Java Champion, leader for the CJUG OSS Initiative, and a regular voice on the *JavaPubHouse Off Heap* podcast.

Share this Page