# Ссылки

<https://projectlombok.org/features>

<https://nuancesprog.ru/p/14675/>

# Что такое Lombok

Lombok — это библиотека Java, которая сокращает объем стандартного кода в классах. Вы добавляете аннотации, и Lombok генерирует код во время компиляции. Например, если аннотировать класс с помощью @Getter, то Lombok сгенерирует методы-геттеры для всех переменных в классе.

**Как работает Lombok**

Начнем с того, как происходит процесс компиляции. В нем есть три основных этапа:

* разбор и ввод;
* обработка аннотаций;
* анализ и генерация.

На этапе разбора и ввода исходные файлы считываются в синтаксическое дерево (AST), и каждое дерево передается на ввод. Все обработчики аннотаций вызываются на этапе обработки аннотаций. Если обработчики аннотаций генерируют новые исходные файлы или файлы классов, процесс компиляции переходит к первым шагам и все запускается заново. Так повторяется до тех пор, пока не появятся новые исходные файлы или файлы классов, созданные обработчиками аннотаций. AST преобразуется в файл класса на этапе анализа и генерации.

Вся магия Lombok происходит на этапе обработки аннотаций. Задача процессора аннотаций заключается в создании новых исходных файлов или классов, но вместо этого Lombok изменяет существующие. Нигде в спецификации компилятора Java не утверждается, могут или не могут обработчики аннотаций изменять существующий исходный файл. Lombok использует эту лазейку в своих интересах. Мы можем изменить класс, который будет сгенерирован из исходного кода, изменив AST. Вот как работает Lombok.

Преимущества Lombok

Lombok упрощает написание кода, позволяя сосредоточиться только на том, что нужно реализовать, а именно — переводе бизнес-требований в код. Написание геттеров, сеттеров, конструкторов, методов equals или реализация шаблона конструктора — это не то, на что мы должны тратить время.

Плохое применение Lombok

Иногда разработчики забывают, что Lombok генерирует код, потому что его не видно в исходном коде.

Прежде чем что-либо реализовать, многие добавляют аннотации Lombok в классы. Например, аннотируют классы DTO, используя аннотацию @Data для создания геттеров и сеттеров. Но они забывают (или не знают), что Lombok генерирует дополнительные методы, такие как equals, hashCode, toString и canEqual. Иногда они просто не нужны, а в репозитории не должно быть ничего, что мы не используем.

Кроме того, лучше не использовать какую-либо аннотацию Lombok, если она изменяет код. Если вы берете @Data, @Getter, @Setter или @AllArgsConstructor, то добавляете новый код в существующий без изменения написанного. Но если взять аннотацию @Utilityclass, она изменит код. Аннотация @Utilityclass преобразует существующий класс в утилитный, делая его окончательным и создавая приватный конструктор по умолчанию. Она также изменяет существующий метод и переменные, делая их статическими. Если вы проверите вышеуказанный класс, то не обнаружите никаких проблем. Нарушений Sonar также нет. Но при проверке скомпилированного кода вы увидите реальные проблемы. Переменная value является общедоступной статической конечной переменной, поэтому она должна соответствовать соглашению об именовании констант. Но когда мы проверяем исходный код, то видим, что это переменная экземпляра.

# @EqualsAndHashCode

Any class definition may be annotated with @EqualsAndHashCode to let lombok generate implementations of the equals(Object other) and hashCode() methods. By default, it'll use all non-static, non-transient fields, but you can modify which fields are used (and even specify that the output of various methods is to be used) by marking type members with @EqualsAndHashCode.Include or @EqualsAndHashCode.Exclude. Alternatively, you can specify exactly which fields or methods you wish to be used by marking them with @EqualsAndHashCode.Include and using @EqualsAndHashCode(onlyExplicitlyIncluded = true).

With Lombok

|  |
| --- |
| **import**lombok.EqualsAndHashCode;  @EqualsAndHashCode **public class**EqualsAndHashCodeExample {   **private transient int**transientVar = 10;   **private**String name;   **private double**score;   @EqualsAndHashCode.Exclude **private**Shape shape = **new**Square(5, 10);   **private**String[] tags;   @EqualsAndHashCode.Exclude **private int**id;      **public**String getName() {     **return this**.name;   }      @EqualsAndHashCode(callSuper=**true**)   **public static class**Square **extends**Shape {     **private final int**width, height;          **public**Square(**int**width, **int**height) {       **this**.width = width;       **this**.height = height;     }   } } |

Vanilla Java

|  |
| --- |
| **import**java.util.Arrays;  **public class**EqualsAndHashCodeExample {   **private transient int**transientVar = 10;   **private**String name;   **private double**score;   **private**Shape shape = **new**Square(5, 10);   **private**String[] tags;   **private int**id;      **public**String getName() {     **return this**.name;   }      @Override **public boolean**equals(Object o) {     **if**(o == **this**) **return true**;     **if**(!(o **instanceof**EqualsAndHashCodeExample)) **return false**;     EqualsAndHashCodeExample other = (EqualsAndHashCodeExample) o;     **if**(!other.canEqual((Object)**this**)) **return false**;     **if**(**this**.getName() == **null**? other.getName() != **null**: !**this**.getName().equals(other.getName())) **return false**;     **if**(Double.compare(**this**.score, other.score) != 0) **return false**;     **if**(!Arrays.deepEquals(**this**.tags, other.tags)) **return false**;     **return true**;   }      @Override **public int**hashCode() {     **final int**PRIME = 59;     **int**result = 1;     **final long**temp1 = Double.doubleToLongBits(**this**.score);     result = (result\*PRIME) + (**this**.name == **null**? 43 : **this**.name.hashCode());     result = (result\*PRIME) + (**int**)(temp1 ^ (temp1 >>> 32));     result = (result\*PRIME) + Arrays.deepHashCode(**this**.tags);     **return**result;   }      **protected boolean**canEqual(Object other) {     **return**other **instanceof**EqualsAndHashCodeExample;   }      **public static class**Square **extends**Shape {     **private final int**width, height;          **public**Square(**int**width, **int**height) {       **this**.width = width;       **this**.height = height;     }          @Override **public boolean**equals(Object o) {       **if**(o == **this**) **return true**;       **if**(!(o **instanceof**Square)) **return false**;       Square other = (Square) o;       **if**(!other.canEqual((Object)**this**)) **return false**;       **if**(!**super**.equals(o)) **return false**;       **if**(**this**.width != other.width) **return false**;       **if**(**this**.height != other.height) **return false**;       **return true**;     }          @Override **public int**hashCode() {       **final int**PRIME = 59;       **int**result = 1;       result = (result\*PRIME) + **super**.hashCode();       result = (result\*PRIME) + **this**.width;       result = (result\*PRIME) + **this**.height;       **return**result;     }          **protected boolean**canEqual(Object other) {       **return**other **instanceof**Square;     }   } } |

# @Getter and @Setter

You can annotate any field with @Getter and/or @Setter, to let lombok generate the default getter/setter automatically.  
A default getter simply returns the field, and is named getFoo if the field is called foo (or isFoo if the field's type is boolean). A default setter is named setFoo if the field is called foo, returns void, and takes 1 parameter of the same type as the field. It simply sets the field to this value.

The generated getter/setter method will be public unless you explicitly specify an AccessLevel, as shown in the example below. Legal access levels are PUBLIC, PROTECTED, PACKAGE, and PRIVATE.

You can also put a @Getter and/or @Setter annotation on a class. In that case, it's as if you annotate all the non-static fields in that class with the annotation.

You can always manually disable getter/setter generation for any field by using the special AccessLevel.NONE access level. This lets you override the behaviour of a @Getter, @Setter or @Data annotation on a class.

To put annotations on the generated method, you can use onMethod=@\_\_({@AnnotationsHere}); to put annotations on the only parameter of a generated setter method, you can use onParam=@\_\_({@AnnotationsHere}). Be careful though! This is an experimental feature. For more details see the documentation on the [onX](https://projectlombok.org/features/experimental/onX) feature.

# @With

The next best alternative to a setter for an immutable property is to construct a clone of the object, but with a new value for this one field. A method to generate this clone is precisely what @With generates: a withFieldName(newValue) method which produces a clone except for the new value for the associated field.

For example, if you create public class Point { private final int x, y; }, setters make no sense because the fields are final. @With can generate a withX(int newXValue) method for you which will return a new point with the supplied value for x and the same value for y.

The @With relies on a constructor for all fields in order to do its work. If this constructor does not exist, your @With annotation will result in a compile time error message. You can use Lombok's own [@AllArgsConstructor](https://projectlombok.org/features/constructor), or as [Value](https://projectlombok.org/features/Value) will automatically produce an all args constructor as well, you can use that too. It's of course also acceptable if you manually write this constructor. It must contain all non-static fields, in the same lexical order.

Like [@Setter](https://projectlombok.org/features/GetterSetter), you can specify an access level in case you want the generated with method to be something other than public:  
@With(level = AccessLevel.PROTECTED). Also like [@Setter](https://projectlombok.org/features/GetterSetter), you can also put a @With annotation on a type, which means a with method is generated for each field (even non-final fields).

# @NoArgsConstructor, @RequiredArgsConstructor, @AllArgsConstructor

@NoArgsConstructor will generate a constructor with no parameters.

@RequiredArgsConstructor generates a constructor with 1 parameter for each field that requires special handling. All non-initialized final fields get a parameter, as well as any fields that are marked as @NonNull that aren't initialized where they are declared. For those fields marked with @NonNull, an explicit null check is also generated.

@AllArgsConstructor generates a constructor with 1 parameter for each field in your class. Fields marked with @NonNull result in null checks on those parameters.

Each of these annotations allows an alternate form, where the generated constructor is always private, and an additional static factory method that wraps around the private constructor is generated. This mode is enabled by supplying the staticName value for the annotation, like so: @RequiredArgsConstructor(staticName="of"). Such a static factory method will infer generics, unlike a normal constructor.

@RequiredArgsConstructor(staticName = "of")  
@AllArgsConstructor(access = AccessLevel.PROTECTED)

# @NonNull

You can use @NonNull on a record component, or a parameter of a method or constructor. This will cause to lombok generate a null-check statement for you. A @NonNull on a primitive parameter results in a warning. No null-check will be generated.

With Lombok

|  |
| --- |
| **import**lombok.NonNull;  **public class**NonNullExample **extends**Something {   **private**String name;      **public**NonNullExample(@NonNull Person person) {     **super**("Hello");     **this**.name = person.getName();   } } |

Vanilla Java

|  |
| --- |
| **import**lombok.NonNull;  **public class**NonNullExample **extends**Something {   **private**String name;      **public**NonNullExample(@NonNull Person person) {     **super**("Hello");     **if**(person == **null**) {       **throw new**NullPointerException("person is marked non-null but is null");     }     **this**.name = person.getName();   } } |

# @Builder

<https://projectlombok.org/features/Builder>

The @Builder annotation produces complex builder APIs for your classes.

@Builder lets you automatically produce the code required to have your class be instantiable with code such as:

1. Person.builder()
2. .name("Adam Savage")
3. .city("San Francisco")
4. .job("Mythbusters")
5. .job("Unchained Reaction")
6. .build();

@Builder can be placed on a class, or on a constructor, or on a method.

A method annotated with @Builder (from now on called the *target*) causes the following 7 things to be generated:

* An inner static class named *Foo*Builder, with the same type arguments as the static method (called the *builder*).
* In the *builder*: One private non-static non-final field for each parameter of the *target*.
* In the *builder*: A package private no-args empty constructor.
* In the *builder*: A 'setter'-like method for each parameter of the *target*: It has the same type as that parameter and the same name. It returns the builder itself, so that the setter calls can be chained, as in the above example.
* In the *builder*: A build() method which calls the method, passing in each field. It returns the same type that the *target* returns.
* In the *builder*: A sensible toString() implementation.
* In the class containing the *target*: A builder() method, which creates a new instance of the *builder*.

The configurable aspects of builder are:

* The *builder's class name* (default: return type + 'Builder')
* The *build()* method's name (default: "build")
* The *builder()* method's name (default: "builder")
* If you want toBuilder() (default: no)
* The access level of all generated elements (default: public).
* (discouraged) If you want your builder's 'set' methods to have a prefix, i.e. Person.builder().setName("Jane").build() instead of Person.builder().name("Jane").build() and what it should be.

Example usage where all options are changed from their defaults:  
@Builder(builderClassName = "HelloWorldBuilder", buildMethodName = "execute", builderMethodName = "helloWorld", toBuilder = true, access = AccessLevel.PRIVATE, setterPrefix = "set")

## @Builder.Default

If a certain field/parameter is never set during a build session, then it always gets 0 / null / false. If you've put @Builder on a class (and not a method or constructor) you can instead specify the default directly on the field, and annotate the field with @Builder.Default:  
@Builder.Default private final long created = System.currentTimeMillis();

## @Singular

By annotating one of the parameters (if annotating a method or constructor with @Builder) or fields (if annotating a class with @Builder) with the @Singular annotation, lombok will treat that builder node as a collection, and it generates 2 'adder' methods instead of a 'setter' method. One which adds a single element to the collection, and one which adds all elements of another collection to the collection. No setter to just set the collection (replacing whatever was already added) will be generated. A 'clear' method is also generated.

# @Data

@Data is a convenient shortcut annotation that bundles the features of [@ToString](https://projectlombok.org/features/ToString), [@EqualsAndHashCode](https://projectlombok.org/features/EqualsAndHashCode), [@Getter / @Setter](https://projectlombok.org/features/GetterSetter) and [@RequiredArgsConstructor](https://projectlombok.org/features/constructor) together.

In other words, @Data generates all the boilerplate that is normally associated with simple POJOs (Plain Old Java Objects) and beans: getters for all fields, setters for all non-final fields, and appropriate toString, equals and hashCode implementations that involve the fields of the class, and a constructor that initializes all final fields, as well as all non-final fields with no initializer that have been marked with @NonNull, in order to ensure the field is never null.

All generated getters and setters will be public. To override the access level, annotate the field or class with an explicit @Setter and/or @Getter annotation.

All fields marked as transient will not be considered for hashCode and equals. All static fields will be skipped entirely (not considered for any of the generated methods, and no setter/getter will be made for them).

@Data can handle generics parameters for fields just fine. In order to reduce the boilerplate when constructing objects for classes with generics, you can use the staticConstructor parameter to generate a private constructor, as well as a static method that returns a new instance.

# @Value

@Value is the immutable variant of [@Data](https://projectlombok.org/features/Data); all fields are made private and final by default, and setters are not generated. The class itself is also made final by default.

Like @Data, useful toString(), equals() and hashCode() methods are also generated, each field gets a getter method, and a constructor that covers every argument (except final fields that are initialized in the field declaration) is also generated.

In practice, @Value is shorthand for: final @ToString @EqualsAndHashCode @AllArgsConstructor @FieldDefaults(makeFinal = true, level = AccessLevel.PRIVATE) @Getter, except that explicitly including an implementation of any of the relevant methods simply means that part won't be generated and no warning will be emitted.

It is possible to override the final-by-default and private-by-default behavior using either an explicit access level on a field, or by using the @NonFinal or @PackagePrivate annotations. @NonFinal can also be used on a class to remove the final keyword.

# val

You can use val as the type of a local variable declaration instead of actually writing the type. When you do this, the type will be inferred from the initializer expression. The local variable will also be made final. This feature works on local variables and on foreach loops only, not on fields. The initializer expression is required.

With Lombok

|  |
| --- |
| **import**java.util.ArrayList; **import**java.util.HashMap; **import**lombok.val;  **public class**ValExample {   **public**String example() {     val example = **new**ArrayList<String>();     example.add("Hello, World!");     val foo = example.get(0);     **return**foo.toLowerCase();   }      **public void**example2() {     val map = **new**HashMap<Integer, String>();     map.put(0, "zero");     map.put(5, "five");     **for**(val entry : map.entrySet()) {       System.out.printf("%d: %s\n", entry.getKey(), entry.getValue());     }   } } |

Vanilla Java

|  |
| --- |
| **import**java.util.ArrayList; **import**java.util.HashMap; **import**java.util.Map;  **public class**ValExample {   **public**String example() {     **final**ArrayList<String> example = **new**ArrayList<String>();     example.add("Hello, World!");     **final**String foo = example.get(0);     **return**foo.toLowerCase();   }      **public void**example2() {     **final**HashMap<Integer, String> map = **new**HashMap<Integer, String>();     map.put(0, "zero");     map.put(5, "five");     **for**(**final**Map.Entry<Integer, String> entry : map.entrySet()) {       System.out.printf("%d: %s\n", entry.getKey(), entry.getValue());     }   } } |

# var

var works exactly like [val](https://projectlombok.org/features/val), except the local variable is *not* marked as final.

The type is still entirely derived from the mandatory initializer expression, and any further assignments, while now legal (because the variable is no longer final), aren't looked at to determine the appropriate type.  
For example, var x = "Hello"; x = Color.RED; does *not* work; the type of x will be inferred to be java.lang.String and thus, the x = Color.RED assignment will fail. If the type of x was inferred to be java.lang.Object this code would have compiled, but that's not howvar works.

# @Log

You put the variant of @Log on your class (whichever one applies to the logging system you use); you then have a static final log field, initialized as is the commonly prescribed way for the logging framework you use, which you can then use to write log statements.

There are several choices available:

**@CommonsLog**

Creates private static final [org.apache.commons.logging.Log](https://commons.apache.org/logging/apidocs/org/apache/commons/logging/Log.html) log = [org.apache.commons.logging.LogFactory.getLog](https://commons.apache.org/logging/apidocs/org/apache/commons/logging/LogFactory.html#getLog(java.lang.Class))(LogExample.class);

**@Flogger**

Creates private static final [com.google.common.flogger.FluentLogger](https://google.github.io/flogger/) log = com.google.common.flogger.FluentLogger.forEnclosingClass();

**@JBossLog**

Creates private static final [org.jboss.logging.Logger](https://docs.jboss.org/jbosslogging/latest/org/jboss/logging/Logger.html) log = [org.jboss.logging.Logger.getLogger](https://docs.jboss.org/jbosslogging/latest/org/jboss/logging/Logger.html#getLogger(java.lang.Class))(LogExample.class);

**@Log**

Creates private static final [java.util.logging.Logger](https://docs.oracle.com/javase/6/docs/api/java/util/logging/Logger.html) log = [java.util.logging.Logger.getLogger](https://docs.oracle.com/javase/6/docs/api/java/util/logging/Logger.html#getLogger(java.lang.String))(LogExample.class.getName());

**@Log4j**

Creates private static final [org.apache.log4j.Logger](https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Logger.html) log = [org.apache.log4j.Logger.getLogger](https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Logger.html#getLogger(java.lang.Class))(LogExample.class);

**@Log4j2**

Creates private static final [org.apache.logging.log4j.Logger](https://logging.apache.org/log4j/2.0/log4j-api/apidocs/org/apache/logging/log4j/Logger.html) log = [org.apache.logging.log4j.LogManager.getLogger](https://logging.apache.org/log4j/2.0/log4j-api/apidocs/org/apache/logging/log4j/LogManager.html#getLogger(java.lang.Class))(LogExample.class);

**@Slf4j**

Creates private static final [org.slf4j.Logger](https://www.slf4j.org/api/org/slf4j/Logger.html) log = [org.slf4j.LoggerFactory.getLogger](https://www.slf4j.org/api/org/slf4j/LoggerFactory.html#getLogger(java.lang.Class))(LogExample.class);

**@XSlf4j**

Creates private static final [org.slf4j.ext.XLogger](https://www.slf4j.org/api/org/slf4j/ext/XLogger.html) log = [org.slf4j.ext.XLoggerFactory.getXLogger](https://www.slf4j.org/api/org/slf4j/ext/XLoggerFactory.html#getXLogger(java.lang.Class))(LogExample.class);

**@CustomLog**

Creates private static final *com.foo.your.Logger* log = *com.foo.your.LoggerFactory.createYourLogger*(LogExample.class);

This option *requires* that you add a configuration to your [lombok.config](https://projectlombok.org/features/configuration) file to specify what @CustomLog should do.

For example:lombok.log.custom.declaration = com.foo.your.Logger com.foo.your.LoggerFactory.createYourLog(TYPE)(TOPIC) which would produce the above statement. First comes a type which is the type of your logger, then a space, then the type of your logger factory, then a dot, then the name of the logger factory method, and then 1 or 2 parameter definitions; at most one definition with TOPIC and at most one without TOPIC. Each parameter definition is specified as a parenthesised comma-separated list of parameter kinds. The options are: TYPE (passes this @Log decorated type, as a class), NAME (passes this @Log decorated type's fully qualified name), TOPIC (passes the explicitly chosen topic string set on the @CustomLog annotation), and NULL (passes null).

The logger type is optional; if it is omitted, the logger factory type is used. (So, if your logger class has a static method that creates loggers, you can shorten your logger definition).

Please contact us if there is a public, open source, somewhat commonly used logging framework that we don't yet have an explicit annotation for. The primary purpose of @CustomLog is to support your in-house, private logging frameworks.

By default, the topic (or name) of the logger will be the (name of) the class annotated with the @Log annotation. This can be customised by specifying the topic parameter. For example: @XSlf4j(topic="reporting").

With Lombok

|  |
| --- |
| **import**lombok.extern.java.Log; **import**lombok.extern.slf4j.Slf4j;  @Log **public class**LogExample {      **public static void**main(String... args) {     log.severe("Something's wrong here");   } }  @Slf4j **public class**LogExampleOther {      **public static void**main(String... args) {     log.error("Something else is wrong here");   } }  @CommonsLog(topic="CounterLog") **public class**LogExampleCategory {    **public static void**main(String... args) {     log.error("Calling the 'CounterLog' with a message");   } } |

Vanilla Java

|  |
| --- |
| **public class**LogExample {   **private static final**java.util.logging.Logger log = java.util.logging.Logger.getLogger(LogExample.**class**.getName());      **public static void**main(String... args) {     log.severe("Something's wrong here");   } }  **public class**LogExampleOther {   **private static final**org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(LogExampleOther.**class**);      **public static void**main(String... args) {     log.error("Something else is wrong here");   } }  **public class**LogExampleCategory {   **private static final**org.apache.commons.logging.Log log = org.apache.commons.logging.LogFactory.getLog("CounterLog");    **public static void**main(String... args) {     log.error("Calling the 'CounterLog' with a message");   } } |

# @Synchronized

@Synchronized is a safer variant of the synchronized method modifier. Like synchronized, the annotation can be used on static and instance methods only. It operates similarly to the synchronized keyword, but it locks on different objects. The keyword locks on this, but the annotation locks on a field named $lock, which is private.  
If the field does not exist, it is created for you. If you annotate a static method, the annotation locks on a static field named $LOCK instead.

If you want, you can create these locks yourself. The $lock and $LOCK fields will of course not be generated if you already created them yourself. You can also choose to lock on another field, by specifying it as parameter to the @Synchronized annotation. In this usage variant, the fields will not be created automatically, and you must explicitly create them yourself, or an error will be emitted.

# @FieldDefaults

The @FieldDefaults annotation can add an access modifier (public, private, or protected) to each field in the annotated class or enum. It can also add final to each field in the annotated class or enum.

To add final to each (instance) field, use @FieldDefaults(makeFinal=true). Any non-final field which must remain nonfinal can be annotated with @NonFinal (also in the lombok.experimental package).

To add an access modifier to each (instance) field, use @FieldDefaults(level=AccessLevel.PRIVATE). Any field that does not already have an access modifier (i.e. any field that looks like package private access) is changed to have the appropriate access modifier. Any package private field which must remain package private can be annotated with @PackagePrivate (also in the lombok.experimental package).

With Lombok

|  |
| --- |
| **import**lombok.AccessLevel; **import**lombok.experimental.FieldDefaults; **import**lombok.experimental.NonFinal; **import**lombok.experimental.PackagePrivate;  @FieldDefaults(makeFinal=true, level=AccessLevel.PRIVATE) **public class**FieldDefaultsExample {   **public final int**a;   **int**b;   @NonFinal **int**c;   @PackagePrivate **int**d;      FieldDefaultsExample() {     a = 0;     b = 0;     d = 0;   } } |

Vanilla Java

|  |
| --- |
| **public class**FieldDefaultsExample {   **public final int**a;   **private final int**b;   **private int**c;   **final int**d;      FieldDefaultsExample() {     a = 0;     b = 0;     d = 0;   } } |