**Lombook**

Широко распространено убеждение, что одной из главных проблем Java является то, что для написания “хорошего” (идиома) кода необходимо часто использовать шаблонный код. Спросите об этом любого человека работающего с Java. Наиболее очевидно это в [паттерне Builder](http://www.informit.com/articles/article.aspx?p=1216151&seqNum=2) (далее “построитель”). В этой статье я представлю вам два решения и обсужу их преимущества и подводные камни.

Предисловие к паттерну построителя

Паттерн построителя хорошо подходит для классов, которые могут иметь сложную инициализацию. Обычно это заключается в в следующем: задаётся несколько переменных, часть из которых будет использована, и часть будет опциональной. Использование [заводских статичных методов](http://www.informit.com/articles/article.aspx?p=1216151&seqNum=1) или конструкторов может привести к **телескопическому паттерну конструктора**, описанному [Joshua Block](https://twitter.com/joshbloch) (как сказано в его твиттере: “эффективный автор Java, API дизайнер, отличный парень”) следующим образом:

…в котором вы представляете конструктор только с обязательными параметрами, другой с одним опциональными параметром, третий с двумя опциональными параметрами и так далее. Кульминацией будет конструктор с только опциональными параметрами.

Даже при наличии только пяти обязательных параметров, все из которых одного типа, разработчик всё равно может легко допустить ошибку в порядке задачи параметров. Это часто может стать причиной для появления сложных для обнаружения багов.

Паттерн построителя предлагает элегантное решение для всех этих проблем. Более того, этот паттерн может помочь разработчику быть уверенным в том, что все предварительные условия объекта будут удовлетворены **перед**постройкой объекта. Это гарантирует, что построенный объект будет в правильном состоянии.

Но в сторону все эти радужные преимущества, на самом деле написание построителя может быть достаточно утомительной и повторяющейся задачей. “И что?, — скажете вы.— По крайней мере я получаю все эти преимущества!”. Так-то оно так… но вернитесь и скажите мне это после того, как напишете свой **десятый**построитель в одном и том же проекте.

Поэтому давайте рассмотрим другие опции.

Сделать это динамически: Lombok

[Lombok](https://projectlombok.org/) — это инструмент/библиотека, хорошо известная среди Java-разработчиков (хотя всё ещё недостаточно, имхо). Как говорится на сайте, “это библиотека Java, которая автоматически подключается к вашему редактору и строит инструменты, подпитывая вашу Java.” [Lombok добавляет настолько много функций через динамическую генерацию кода](https://projectlombok.org/features/all), что они просто не могут быть полностью покрыты здесь.

[@Builder](https://projectlombok.org/features/Builder) аннотация используется для автоматической генерации всех возможных конструкторов в любом POJO¹. Для использования можно поместить её в класс, конструктор или статичный метод ([посмотрите документацию](https://projectlombok.org/features/Builder) для деталей). Просто поместите аннотацию в класс Person:

|  |
| --- |
| @Builder |
|  | public class Person { |
|  |  |
|  | private String firstName; |
|  |  |
|  | private String middleName; |
|  |  |
|  | private String lastName; |
|  |  |
|  | private String city; |
|  |  |
|  | private String jobTitle; |
|  |  |
|  | } |

Далее, с помощью магии Lombok, класс PersonBuilder будет автоматически и мгновенно сгенерирован. Теперь мы можем создать экземпляр объекта Person используя шаблон построителя:

PersonBuilder bildr = Person.**builder()**.firstName("Adam").lastName("Savage")  
 .city("San Francisco").jobTitle("TV Personality").**build()**;

или

PersonBuilder bildr = Person.**builder()**;

bildr.firstName("John").lastName("Smith").city("Albuquerque");

return bildr.**build()**;

Кроме того, поведение аннотации @Builder можно настраивать:

* Параметр ***builderClassName*** задаёт имя внутреннего класса (по умолчанию «конструируемый ***тип+Builder***»).
* Параметр ***buildMethodName*** задаёт имя метода, создающего ваш класс (по умолчанию ***build()***)
* Параметр ***builderMethodName*** задаёт имя статического метода, возвращающего Builder (по умолчанию ***builder()***)
* Параметр **toBuilder=true** генерирует метод toBuilder(), который создаёт объект билдера, заполненный всеми значениями этого экземпляра. По умолчанию такой метод не создаётся.

Размещение аннотации @Builder в классе Person

1. создаёт класс PersonBuilder с [плавными сеттерами](https://en.wikipedia.org/wiki/Fluent_interface) для всей свойств класса Person и метода build() для создания объекта Person с использованием заданных значений, а также
2. добавляет метод public static builder() классу Person, который возвращает новый экземпляр PersonBuilder.

Размещение аннотации @Builder в конструкторе или статичном методе делает всё вышеописанное, но генерирует сеттеры только для перечисленных в конструкторе/статичном методе параметрах.

Добавление аннотации @NonNull в поле класса Person делает это **обязательным** параметром. Не устанавливая его значение при использовании соответствующего сеттер-метода PersonBuilder, будет бросать NullPointerException при вызове метода build().

Прелесть Lombok в том, что ничего из генерированного кода не будет видно в файле-источнике (хотя Eclipse покажет покажет сгенерированные классы, методы и поля в виде Outline), что оставляет его чистым и аккуратным. Разработчику нужно будет сконцентрироваться только на важных частях POJO, а не на утомительной, практической и скучной (зачастую) реализации. Сделайте любимые изменения в POJO и Lombok мгновенно обновит/сгенерирует релевантный код.

**Еще один пример**

@Builder(toBuilder = true)

public class User {

@NonNull

UUID userId;

@NonNull

String email;

@Singular

Set<String> favoriteFoods;

@NonNull

@Builder.Default

String avatar = “default.png”;

}

User user = User.builder()

.userId(UUID.random())

.email(“grubhub@grubhub.com”)

.favoriteFood(“burritos”)

.favoriteFood(“dosas”)

.build()

Обратите внимание, что поле аватара аннотировано @NonNull, но не задано. Дело в том, что аннотация @Builder.Default по умолчанию указывает на default.png.

Также обратите внимание, как билдер использует favoriteFood, единственное название свойства в нашем объекте. При размещении аннотации @Singular на свойстве коллекции Lombok создаёт специальные методы билдера для индивидуального добавления элементов в коллекцию, а не для одновременного добавления всей коллекции. Это особенно хорошо для тестов, потому что способы создания маленьких коллекций в Java нельзя назвать простыми и быстрыми.

Наконец, параметр ***toBuilder*** = true добавляет метод экземпляра toBuilder(), который создаёт объект билдера, заполненный всеми значениями этого экземпляра. Так легко создаётся новый инстанс, предварительно заполненный всеми значениями из исходного, так что остаётся изменить лишь необходимые поля. Это особенно полезно для классов @Value, поскольку поля неизменяемы.

Несколько примечаний дополнительно настраивают специальные функции сеттера: @Wither - как сеттер, только для неизменяемых классов(*в которых все переменные* ***final***), создаёт методы withX для каждого свойства. На входе — значение, на выходе — клон экземпляра с обновлённым значением одного поля.

@Accessors - позволяет настраивать автоматически созданные сеттеры. Параметр fluent=true отключает конвенцию “get” и “set” для геттеров и сеттеров. В определённых ситуациях это может быть полезной заменой @Builder.

Для ограничения видимости:

- конструкторов используется атрибут access

- геттеров/сеттеров  используется атрибут  value

@Entity

@Builder(toBuilder = true)

@AllArgsConstructor(access = AccessLevel.PACKAGE)

@NoArgsConstructor(access = AccessLevel.PACKAGE)

@Setter(value = AccessLevel.PACKAGE)

@Getter

public static class Person {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

---------------

}

Для работы таких аннотаций как @NonNull, @NotBlank, @NotEmpty необходимо в *pom.xml* добавить зависимость:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

@EqualsAndHashCode - легкое создание методов Equals и HashCode.

Не подходит для сущностей с автоматически генерируемыми БД полей “*id”* .

Подходит для сущностей с UUID id.

*@EqualsAndHashCode(of = {"id", "name", "price"})*

@ToString - для реализации стандартного toString метода, но не подходит для сущностей с лениво загруженными частями объекта:

*@ToString(of = {"id", "name", "price"})*

@Data

@Data

public class User {

private UUID userId;

private String email;

}

Не применять аннотацию @Data к сущностям (“entity”) у которых:

- “id” назначает БД, т.к. поломает встроенную в @Data логику формирования [@EqualsAndHashCode](https://projectlombok.org/features/EqualsAndHashCode) и выдаст ошибку.

- лениво загруженны части объекта, т.к. вызовет проблемы с методом [@ToString](https://projectlombok.org/features/ToString).

@Data — аннотация, которая применяет сразу несколько аннотаций Lombok:

* [@EqualsAndHashCode](https://projectlombok.org/features/EqualsAndHashCode) генерирует реализации equals и hashCode, которые по умолчанию используют нестатические и нестационарные поля, но настраиваются.
* [@ToString](https://projectlombok.org/features/ToString) генерирует реализацию для метода toString(), которая состоит из аккуратного представления объекта: имя класса, все поля и их значения.
* [@Getter / @Setter](https://projectlombok.org/features/GetterSetter) генерирует геттеры и сеттеры для частных полей. Аннотация @Data делает сгенерированные сеттеры public, применение @Setter(value = AccessLevel.PROTECTED) к полям делает их protected.
* [@RequiredArgsConstructor](https://projectlombok.org/features/constructor) создаёт конструктор с требуемыми аргументами, где обязательными являются *final* поля и поля с аннотацией @NonNull (подробнее об этом ниже).

Одна эта аннотация просто и элегантно охватывает многие типичные случаи использования. Но POJO не всегда покрывает необходимую функциональность. @Data — полностью изменяемый класс, злоупотребление которым может повысить сложность и ограничить параллелизм, что негативно отражается на живучести приложения.

@Value

Это неизменяемый вариант @Data, все поля по умолчанию являются *privat* и *final*, а сеттеры не создаются. Благодаря этому объекты @Value сразу становятся неизменяемыми. Поскольку все поля являются окончательными, конструктора аргументов нет. Вместо этого Lombok использует @AllArgsConstructor. В результате получается полностью функциональный, неизменяемый объект.

@Value

@Builder(toBuilder = true)

public class User {

@NonNull

UUID userId;

….

}

Вы можете пометить любой конструктор или метод аннотацией @lombok.experimental.Tolerate, чтобы скрыть их от Lombok.

Можно переопределить final-по-умолчанию и private-по-умолчанию с помощью явного указания уровня доступа для поля или с помощью аннотаций @NonFinal или @PackagePrivate.

Можно переопределить любое поведение любой «части» @Value с помощью явного указания соответствующих аннотаций.

Хотя в Java 9 появилось ключевое слово **var**(тип переменной java определяет сама – применяется при создании объектов классов через ***new***), но переменную всё равно можно переназначить. В Lombok есть ключевое слово val, которое выводит окончательный тип локальной переменной.

val map = new HashMap<Integer, String>();

*аналог*

final Map map = new HashMap<Integer, String>();

Некоторые классы c чисто статическими функциями не предназначены для инициализации. Один из способов предотвратить создание экземпляра — объявить приватный конструктор, который выбрасывает исключение. Lombok кодифицировал этот шаблон в аннотации @UtilityClass. Она генерирует приватный конструктор, который создаёт исключение, окончательно выводит класс и делает все методы статическими.  
  
@UtilityClass

*// will be made final*

public class UtilityClass {

*// will be made static*

private final int GRUBHUB = “ GRUBHUB”;

*// autogenerated by Lombok* *// private UtilityClass() {*

*// throw new java.lang.UnsupportedOperationException("This is a utility class and cannot be instantiated");*

*//}*

*// will be made static*

public void append(String input) {

return input + GRUBHUB;

}

}

Java часто критикуют за многословность из-за проверяемых исключений.

Отдельная аннотация Lombok устраняет их: @SneakyThrows. Как и следовало ожидать, реализация довольно хитрая. Она не перехватывает исключения и даже не оборачивает исключения в RuntimeException. Вместо этого она полагается на тот факт, что во время выполнения JVM не проверяет согласованность проверяемых исключений. Так делает только javac. Поэтому Lombok с помощью преобразования байт-кода во время компиляции отключает эту проверку. В результате получается запускаемый код.

public class SneakyThrows {

@SneakyThrows public void sneakyThrow() {

throw new Exception();

}

}

В плагине IDE есть функция “de-lombok”, которая приблизительно преобразует большинство аннотаций Lombok в нативный Java-код (аннотации @NonNull не конвертируются). Таким образом, любая IDE с установленным плагином сможет конвертировать большинство аннотаций в собственный код Java и обратно.

Ведение журнала — базовое требование для всех частей программы. Любой класс, выполняющий значимую работу, должен записывать лог. Таким образом, стандартный логгер становится шаблоном для каждого класса. Lombok упрощает этот шаблон до одной аннотации @Log для util.logging, так же позволяет выбрать вид логгера: @Slf4j для Logback и Slf4j, @CommonsLog, @Flogger, @JBossLog, @Log4j, @Log4j2или @XSlf4j, которая автоматически определяет и создаёт экземпляр логгера с правильным именем класса. Существует несколько различных аннотаций в зависимости от структуры журнала.

@Slf4j

public class Example {

public static void main(String... args) {

log.error("error");

}

или  
  
@Slf4j *// also: @CommonsLog @Flogger @JBossLog @Log @Log4j @Log4j2 @XSlf4j*

public class UserService {

*// created automatically*

*// private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(UserService.class);*

}

После объявления логгера добавляем наши зависимости:  
  
@Slf4j

@RequiredArgsConstructor

@FieldDefaults(makeFinal=true, level=AccessLevel.PRIVATE)

public class UserService {

@NonNull UserDao userDao;

}

@RequiredArgsConstructor создаёт конструктор, который устанавливает экземпляр UserDao.

Аннотация @FieldDefaults добавляет ко всем полям ***final****(*makeFinal=true*)* и ***privat****(*level=AccessLevel.PRIVATE*)* модификаторы.

@Cleanup - простое определение ресурсов, так чтобы они автоматически закрывались  
после окончания работы кода. (не так актуально при использовании **try with resources** )

@Cleanup InputStream in = new FileInputStream(args[0]);

@Cleanup OutputStream out = new FileOutputStream(args[1]);

Недостатки Lombok

Не смотря на всю свою магию, Lombok не идеален.

Один из наибольших недостатков генерации построителей через Lombok заключается в том, что если вам в вашем построителе необходима валидация более сложная, чем простая проверка на заполненность, то здесь его магия заканчивается. Вы можете сделать [delombok](https://projectlombok.org/features/delombok), скопировать весь код и модифицировать его вручную, но это утомительно и убирает всё удобство от использования Lombok.

Недостаток, хотя не все с этим согласятся, в том, что проверка на заполненность обязательного значения и бросок NullPointerException делается **в конструкторе** POJO, а не **до**построения POJO. Я предпочитаю не строить POJO совсем, если его предварительные условия не удовлетворительны.

Более общая проблема с Lombok заключается в том, что он не очень хорошо работает с инструментами рефакторинга в IDE. Рассмотрим следующий (достаточно глупый) код:

|  |
| --- |
| public final class CreateAPerson { |
|  |  |
|  | private CreateAPerson() {} |
|  |  |
|  | public static Person doIt() { |
|  | // Class name PersonBuilder here can't be updated automatically by IDE refactoring |
|  | PersonBuilder bildr = Person.builder(); |
|  | bildr.firstName("John").lastName("Smith").city("Albuquerque"); |
|  | return bildr.build(); |
|  | } |
|  | } |

Если бы мы переименовали класс Person в HumanBeing, то соответствующего сгенерированное имя построителя станет HumanBeingBuilder. Но инструменты рефакторинга IDE не смогут обновить PersonBuilder в выше написанном коде до HumanBeingBuilder(по крайней мере в Eclipse :-)).

Тем не менее, существует простой способ решения этой проблемы. Просто используйте опцию builderClassName аннотации Builder, чтобы установить фиксированное имя для класса построителя: @Builder(builderClassName = "Builder"). Тогда все отсылки на класс построителя станут Person.Builder, которые в свою очередь будут правильно изменены путем реорганизации в HumanBeing.Builder.

[IDEA JetBrains IntelliJ](https://www.jetbrains.com/idea/) редактор совместим с Ломбок.

Добавьте [плагин Lombok IntelliJ,](https://plugins.jetbrains.com/plugin/6317) чтобы добавить поддержку lombok для IntelliJ:

Вы также можете проверить [Настройка Lombok с Eclipse и IntelliJ](https://www.baeldung.com/lombok-ide) , статьей в блоге о baeldung.

Maven

To set up lombok with any build tool, you have to specify that the lombok dependency is required to compile your source code, but does not need to be present when running/testing/jarring/otherwise deploying your code. Generally this is called a 'provided' dependency. This page explains how to integrate lombok with the [Apache Maven](https://maven.apache.org/) build tool.

Lombok is available in maven central, so telling Maven to download lombok is easy.

Adding lombok to your pom file

To include lombok as a 'provided' dependency, add it to your <dependencies> block like so:

<dependencies>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.12</version>

<scope>provided</scope>

</dependency>

</dependencies>

JDK9+ with module-info.java

The configuration of the compiler plug-in should contain the following:

<annotationProcessorPaths>

<path>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.12</version>

</path>

</annotationProcessorPaths>

**Поддерживаемые ключи конфигурации**

|  |  |
| --- | --- |
| 1 | lombok.accessors.chain = [true | false] (default: false) |

Если установлен в true, то сгенерированные методы установки значения возвращают this (вместо void). Явно указанный параметр chain в аннотации @Accessors имеет приоритет перед этой настройкой.

|  |  |
| --- | --- |
| 1 | lombok.accessors.fluent = [true | false] (default: false) |

Если установлено в true, то генерирует методы установки и получения значений без стандартных префиксов get и set, а вместо этого методы используют то же имя, что и поле. Явно указанный параметр fluent в @Accessors имеет приоритет над этой настройкой.

|  |  |
| --- | --- |
| 1 | lombok.accessors.prefix += a field prefix (default: empty list) |

Это свойство-список. Значения могут быть добавлены с помощью оператора +=. Унаследованные префиксы из родительского файла конфигурации могут быть удалены с помощью оператора -=. Lombok разделяет любые подходящие префиксы от имени, для того чтобы определить имя сгенерированных методов получения/установки значений. Например, если m — это один из префиксов из этой настройки, то поле с именем mFoobar будет иметь метод установки значения getFoobar(), а не getMFoobar(). Явно указанный параметр prefix в @Accessors имеет приоритет перед этой настройкой.

|  |  |
| --- | --- |
| 1 | lombok.getter.noIsPrefix = [true | false] (default: false) |

Если установлено в true, то сгенерированные методы получения значений для логических полей будут использовать префикс get вместо префикса по умолчанию, и любой сгенерированный код, вызывающий методы получения значения, вроде @ToString, будет также использовать get вместо is.

|  |  |
| --- | --- |
| 1 | lombok.setter.flagUsage = [warning | error] (default: not set) |

Lombok будет помечать любое использование @Setter как предупреждение или ошибку, если настроено.

|  |  |
| --- | --- |
| 1 | lombok.getter.flagUsage = [warning | error] (default: not set) |

Lombok будет помечать любое использование @Getter как предупреждение или ошибку, если настроено.

**Мелким шрифтом**

Для генерации имён методов первый символ поля, если он в нижнем регистре, приводится к верхнему регистру, в противном случае он оставляется неизменным. Затем используются префиксы get/set/is.

Никакой метод не генерируется, если уже существует метод с таким же именем (не зависит от регистра) и тем же количеством параметров. Например, метод getFoo() не будет генерироваться, если уже существует метод getFoo(String… x), даже если технически возможно сделать такой метод. Это предостережение, для того чтобы предотвратить недоразумения. Если генерация метода пропущена по этой причине, то появляется предупреждение.

Вы можете пометить любой метод @lombok.experimental.Tolerate, чтобы скрыть их.

Для логических полей, начинающихся с is с последующим символов в верхнем регистре, префикс не используется для генерации имени метода получения значения.

Любые вариации логического типа не будут иметь префикс is вместо get. Например, возвращаемый тип java.lang.Boolean будет иметь префикс get, а не is.

Любые аннотации с именем @NonNull (не зависит от регистра) для поля интерпретируются как то, что поле никогда не может содержать null. По этой причине эти аннотации добавляют явную проверку на null в сгенерированных методах установки значения. Также эти аннотации (как и любые аннотации с именем @Nullable или @CheckForNull) копируются к параметру метода установки значения и к методу получения значения.

Вы можете добавить аннотацию @Getter или @Setter к классу. Это эквивалентно добавлению этих аннотаций ко всем нестатическим полям класса. Аннотации @Getter/@Setter для полей имеют приоритет над аннотациями для классов.

Использование AccessLevel.NONE просто ничего не генерирует. Оно полезно только в комбинации с @Data для класса или для установленных для класса аннотаций @Getter или @Setter.

@Getter может также использоваться в перечислениях. @Setter — не может, но не по технической причине. Методы установки значений для перечислений — это очень экстремальная и плохая идея.