

# Compile-Time Annotation Processing

**GRUBHUB**<sup>TM</sup>



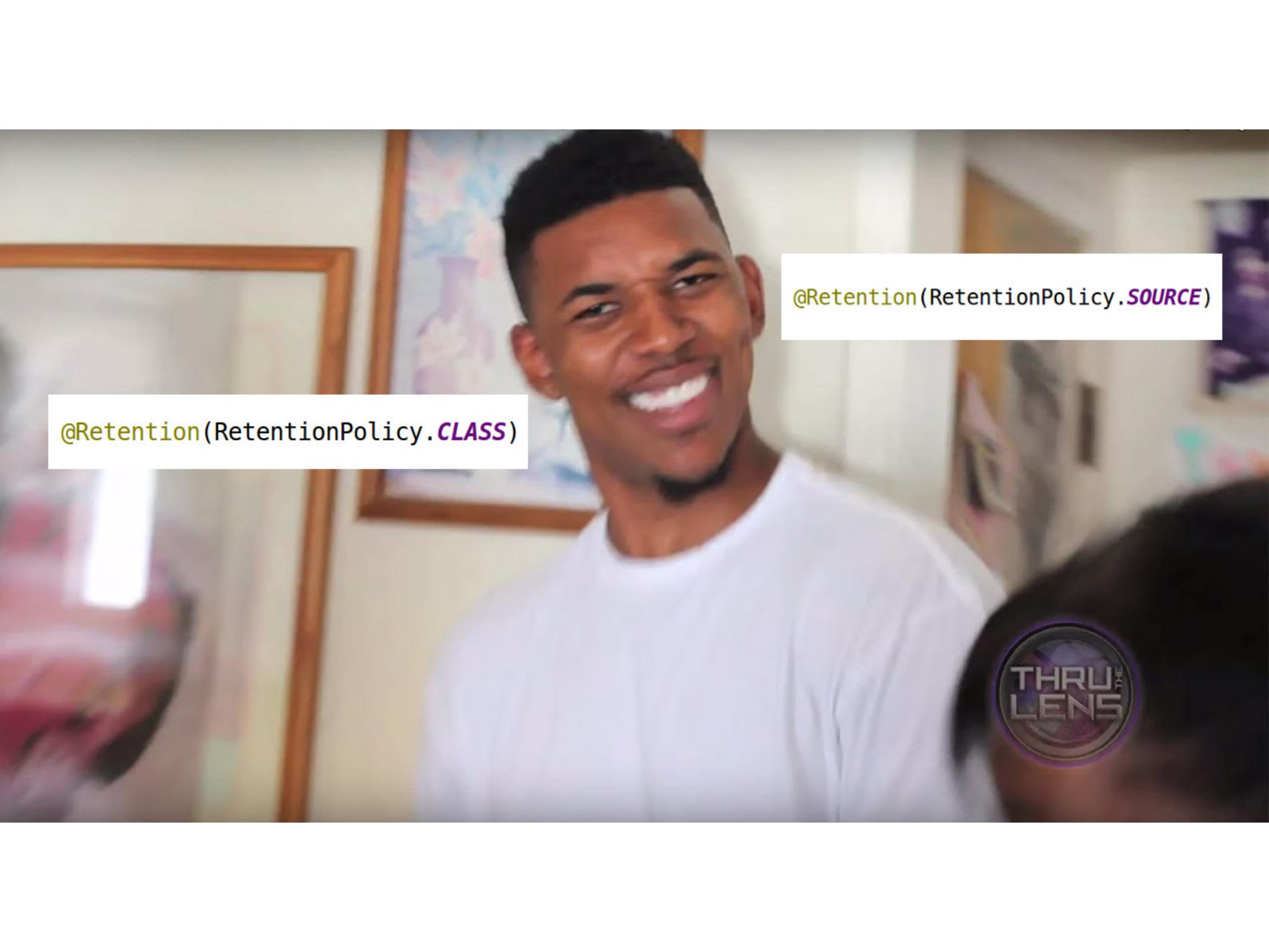
?

Argumentum ad verecundiam

- Что?
- Зачем?
- Где и как?
- Страхование коленных чашечек
- Итоги
- Вопросы

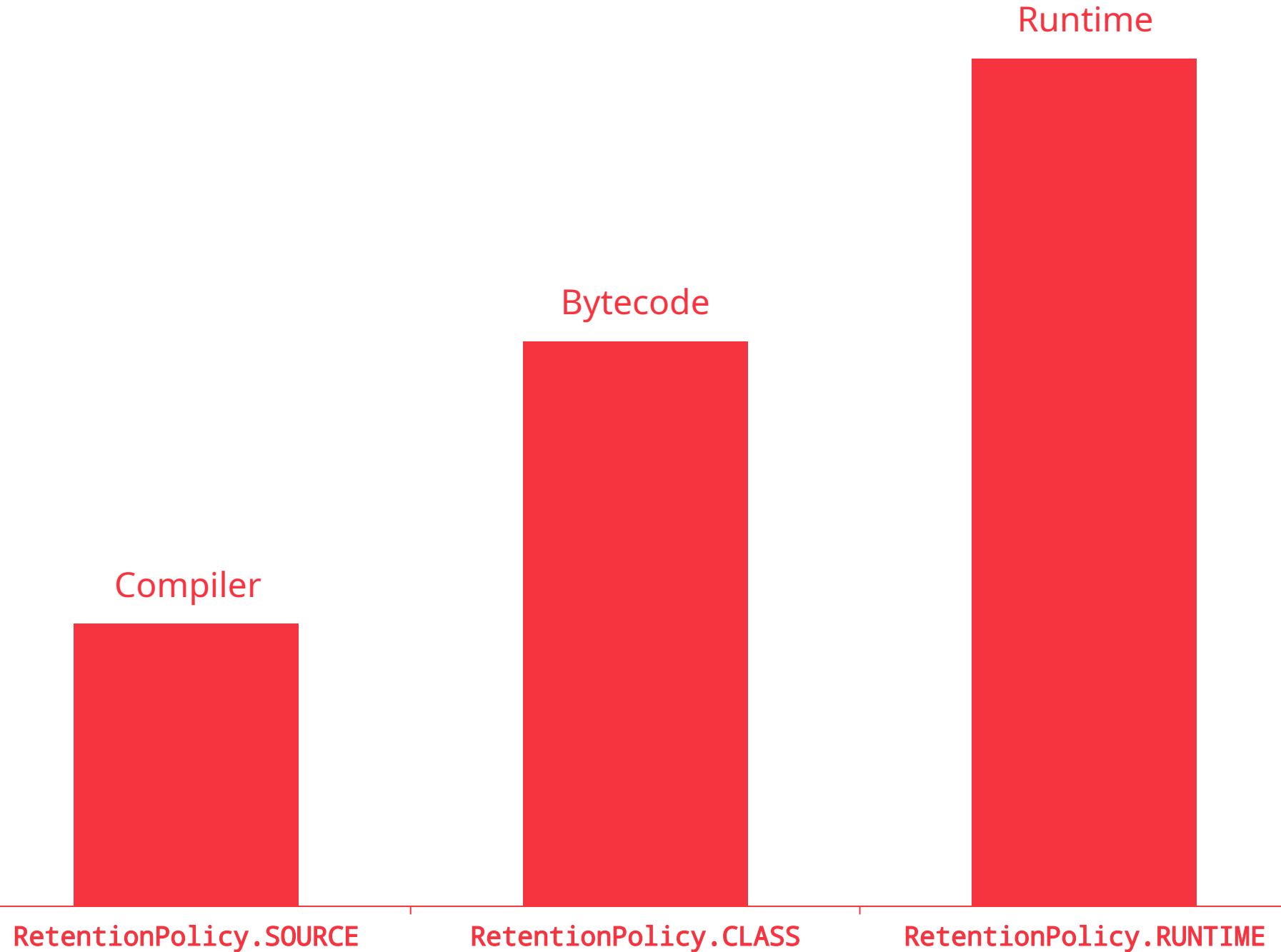
Что?

```
@Retention(RetentionPolicy.RUNTIME) // ???
```



```
@Retention(RetentionPolicy.CLASS)
```

```
@Retention(RetentionPolicy.SOURCE)
```



# RetentionPolicy.SOURCE

- Видно статическому анализу
- Видно компилятору (уay!)

Позволяет создавать производные на основе исходного кода

# RetentionPolicy.CLASS

- Видно в байткоде
- Видно при инструментации (наверное)

Позволяет производить онлайн- и оффлайн-инструментацию и анализ кода

# Процессинг аннотаций во время компиляции

- Позволяет найти все аннотированные чем-нибудь элементы
- (Благодаря этому) позволяет создавать дополнительные ресурсы и классы
- Позволяет хакать компиляцию (Lombok)

# МЕТАПРОГРАММИРОВАНИЕ!

# Что можно делать

- Авто-обнаружение классов (YAY!)
- Создавать ресурсы (например, SPI)
- Генерировать код и сразу отправлять его в компилятор (YAY!)

# Что нельзя делать

- Модифицировать исходный код (если не считать хакинга компилятора)
- Создавать nested classes (не считая создания нужных артефактов напрямую)
  - Перебивать старших

# Что можно, но не стоит делать

- Вызывать компилятор напрямую
- Создавать новые файлы в сорц- директории
- Хакать компиляцию (оставьте это ломбоку)

Зачем?

# Зачем?

- Автоматизация создания ресурсов (SPI, web.xml, log4.properties – любого ресурса) - см. Google Auto.Service
- Автоматическое обнаружение языковых элементов в коде вплоть до переменных
- Создание производных (e.g. Compile-Time AOP)
- Придумайте сами

# Почему просто не сканировать ClassPath?

- Это дешевле
- Это происходит не в рантайме
- Это проще, если требуется искать не только классы
- Это не загружает классы (хотя FastClassPathScanner вроде как умеет их и не загружать)
- Генерировать какие-либо производные на лету куда больней

Где и как?

Что происходит?

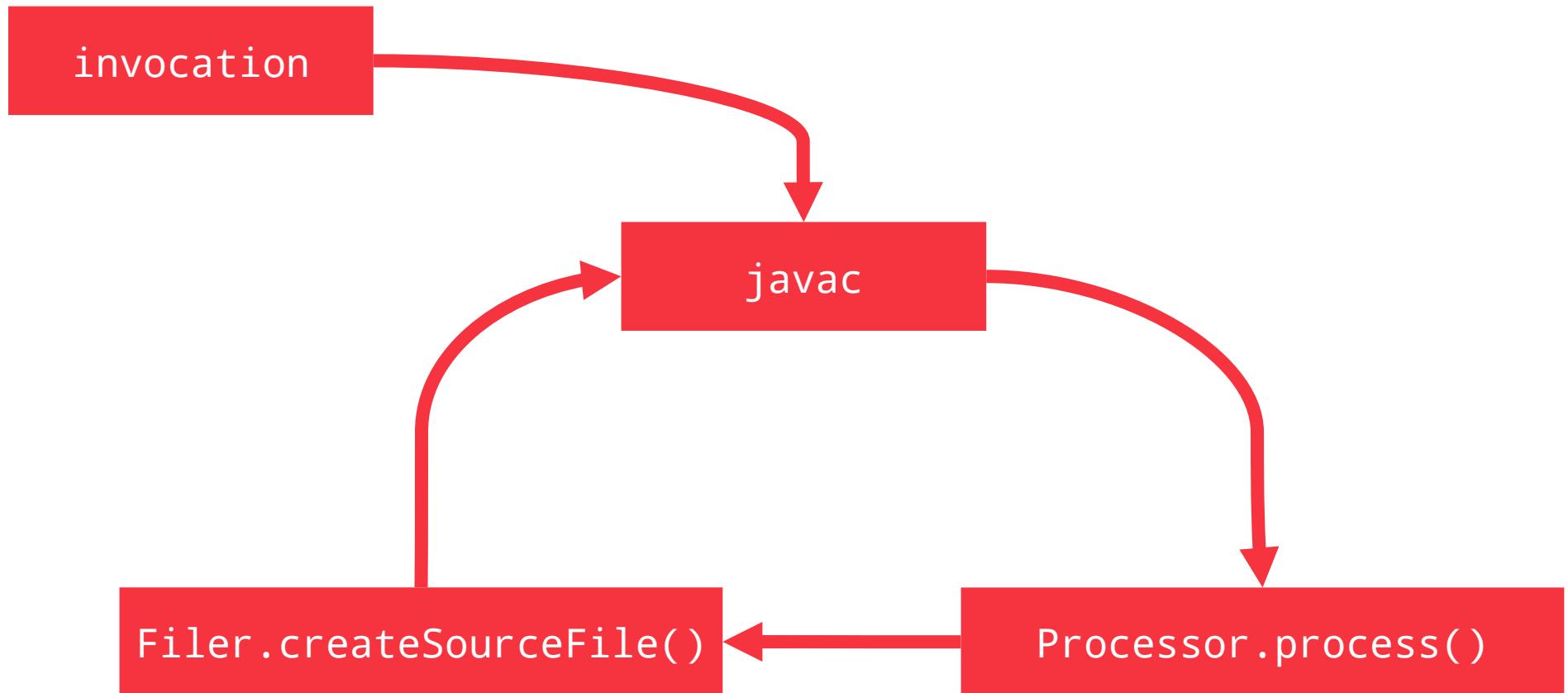
# Что происходит?

- Кто-то вызывает javac
- Если javac был вызван с ключом **-processor**, то процессор (единственный) находится по переданному FQCN
- Если javac был вызван без ключей, то он ищет имплементации с помощью SPI
- Конечно, можно дополнительно передать опции, но про это позже

# Что происходит?

- javac поднимает JVM и инстанцирует найденные процессоры.
- У каждого процессора вызывается метод `.init()`, куда передаются объекты API (`Filer`, `Messager`, etc).
- Каждый процессор объявляет, за какими аннотациями он хочет следить.
- javac компилирует всё, что видит, и затем вызывает метод `.process()` у процессоров.
- Процессоры могут генерировать новые классы, что повторно вызывает предыдущий шаг до того момента, когда за проход не будет создано ни одного нового подходящего класса.

# Oh, the infinite loop!



Какие классы используются?

# javax.annotation.processing.Processor

Интерфейс, который должен имплементировать  
процессор

- .init()
- .process()
- .getSupportedAnnotationTypes()
- неинтересное

# javax.annotation.processing.AbstractProcessor

Абстрактный класс, который берет всю необходимую информацию из аннотаций, сохраняет объекты API и оставляет для реализации метод `.process()`.

# javax.annotation.processing.AbstractProcessor

```
@SupportedAnnotationTypes({Constants.ANNOTATION})
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class LoudAnnotationProcessor extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment environment) {
        annotations.stream()
            .flatMap(annotation -> {
                return environment.getElementsAnnotatedWith(annotation)
                    .stream();
            })
            .forEach(element -> {
                processingEnv.getMessager().
                    printMessage(Diagnostic.Kind.NOTE, "Look what i've found!", element);
            });
        return false;
    }
}
```

# javax.lang.model.element.Element

Данный интерфейс представляет собой точку отсчета для отображения всех элементов языка, которые могут встретиться – метод, класс, тип дженерика, etc., etc. Конкретные элементы представлены отдельными отнаследованными интерфейсами.

При обработке аннотации процессор будет получать аннотированные элементы именно в таком виде.

# Кодогенерация

# Кодогенерация

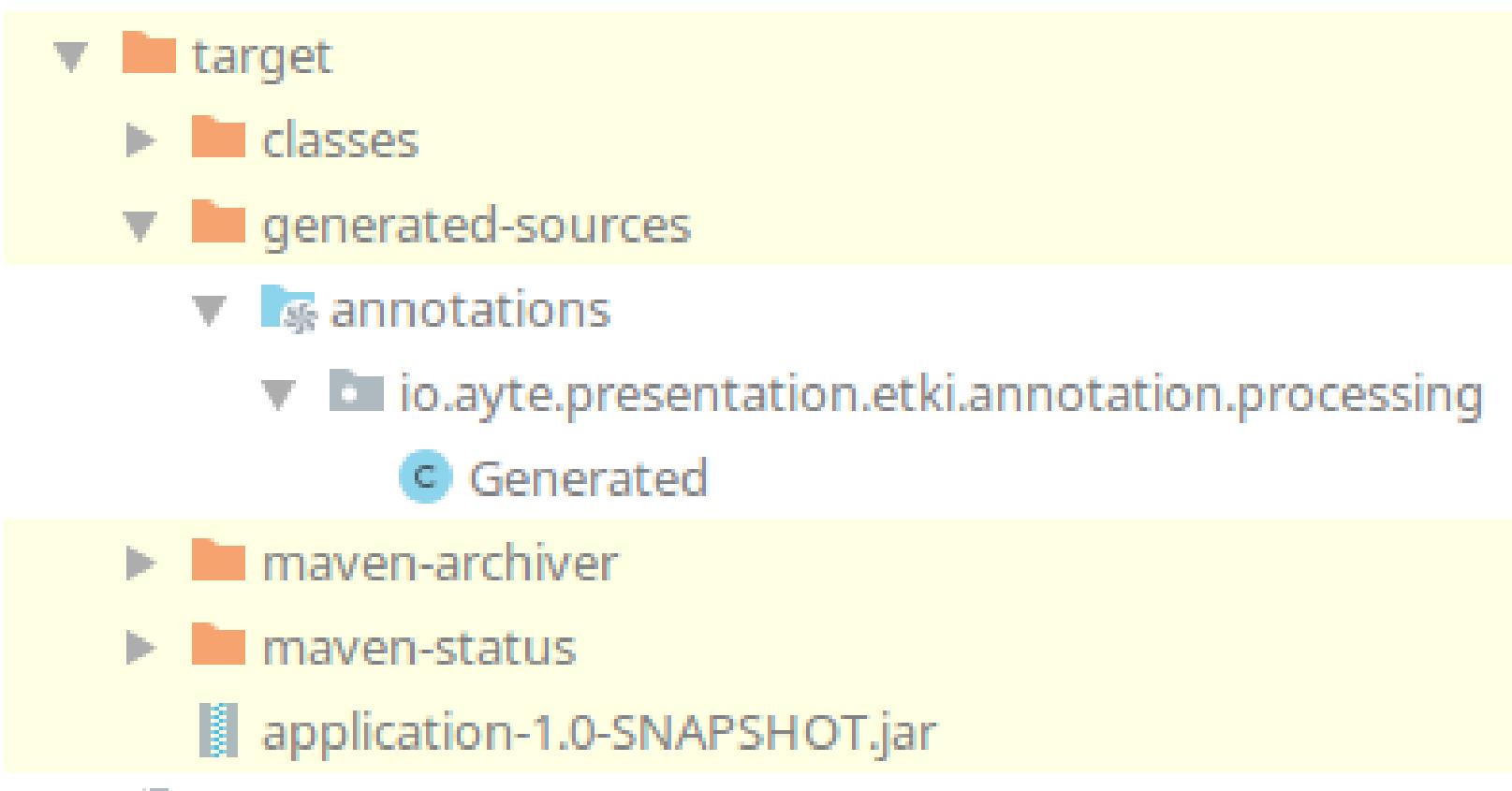
Класс ProcessingEnvironment, чей экземпляр передается в init(), содержит в себе инстанс класса Filer, который создан для работы с созданием файлов.

В том числе, файлов с исходным кодом, которые будут скомпилированы и доступны на следующей итерации.

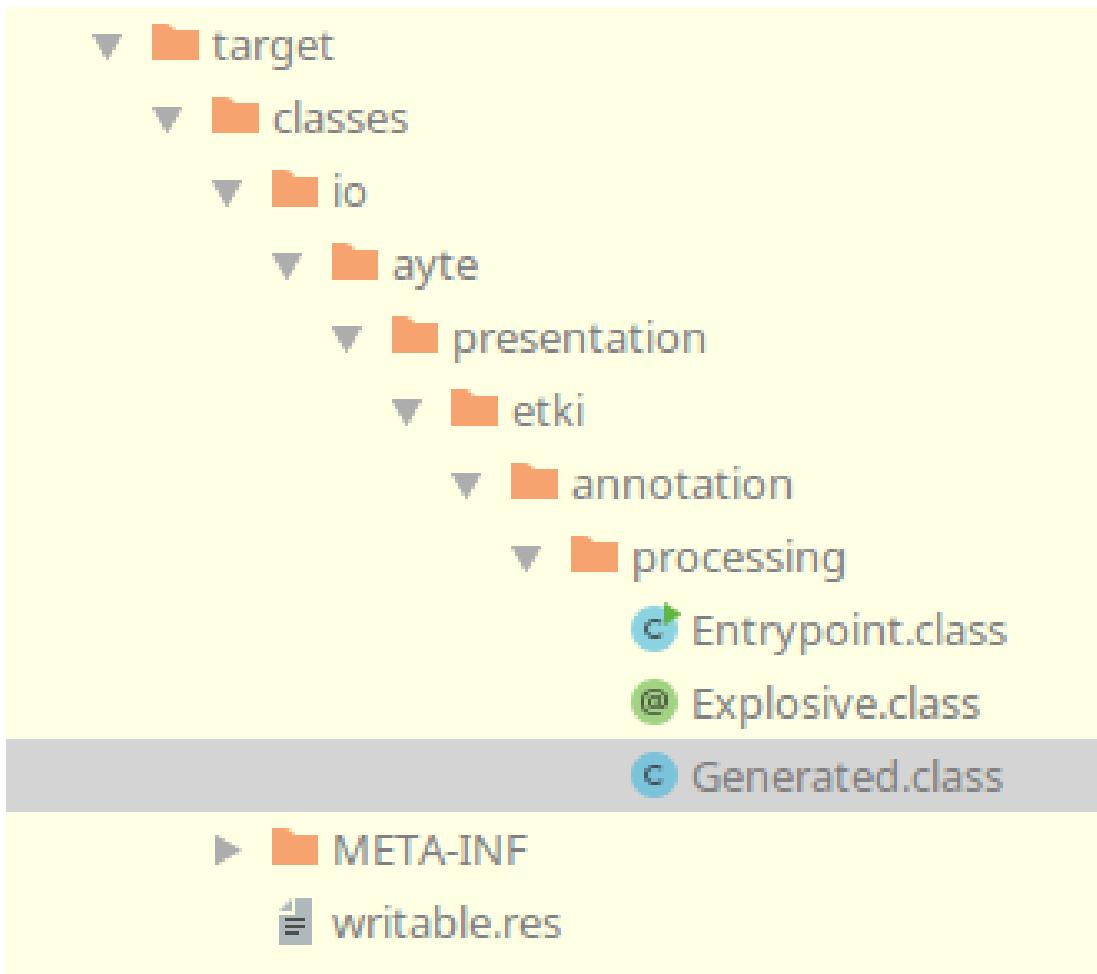
# Кодогенерация

```
processingEnv  
    .getFiler()  
    .createSourceFile("pkg.Generated")  
    .openWriter()  
    .append("package pkg;\n\n")  
    .append("class Generated {}\n")  
    .close();
```

# Кодогенерация



# Кодогенерация



# Добавление скомпилированных классов

# Добавление скомпилированных классов

```
processingEnv  
    .getFiler()  
    .createClassFile("pkg.Generated")  
    .openOutputStream()  
    .write(bytes);
```

# Добавление скомпилированных классов

Но вот только зачем?

# Создание ресурсов

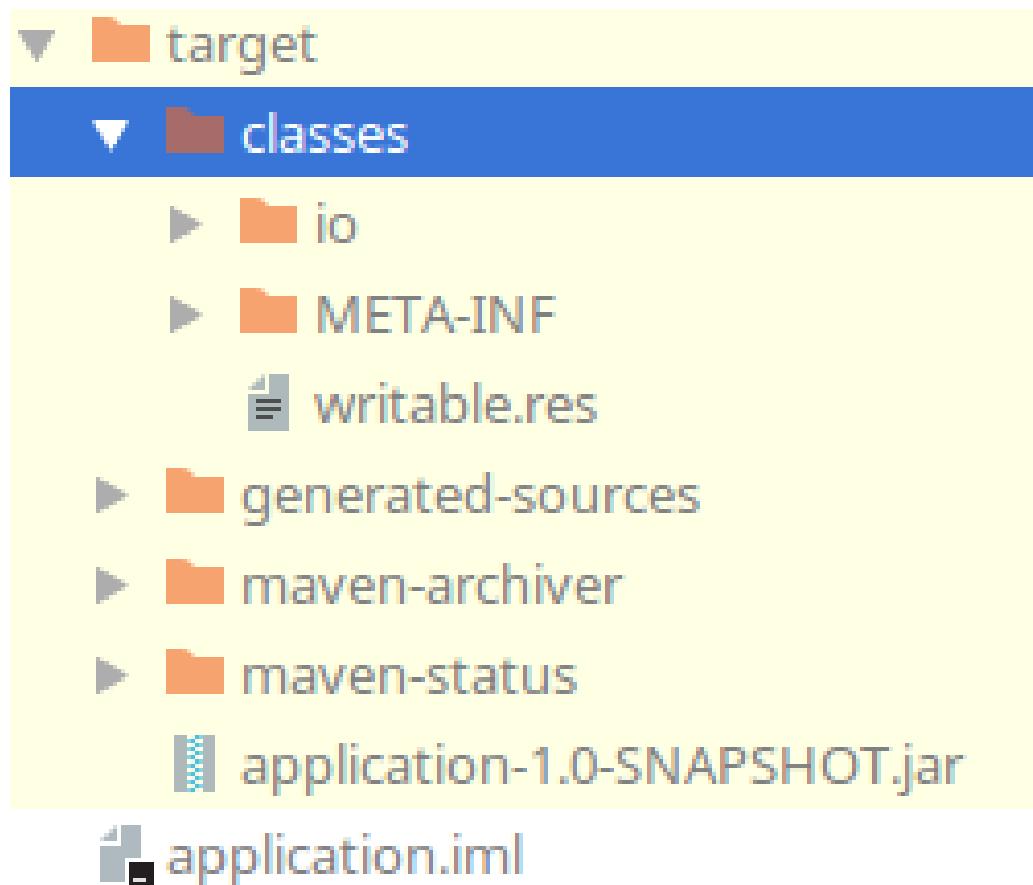
# Создание ресурсов

Также объект Filer дает возможность писать  
обычные ресурсы в локацию CLASS\_OUTPUT

# Создание ресурсов

```
processingEnv  
.getFiler()  
.createResource(  
    StandardLocation.CLASS_OUTPUT,  
    "",  
    "writable.res"  
)  
.openWriter()  
.append("Hey there!");
```

# Создание ресурсов



Доступ к существующим  
ресурсам

# Доступ к существующим ресурсам

Объект Filer позволяет не только создавать новые ресурсы, но и читать существующие

# Доступ к существующим ресурсам

Но есть одна проблема: у меня это не получилось

# Доступ к существующим ресурсам

```
processingEnv  
    .getFiler()  
    .getResource(  
        location,  
        pkg,  
        relativeName  
    )  
    .openReader();
```

???

# Доступ к существующим ресурсам

Впрочем, довольно сложно придумать ситуацию,  
где это действительно было бы нужно

# Логирование

# Логирование

Процессорам аннотаций доступен объект `Messager`, с помощью которого можно отправлять сообщения во внешний мир. Также методы `Messager` могут принимать на вход конкретный элемент, чтобы более наглядно описать сообщение.

# Логирование

```
processingEnv  
.getMessenger()  
.printMessage(  
    Diagnostic.Kind.NOTE,  
    "Look what i've found!",  
    element  
);
```

# Логирование

...Note: Look what i've found!  
public class Entrypoint {  
 ^

# Логирование

...Как правило, общаться с внешним миром не требуется вовсе.

# Передача опций (конфигурирование)

# Передача опций (конфигурирование)

Компилятор (javac) предусматривает передачу параметров процессорам аннотаций с помощью префикса -A (по аналогии с -D)

# Передача опций (конфигурирование)

```
javac \
    -processor io.discovery.Processor \
    -Aio.discovery.Processor.verbose=true \
    %file%
```

# Передача опций (конфигурирование)

Получить опции можно из передаваемого в процессор ProcessingEnvironment:

```
processingEnv  
    .getOptions()  
    .get("io.discovery.Processor.verbose")
```

# Передача опций (конфигурирование)

Для названия опций нет каких-либо ограничений, но я рекомендую все-таки придерживаться стандартной reverse dns name-нотации, чтобы избежать коллизий с другими процессорами.

Как правило, процессор не должен конфигурироваться вообще, за исключением, может быть, уровня verbosity.

# Демонстрация

(торжественный запуск mvn и  
javac, бурные аплодисменты)

# Страхование коленных чашечек

1. Классы не существуют

1.1 Классы не существуют  
(во время компиляции)

# 1.1 Классы не существуют (во время компиляции)

```
Class.forName(element.getQualifiedName().toString());
```

# 1.1 Классы не существуют (во время компиляции)

```
Class.forName(element.getQualifiedName().toString());
```

Да будь же ты человеком, он же еще не родился!

# 1.1 Классы не существуют (во время компиляции)



Куда же он пропал...

# 1.1 Классы не существуют (во время компиляции)

```
Class.forName(element.getAttribute("name").toString());
```

Да будьте умны человеком, он же еще не родился!

1.2 Классы не существуют  
(в рантайме)

# 1.2 Классы не существуют (в рантайме)

- commons.jar
  - @Explosive class Dynamite {}
  - scope: provided
- fat.jar: discover(Explosive.class).map(Class::forName)
  - JVM стреляет ClassNotFoundException, потому что класс Dynamite не включен в fat.jar

## 1.2 Классы не существуют (в рантайме)



Просто ты опять забыл  
положить меня в JAR (((

# 1.2 Классы не существуют (в рантайме)

Решение:

оперировать только FQCN в виде строк и  
проверять существование

2. Артефакты существуют

2.1. Существующие артефакты  
накапливаются

# 2.1 Существующие артефакты накапливаются

- dependency-a.jar
  - **@Explosive class C4 {}**
  - output: pkg.ExplosiveRepository
  - output: META-INF/explosives.yml
- dependency-b.jar
  - **@Explosive class TNT {}**
  - output: pkg.ExplosiveRepository
  - output: META-INF/explosives.yml
- fat.jar:
  - + pkg.ExplosiveRepository (dep-a? dep-b?)
  - + META-INF/explosives.yml (dep-a? dep-b?)

## 2.1 Существующие артефакты накапливаются



**Dear, this jar is not fat  
enough for the two of us**

## 2.1 Существующие артефакты накапливаются

**Псевдорешение: проверять текущие артефакты и делать автоинкремент**

- ExplosiveStash1
- ExplosiveStash2

## 2.1 Существующие артефакты накапливаются

- dependency-a.jar:
  - ExplosiveStash1.class
- dependency-b.jar:
  - ExplosiveStash1.class
- fat.jar
  - ExplosiveStash1.class (dep-a? dep-b?)

## 2.1 Существующие артефакты накапливаются



А оно уже третий  
день как в проде

## 2.1 Существующие артефакты накапливаются

**Решение: не создавать блин артефакты с повторяющимися именами**

Лучше использовать рандом (например, UUID)  
для создания уникального имени

Перфекционист страдает, функциональность - нет

2.2. Артефакты создаются и  
рекурсируют

## 2.2. Артефакты создаются и рекурсируют

- anything.jar
  - output: @Explosive pkg.ExplosiveStash1
  - output: @Explosive pkg.ExplosiveStash2
  - output: @Explosive pkg.ExplosiveStash3



## 2.2. Артефакты создаются и рекурсируют



## 2.2. Артефакты создаются и рекурсируют

Просто будьте аккуратны

3. Классы существуют  
(в рантайме)  
(в рантайме компиляции)

# Классы существуют (в рантайме)

- fresh-processor:0.1.0
  - guava:23.0
- legacy-processor:213.2.13.14455
  - guava:12.0

# Классы существуют (в рантайме)

JAR hell



# Классы существуют (в рантайме)

**Решение: шейдить джарник**

**com.google.\* → your.artifact.id.com.google.\***

# Классы существуют (в рантайме)

Так как процессор нужен только для компиляции  
(mvn scope: provided), это не будет засорять  
конечный артефакт

домашнее  
задание

Видит ли процессор аннотаций  
аннотации из джарника-  
зависимости?

Как прочитать существующий  
ресурс?

# Итоги

# Общее понимание процессинга аннотаций во время компиляции

# Общее понимание процессинга аннотаций

Java вижиж процессор аннотаций  
вижиж дисковери SPI кодогенерация  
вижиж шейди джарники свои вижиж  
**МАГИЯ**

# Lombok

# Lombok

- Теперь вы знаете, как это работает (немношк)
- Теперь вы понимаете, почему он работает на аннотациях и почему даже он не может предоставить нам алиасы в джаве

# Lombok

<https://projectlombok.org/contributing/lombok-execution-path>

# Полезные ссылки

# *The* 101 Article

<http://hannesdorfmann.com/annotation-processing/annotationprocessing101>

# Google Auto.Service

<https://github.com/google/auto>

# JavaPoet

<https://github.com/square/javapoet>

# ByteCode Engineering Library

<https://github.com/apache/commons-bcel>

# Эта презентация

В недрах <https://github.com/ayte-io/slides>

бнопняш & Answers