**Аннотации — Java: Корпоративные приложения на Spring Boot**

* [Встроенные аннотации](https://ru.hexlet.io/courses/java-spring/lessons/annotations/theory_unit#vstroennye-annotatsii)
* [Кастомные аннотации](https://ru.hexlet.io/courses/java-spring/lessons/annotations/theory_unit#kastomnye-annotatsii)
  + [Устройство кастомных аннотаций](https://ru.hexlet.io/courses/java-spring/lessons/annotations/theory_unit#ustroystvo-kastomnyh-annotatsiy)
  + [Параметры кастомных аннотаций](https://ru.hexlet.io/courses/java-spring/lessons/annotations/theory_unit#parametry-kastomnyh-annotatsiy)

В Java аннотации встречаются часто, но особенно много их в Spring Boot. Чтобы понять, как работает фреймворк, нужно разобраться в устройстве аннотаций. В этом уроке мы познакомимся с ними и узнаем, как они работают.

**Аннотации** — это механизм со своим синтаксисом, который позволяет добавлять метаданные в код. Например, так мы можем добавить какую-то дополнительную информацию, которую затем можно прочитать из исходного кода class-файлов или получить **в рантайме** — то есть во время работы программы. Сами по себе аннотации на код никак не влияют, в этом смысле они похожи на комментарии. Все действия происходят в коде, который ищет аннотации и на их основе меняет поведение.

Аннотации можно указывать на разных уровнях кода. Сюда входят классы, методы и параметры:

**package** io.hexlet.spring**;**

**import** org.springframework.boot.SpringApplication**;**

*// Определения аннотаций — это обычный код, который нужно импортировать*

**import** org.springframework.boot.autoconfigure.SpringBootApplication**;**

**import** org.springframework.web.bind.annotation.GetMapping**;**

**import** org.springframework.web.bind.annotation.RestController**;**

**import** org.springframework.web.bind.annotation.RequestMapping**;**

**import** org.springframework.web.bind.annotation.RequestBody**;**

**import** org.springframework.web.bind.annotation.RequestParam**;**

**@RestController** *// Аннотация на уровне класса*

**@RequestMapping(**"/api"**)** *// Аннотация на уровне класса*

**@SpringBootApplication**

**public** **class** **Application** **{**

**@Autowired** *// Аннотация на уровне поля*

**private** **UserRepository** userRepository**;**

**public** **static** **void** **main(String[]** args**)** **{**

**SpringApplication.**run**(Application.**class**,** args**);**

**}**

**@GetMapping(**"/hello"**)** *// Аннотация на уровне метода*

**public** **String** **sayHello(@RequestParam(**name **=** "name"**,** required **=** **false,** defaultValue **=** "Guest"**)** **String** name**)** **{** *// Аннотация на уровне параметров*

**return** "Hello, " **+** name **+** "!"**;**

**}**

**@PostMapping(**"/greet"**)** *// Аннотация на уровне метода*

**public** **String** **greet(@RequestBody** **String** name**)** **{** *// Аннотация на уровне параметров*

**return** "Greetings, " **+** name **+** "!"**;**

**}**

**}**

Top of Form

Bottom of Form

Некоторые аннотации выглядят как метка — @RestController, другие похожи на вызов метода с параметрами — @RequestMapping("/api"). Принцип работы от этого не меняется: аннотация не превращается в вызов метода, она остается меткой с дополнительными данными.

Зачем аннотации вообще нужны? Во-первых, они значительно сокращают объем **шаблонного кода** — это повторяющийся одинаковый код, который нужен для конфигурации приложения, соединения его частей друг с другом или других задач. Раньше ту же задачу решали с помощью конфигурационных XML-файлов, которые иногда были просто огромными. Из-за этого Java-программистов часто называли XML-программистами. Активное использование аннотаций существенно упростило этот процесс.

**Встроенные аннотации**

Подавляющее большинство аннотаций в реальных проектах написаны разработчиками библиотек, а еще буквально несколько аннотаций встроено прямо в Java. Например, аннотация @Deprecated позволяет отметить класс или метод как устаревший. Эту информацию затем можно увидеть в подсказках редактора. Такая аннотация помогает другим программистам при выборе классов и методов для реализации их задач:

**@Deprecated**

**public** **void** **oldMethod()** **{**

*// Дальше продолжается какой-то код*

**}**

Top of Form

Bottom of Form

Самая часто используемая аннотация — это @Override. Она указывает, что помеченный метод должен переопределять метод наследуемого класса или реализовывать метод интерфейса. Сама аннотация не обязательна при переопределении, но она помогает избежать ошибок и сделать код проще для чтения:

*// Пример из Spring Boot*

**public** **class** **User** **implements** **UserDetails** **{**

**@Override**

**public** **boolean** **isAccountNonExpired()** **{**

**return** **true;**

**}**

**@Override**

**public** **boolean** **isAccountNonLocked()** **{**

**return** **true;**

**}**

**@Override**

**public** **boolean** **isCredentialsNonExpired()** **{**

**return** **true;**

**}**

**}**

Top of Form

Bottom of Form

**Кастомные аннотации**

С такими аннотациями мы будем встречаться чаще всего. Изучить их работу заранее невозможно — каждая конкретная аннотация обрабатывается своим образом и приводит к своим последствиям. Причем в большинстве случаев программист до конца не знает, что на самом деле происходит внутри.

С одной стороны, это хорошо — можно сфокусироваться на важном. С другой стороны, из-за этого код начинает работать как магия, и это может стать проблемой.

Изучим пример типичного контроллера на Spring Boot. Здесь можно насчитать около десятка аннотаций, причем из разных пакетов:

**import** org.springframework.beans.factory.annotation.Autowired**;**

**import** org.springframework.http.HttpStatus**;**

**import** org.springframework.web.bind.annotation.GetMapping**;**

**import** org.springframework.web.bind.annotation.PathVariable**;**

**import** org.springframework.web.bind.annotation.PostMapping**;**

**import** org.springframework.web.bind.annotation.RequestBody**;**

**import** org.springframework.web.bind.annotation.RequestMapping**;**

**import** org.springframework.web.bind.annotation.ResponseStatus**;**

**import** org.springframework.web.bind.annotation.RestController**;**

**import** jakarta.validation.Valid**;**

**@RestController**

**@AllArgsConstructor**

**@RequestMapping(**"/api"**)**

**public** **class** **PostsController** **{**

**@Autowired**

**private** **final** **PostRepository** repository**;**

**@PostMapping(**"/posts"**)**

**@ResponseStatus(HttpStatus.**CREATED**)**

**PostDTO** **create(@Valid** **@RequestBody** **PostDTO** postData**)** **throws** **JsonProcessingException** **{**

*// Тут логика*

**}**

**@GetMapping(**"/posts/{id}"**)**

**@ResponseStatus(HttpStatus.**OK**)**

**PostDTO** **show(@PathVariable** **Long** id**)** **{**

*// Тут логика*

**}**

**}**

Top of Form

Bottom of Form

Как классы или интерфейсы, аннотации тоже имеют свое определение, поэтому их необходимо импортировать. Редактор делает это самостоятельно, поэтому тут проблем возникнуть не должно.

**Устройство кастомных аннотаций**

Посмотрим, как определять и обрабатывать аннотации. Эти знания помогут разобраться в принципе работы аннотаций, а еще вопросы на эту тему часто задают на собеседованиях. Напишем аннотацию @LogExecutionTime, которая замеряет время выполнения помеченного ей метода:

**import** java.lang.annotation.ElementType**;**

**import** java.lang.annotation.Retention**;**

**import** java.lang.annotation.RetentionPolicy**;**

**import** java.lang.annotation.Target**;**

**@Retention(RetentionPolicy.**RUNTIME**)**

**@Target(ElementType.**METHOD**)**

**public** **@interface** **LogExecutionTime** **{**

**}**

Top of Form

Bottom of Form

Иронично, что определение аннотации само помечено ими. В коде выше мы видим три обязательные аннотации:

* @interface определяет саму аннотацию
* @Retention определяет жизненный цикл аннотации, то есть указывает, как долго аннотация должна оставаться с кодом. В этом случае аннотация должна быть доступна в рантайме, потому что именно так мы будем ее обрабатывать
* @Target определяет, где мы будем применять аннотацию (например, в методах)

Аннотация готова, можно начинать применять ее. При этом в работе кода ничего не поменяется, потому что обработчик еще не написан:

*// Мы должны импортировать нашу аннотацию*

**import** <какой-то путь>.LogExecutionTime**;**

**public** **class** **SomeService** **{**

**@LogExecutionTime**

**public** **void** **serve()** **throws** **InterruptedException** **{**

**Thread.**sleep**(**1500**);** *// Выполняем какую-то задачу*

**}**

**public** **void** **anotherMethod()** **{**

*// Этот метод еще не отмечен аннотацией,*

*// поэтому время выполнения метода не измеряется и не логируется*

**}**

**}**

Top of Form

Bottom of Form

Дальше мы напишем обработчик аннотации. Это обычный Java-код, поэтому нужно убедиться, что он выполняется до того, как исполнение дойдет до кода с аннотациями. В нашем случае обработчик выполняется в методе main():

**import** java.lang.reflect.Method**;**

**public** **class** **Main** **{**

**public** **static** **void** **main(String[]** args**)** **{**

**var** service **=** **new** **SomeService();**

*// Итерируем все методы класса*

**for** **(Method** method **:** **SomeService.**class**.**getDeclaredMethods**())** **{**

*// Проверяем, есть ли у метода аннотация @LogExecutionTime*

**if** **(**method**.**isAnnotationPresent**(LogExecutionTime.**class**))** **{**

**var** startTime **=** **System.**currentTimeMillis**();**

**try** **{**

*// Выполняем метод с аннотацией LogExecutionTime*

method**.**invoke**(**service**);**

**}** **catch** **(Exception** e**)** **{**

e**.**printStackTrace**();**

**}**

**long** endTime **=** **System.**currentTimeMillis**();**

**long** executionTime **=** endTime **-** startTime**;**

**System.**out**.**println**(**"Executed method: " **+** method**.**getName**());**

**System.**out**.**println**(**"Execution time: " **+** executionTime **+** " milliseconds"**);**

**}**

**}**

**}**

**}**

Top of Form

Bottom of Form

Здесь мы видим **рефлексию** — технику, которая отображает информацию о программе во время ее работы. Чтобы использовать ее, мы берем все методы класса SomeService, находим методы с аннотацией LogExecutionTime и вызываем их методы, проверяя время выполнения.

**Параметры кастомных аннотаций**

Добавим в @LogExecutionTime два параметра. Первый временно выключит логирование, а второй задаст минимальное время выполнения, ниже которого логировать не нужно:

**import** java.lang.annotation.ElementType**;**

**import** java.lang.annotation.Retention**;**

**import** java.lang.annotation.RetentionPolicy**;**

**import** java.lang.annotation.Target**;**

**@Retention(RetentionPolicy.**RUNTIME**)**

**@Target(ElementType.**METHOD**)**

**public** **@interface** **LogExecutionTime** **{**

**boolean** **enabled();**

**long** **threshold()** **default** 0**;** *// Пороговое время в миллисекундах*

**}**

Top of Form

Bottom of Form

Параметры описываются внутри тела аннотации определенным способом. Он похож на определение методов с отсутствующим телом и возможностью указать значение по умолчанию. Кстати, значение по умолчанию можно и не прописывать. В таком случае компилятор потребует указать его при добавлении аннотации:

**public** **class** **SomeService** **{**

**@LogExecutionTime(**enabled **=** **true)**

**public** **void** **serve()** **throws** **InterruptedException** **{**

**Thread.**sleep**(**1500**);** *// Выполняем какую-то задачу*

**}**

**@LogExecutionTime(**enabled **=** **true,** threshold **=** 100**)**

**public** **void** **anotherMethod()** **{**

**}**

**}**

Вопрос: 1 из 4. Вы ответили верно на 0

Top of Form

Bottom of Form

Top of Form

Bottom of Form

Что можно делать с помощью аннотаций?

Top of Form

Добавлять в код информацию, которую можно получить в рантайме или прочитать из исходного кода class-файлов



Описывать метод



Комментировать код

Bottom of Form