Аннотации в Java. Не путать с комментариями

Что такое аннотации в Java, зачем они нужны, как выглядят и работают. Краткий обзор для новичков.

08 ноября 20175 минут14445



Аннотации в Java – зачем они нужны

Аннотации – это пометки, с помощью которых программист указывает компилятору Java и средствам разработки, что делать с участками кода помимо исполнения программы. Аннотировать можно переменные, параметры, классы, пакеты. Можно писать свои аннотации или использовать стандартные – встроенные в Джава.

Вы узнаете аннотацию по символу @ в начале имени: @Override – стандартная аннотация Javа, которая предупреждает, что ниже мы что-то переопределим:

class SomeClass {

   void method() {

       System.out.println("Работает метод родительского класса.");

   }

}

class AnotherClass extends SomeClass { // наследуем методы SomeClass в новом классе

@Override

   void method() { // переопределяем метод

       System.out.println("Работает метод класса-потомка.");

   }

}

Если в имени метода из AnotherClass будет опечатка, компилятор учтет @Override и выдаст ошибку. Без аннотации он не заметил бы подвоха и безропотно создал бы новый метод в дополнение к method из SomeClass.

Обратите внимание, сама аннотация никак не влияет на переопределение метода, но позволяет контролировать успешность переопределения при компиляции или сборке. Мы защитили участок кода от неприметной ошибки, на поиск которой в большой программе ушли бы часы. Это лишь одно из многих применений аннотаций.

Зачем нужны аннотации Java

Они позволяют:

* автоматически создавать конфигурационные XML-файлы и дополнительный Java-код на основе исходного аннотированного кода;
* документировать приложения и базы данных параллельно с их разработкой;
* проектировать классы без применения маркерных интерфейсов;
* быстрее подключать зависимости к программным компонентам;
* выявлять ошибки, незаметные компилятору;
* решать другие задачи по усмотрению программиста.

Поясним понятие «маркерный интерфейс». Интерфейсы без каких-либо методов действуют как маркеры. Они лишь говорят компилятору, что объекты классов, которые имплементируют такой интерфейс без методов, должны иметь отличительные черты, восприниматься иначе. Например, java.io.Serializable, java.lang.Cloneable, java.util.EventListener. Маркерные интерфейсы ещё известны как «теги» — они добавляют общий тег ко всем унаследованным классам и объединяют их в одну категорию.

При первом появлении в Java EE 5 аннотации были представлены как инструмент, который ускоряет разработку больших web-сервисов и клиентских приложений. Как это работает?

Обработка аннотации в Джава

На основе аннотаций компилятор может с помощью специальных обработчиков генерировать новый код и файлы конфигурации.  
  
Обработчиками обычно выступают библиотеки и утилиты, которые можно брать у сторонних авторов (или создавать самостоятельно) и прикреплять к проекту в среде разработки. Способ подключения зависит от IDE или системы сборки. В Maven обработчики подключают с помощью модуля annotation-user или плагина maven-compiler-plugin.  
  
Парсинг аннотаций происходит циклически. Компилятор ищет их в пользовательском коде и выбирает подходящие обработчики. Если вызванный обработчик на основе аннотации создаёт новые файлы с кодом, начинается следующий этап, где исходным материалом становится сгенерированный код. Так продолжается до тех пор, пока не будут созданы все необходимые файлы.

Пишем первую аннотацию на Java

Допустим, у нас есть веб-сервис, который поддерживает несколько версий одного функционала для соблюдения совместимости. И есть обработчик аннотаций, который позволяет компилятору выбирать нужные версии. На минутку забудем о существовании Git :)

Где хранить данные о версии и авторе функционала? Конечно же, в аннотации. Напишем её. Новую аннотацию объявляют с помощью ключевого слова @interface:

public class SomeClass {

public @interface version {

private float v(); // номер версии

private String author() default  “Аноним"; // автор

}

// остальное содержимое класса

}

Это немного искусственный, но зато простой и наглядный пример аннотации на Java. Мы добавили два атрибута, которые выглядят как методы. Отличие в том, что при объявлении атрибутов никогда не используют оператор throws и  не назначают параметров. Значениями могут выступать:

* примитивные типы Java,
* классы или снабженные параметрами обращения к классам,
* перечисления,
* другие аннотации,
* массивы из вышеперечисленных элементов.

Можно указывать значения по умолчанию, что мы и сделали выше с полем author. При постановке аннотации атрибуты с дефолтными значениями можно пропускать.

@version(v=1.0f); // автор остаётся “Анонимом”, а "f" после числа ставим для явного указания на тип float

SomeClass {

// ...

}

Чтобы нашу аннотацию использовали только по назначению, вернёмся к её объявлению и укажем, где и когда она должна работать:

@Target(ElementType.TYPE) /\* Аннотация применима только к классам, а не к пакетам,

отдельным методам, переменным и т.д.

\*/

@Retention(RetentionPolicy.RUNTIME)

/\* Применяется во время выполнения программы.

Если бы нам нужно было применять аннотацию к исходному коду

на этапе компиляции программы, мы бы указали RetentionPolicy.SOURCE.\*/

public @interface version {

// ...

}

Остается ассоциировать аннотацию с нужным классом и запустить программу.

Используем аннотации Java для сравнения баз данных

Напишем что-нибудь более сложное и полезное. Нам нужно свести несколько БД в одну. Для начала сравним их, чтобы найти одинаковые поля и значения, устранить дубли и внести новую информацию из каждого источника. Могут возникнуть и сопутствующие задачи: одни значения потребуется отформатировать, другие  —   проигнорировать, всё вместе вывести в виде .xls-отчёта.

Мы наметили конфигурацию работы с данными, остаётся её реализовать. Вот здесь на сцену выходят аннотации. С их помощью мы можем задать (а если потребуется — и скорректировать) способы обработки каждого поля. При этом конфигурация будет выглядеть абсолютно прозрачно: читатель кода сразу поймёт, что к чему применено. И ход исполнения программы не будет затронут. Волшебно! Пишем:

// Сначала укажем, что аннотацию надо применять на уровне ПОЛЯ

@Target(ElementType.FIELD)

// Использовать аннотацию надо во время выполнения программы

@Retention(RetentionPolicy.RUNTIME)

public @interface FieldInspector { //создаём аннотацию, придумываем ей имя

// Выбираем, сравнивать ли источники. По умолчанию — да.

boolean compareSources() default true;

/\* Получаем формат значения в отчёте, а если не получаем — подставляем формат по умолчанию — native. \*/

СhooseTheFormat displayFormat() default СhooseTheFormat.NATIVE;

// Получаем ID текущего поля для сопоставления в нескольких базах

int id();

// Уточняем, какое имя использовать для поля в таблице.

String field\_name() default "";

// Какие источники сравнивать  —  формируем список.

TheSource[] sourcesToCompare() default {};

}

Теперь ассоциируем аннотацию с полем. Допустим, у нас в БД книги:

@FieldInspector(id = 2, field\_name = "TITLE")

private String book\_title; // переменная хранит название книги

Присвоили id, присвоили имя полю в отчёте. Сверка значений со всеми базами запустится по умолчанию, т.к мы не указали для compareSources значение false. А если мы заведомо знаем, что искать нужно не во всех БД, а в конкретных? Например, сюжет есть не у любой книги. Задаём источники вручную:

// проверять только пьесы и беллетристику

@FieldInspector(id = 3, field\_name = "PLOT" sourcesToCompare ={TheSource.PLAYS, TheSource.FICTION})

private String plot;

Если у нас есть поле «Примечания», значение которого не нужно сверять, мы можем перед переменной для хранения примечаний поставить такую аннотацию:

@FieldInspector(id = 300, field\_name = "NOTES", compareSources = false)

private String notes;

Неплохо, но как сделать, чтобы правила обработки выбирались в зависимости от значения поля? Можно связать правила с классом обработчика, чтобы тот выбирал, как поступать с каждым полем. Для этого потребуется ещё одна короткая аннотация:

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

public @interface RuleApplier {

// массив строк, где хранятся параметры обработчика, по умолчанию — параметров нет

String[] parameters() default {};

// класс, который отвечает за применение правил к каждому источнику

Class<?> processor() default MyRuleApplier.class;

}

Проверим, как это работает. Давайте не сравнивать данные о литературе, которая поступила в библиотеку до 1990 года  —  допустим, эти данные у нас были только в одной базе:

@FieldInspector(id = 4, label = "THE YEAR OBTAINED", displayFormat =

СhooseTheFormat.YEAR\_FORMAT})

@RuleApplier(processor = IgnoreWhatWasObtainedBefore.class, parameters = { "1990" })

private int book\_obtained;

Теперь реагировать на значения менее 1990 в поле «год поступления» мы доверяем конкретному пользовательскому классу. Это даёт нам больше гибкости в работе с данными, но помните, что аннотации  —  вещь статическая. Какое значение получил обработчик на входе, с тем и разбирается. Интерактива с динамической сменой значений пользователем не получится.