УДК 004.43 **DOI:** 10.31673/2412-9070.2021.045659

Д. Є. АЛТИННІКОВ, студент;
О. О. ШЕВЧЕНКО, аспірантка;
І. І. БЕРДНИК, аспірантка;
О. В. ЗУБ, аспірант;
В. А. САГАЙДАК, аспірант,
Державний університет телекомунікацій, Київ

ВИКОРИСТАННЯ JAVA-АНОТАЦІЙ ЯК ІНСТРУМЕНТУ НАДАННЯ API

Java-анотації є потужною частиною мови програмування, але здебільшого їх використовують зазвичай користувачі деякої логіки, а не автори анотацій. Наприклад, не важко знайти вихідний код Java, який містить анотацію @Override, оброблену компілятором Java, анотацію @Autowired, яка використовується фреймворком Spring, або анотацію @Entity, що застосовується фреймворком Hibernate, але рідко можна побачити анотації, написані користувачем. Незважаючи на те, що користувацькі анотації є аспектом мови Java, яку не дуже часто використовують, вони можуть бути доволі корисним ресурсом під час розроблення коду, який приємно читати і водночас лаконічно досягати своїх цілей, оскільки це роблять такі фреймовки, як Spring або Hibernate.

Ключові слова: Java; JVM; анотація.

Вступ

Анотації — це декоратори, які застосовують конструкцій Java, зокрема класи, методи або поля, що пов'язують метадані з конструкцією. Ці декоратори є доброякісними і не виконують жодного коду самостійно, але можуть використовуватися в межах виконання або компілятором для здійснення певних дій. Специфікація мови Java (JLS), розділ 9.7, містить таке визначення: анотація — це маркер, який пов'язує інформацію з конструктором програми, але не діє під час виконання.

Одним із найважливіших пунктів є те, що анотації не впливають на програму під час виконання. Це не свідчить про те, що фреймворк не може змінювати свою поведінку на основі наявності анотації під час виконання, але це означає, що наявність самої анотації не змінює поведінку програми під час виконання. Хоча таке може сприйматись як нюанс, але це дуже важливий аспект, який потрібно розуміти, щоб усвідомити корисність анотацій.

Основна частина

Для створення анотації потрібні дві частини інформації: *політика збереження* та *ціль*. ***Політика збереження*** визначає, як довго, з погляду життєвого циклу програми, анотація має зберігатися. Наприклад, анотації можуть зберігатися під час компіляції або під час виконання, залежно від політики збереження, пов'язаної з анотацією. Станом на Java 9 існує три стандартні політики збереження:

Політика збереження	Опис
Джерело	Анотації видаляються компілятором
Клас	Анотації записуються у файл класу, створений компілятором, але їх не потрібно зберігати віртуальною машиною Java (JVM), яка обробляє файл класу під час виконання
Під час виконання	Анотації записуються у файл класу компілятором і зберігаються під час виконання JVM

Як ми незабаром побачимо, варіант із політикою «Під час виконання» для збереження анотацій є одним із найпоширеніших, оскільки дає можливість програмам на Java рефлекторно отримувати доступ до анотації та виконувати код на основі наявності анотації, а також отримувати доступ до даних, пов'язаних із анотацією. Досить важливо зауважити, що анотація може мати лише одну політику збереження.

Ціль	Опис
Тип анотації	Позначає іншу анотацію
Конструктор	Позначає конструктор
Поле	Позначає поле, зокрема змінну екземпляра класу або константу переліку
Локальна змінна	Позначає локальну змінну
Метод	Позначає метод класу
Модуль	Позначає модуль
Пакет	Позначає пакет
Параметр	Позначає параметр методу або конструктору
Тип	Позначає тип, наприклад клас, інтерфейси, типи анотацій або оголошення переліку
Тип, параметр	Позначає параметри типу, зокрема ті, що використовуються як офіційні загальні параметри

Ціль анотації визначає, до якої конструкції Java можна застосувати анотацію. Наприклад, деякі анотації можуть бути дійсними лише для методів, тоді як інші можуть бути дійсними як для класів, так і для полів. Починаючи з Java 9, існує одинадцять стандартних цілей анотацій:

Важливо зазначити, що одна або кілька цілей можуть бути пов'язані з анотацією. Наприклад, якщо цільові поля та конструктори пов'язані з анотацією, то її можна використовувати як у полях, так і в конструкторах. Водночас якщо анотація має лише відповідну ціль до методу, то застосування анотації до будь-якої конструкції, крім методу, призводить до помилки під час компіляції.

Анотації також можуть мати пов'язані параметри. Вони можуть бути примітивними (наприклад, int або double), String, class, enum, annotation або масивом будь-якого з п'яти попередніх типів. Пов'язування параметрів з анотацією дає їй можливість надавати контекстну інформацію або параметризувати певні значення.

Наприклад, нам потрібно створити анотацію, яка могла б анотувати поля класу автомобіля (зокрема марку та модель) за допомогою нашої анотації, що буде нам необхідно для серіалізації об'єкта автомобіля в JSON. Створений JSON об'єкт буде мати ключі make і model, де значення подають значення полів відповідно make та model. Для простоти вважатимемо, що ця анотація буде використовуватися лише для полів типу String, гарантуючи, що значення поля можна безпосередньо серіалізувати як рядок.

Щоб створити таку анотацію поля, ми оголошуємо нову анотацію за допомогою ключового слова @interface:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.JFIELD)
public @interface JField {
  public String value() default "";}
```

Ядром нашої декларації є публічний @interface JField, який декларує тип анотації з public modifier, що дає змогу використовувати нашу анотацію в будь-якому пакеті (за умови, що пакет належним чином імпортовано). Тіло анотації оголошує єдиний параметр String, і значення за замовчуванням для нього — це порожній рядок.

Важливо зауважити, що значення імені змінної має особливу вагу: воно визначає одноелементну анотацію і дає можливість користувачам нашої анотації надавати єдиний параметр анотації без зазначення назви параметра. Наприклад, користувач може анотувати поле за допомогою @JField ("someFieldName") і не зобов'язаний оголошувати анотацію як @JField (value = "someFieldName"), хоча останній варіант досі може використовуватися (але це не обов'язково). Додавання значення за замовчуванням до порожнього рядка дозволяє пропустити значення. Це призводить до того, що зна-

чення містить порожній рядок, якщо явно не вказано жодного значення. Наприклад, якщо користувач оголошує таку анотацію, застосовуючи форму @JField, тоді для параметра значення встановлюється порожній рядок.

Політика збереження та ціль оголошення анотації визначаються відповідно до анотацій @Retention та @Target. Політику збереження можна схарактеризувати за допомогою переліку java.lang.annotation.RetentionPolicy, що містить константи для кожної з трьох стандартних політик збереження. Аналогічно ціль формулюється через java.lang.annotation.ElementTypeenum, який має константи для кожного з одинадцяти стандартних типів цілей. Отже, нами створено загальнодоступну одноелементну анотацію з назвою JField, яка зберігається JVM під час виконання і може бути застосовною лише до полів. Ця анотація має єдиний параметр, значення типу String зі значенням за замовчуванням порожнього рядка. Створивши анотацію, тепер можемо прокоментувати поля для серіалізації.

Використання анотації вимагає лише того, щоб її було розміщено перед відповідною конструкцією (будь-яка дійсна ціль для анотації). Наприклад, ми можемо створити клас Car з такими полями: String make, String model, String year.

У цьому класі є два основних варіанти застосування анотації @JField: *з явним значенням* та *зі значенням за замовчуванням*. Ми могли б також коментувати поле, скориставшись формою @JField (value = "someName"), але цей стиль надто багатослівний і не допомагає читабельності нашого коду. Тому приєднання імені параметра до одноелементної анотації не додає читабельності коду, а отже, ним можна знехтувати. Для анотацій із більш ніж одним параметром ім'я кожного параметра потрібно для диференціації між параметрами.

З огляду на зазначене під час використання анотації @JField ми очікуємо, що автомобіль буде серіалізовано в рядок JSON виду {"make": "someMake", "model": "someModel"}. Перш ніж продовжити, важливо зауважити, що додавання анотацій @JField не змінює поведінку класу Car під час виконання. Якщо ми скомпілюємо цей клас, приєднання анотацій @JField не покращить поведінку класу Car більше, ніж якби ми пропустили анотації. Ці анотації просто записуються разом із значенням параметра value у файл класу для класу Car. Зміна поведінки нашої системи під час виконання вимагає, щоб ми обробляли ці примітки.

Оброблення анотацій здійснюється за допомогою Java Reflection API. Завдяки Reflection API ми можемо писати код, який перевірятиме класи, методи, поля та інші цілі певного об'єкта. Наприклад, якщо створити метод, котрий приймає об'єкт Car, ми зможемо перевірити клас цього об'єкта (а саме, Car) і виявити, що цей клас має три поля: *make*, *model* та *year*. Крім того, ми матиме змогу перевірити ці поля, аби виявити, чи кожне з них коментується певною анотацією. Скориставшись цією можливістю, ми зможемо перебрати кожне поле класу, пов'язане з об'єктом, переданим нашому методу, і виявити, які з цих полів коментуються анотацією @JField. Якщо поле коментується анотацією @JField, ми записуємо назву поля та його значення. Після того, як усі поля буде оброблено, ми можемо створити рядок JSON, застосувавши ці імена та значення полів.

Визначення назви поля вимагає більш складної логіки, ніж визначення значення. Якщо @JField містить надане значення параметра (наприклад, make у попередньому використанні @JField), ми будемо послуговуватися цією наданою назвою поля. Якщо значення параметра value — це порожній рядок, ми знаємо, що ім'я поля явно не було надано (оскільки це значення за замовчуванням для параметра value), або ж явно було надано порожній рядок. У будь-якому разі ми використовуватимемо назву змінної поля як ім'я поля. І в результаті отриманої інформації з полів нашого класу з їхніми значеннями далі не буде складності для сформування JSON подання нашого об'єка.

Висновки

Анотації Java — це дуже потужна функція в мові Java, але найчастіше людина, яка працює з Java, є користувачем стандартних анотацій (зокрема @Override) або загальних анотацій із фреймворків (наприклад, @Autowired), а не розробником самих анотацій. Хоча анотації не слід використовувати замість інтерфейсів або інших мовних конструкцій, які належним чином виконують завдання об'єктно-орієнтованого підходу, вони можуть значно спростити повторювану логіку. Наприклад, замість того, щоб створювати метод toJsonString в інтерфейсі та мати всі класи, які можна серіалізувати, реалізувати цей інтерфейс, ми можемо анотувати кожне поле, що піддається серіалізації. Це бере повторювану логіку процесу серіалізації (зіставлення імен полів зі значеннями полів) і поміщає її

в єдиний клас серіалізатора. Він також відокремлює логіку серіалізації від логіки домену, усуваючи безлад ручної серіалізації з лаконічності логіки домену. Хоча користувацькі анотації не часто використовують у більшості програм Java, знання цієї функції є обов'язковою умовою для будь-якого середнього або обізнаного користувача мови Java. Знання цієї функції не тільки покращить набір інструментів розробника, що так само важливо, а й допоможе зрозуміти загальні анотації в найпопулярніших платформах Java.

Список використаної літератури

1. ***Head*** *First Java, 2nd Edition by Kathy Sierra, Bert Bates. 2005. С. 233–249.*

2. ***Java:*** *The Complete Reference, Eleventh Edition 11th Edition by Herbert Schildt. 2018. С. 301–323.*

3. ***Java*** *Professional Library by David Flanagan. 2000. С. 171–188.*

4. ***Java*** *Developer's Reference by Bryan Morgan, Michael Morrison, Michael T. Nygard, Dan Joshi, Tom Trinko, Mike Cohn. 1996. С. 201–212.*

5. ***Effective*** *Java 3rd Edition by Joshua Bloch. 2017. С. 99–111.*

6. ***Thinking*** *in Java 4th Edition by Bruce Eckel. 2006. С. 247–255.*

Д. Е. Алтынников, О. А. Шевченко, И. И. Бердник, А. В. Зуб, В. А. Сагайдак

ИСПОЛЬЗОВАНИЕ JAVA-АННОТАЦИЙ КАК ИНСТРУМЕНТА ПРЕДОСТАВЛЕНИЯ API

Java-аннотации являются мощной частью языка программирования, но в большинстве случаев их используют обычно пользователи некоторой логики, а не авторы аннотаций. Например, не трудно найти исходный код Java, содержащий аннотацию @Override, обработанную компилятором Java, аннотацию @Autowired, используемую фреймворком Spring, или аннотацию @Entity, используемую фреймворком Hibernate, но редко можно увидеть аннотации, написанные пользователем. Несмотря на то, что пользовательские аннотации являются аспектом не очень часто используемого языка Java, они могут быть достаточно полезным ресурсом при разработке кода, который приятно читать и одновременно лаконично достигать своих целей, поскольку это делают такие фреймовки, как Spring или Hibernate.

Ключевые слова: Java; JVM; аннотация.

D. E. Altynnikov, O. O. Shevchenko, I. I. Berdnyk, O. V. Zub, V. A. Sahaydak

USING JAVA ANNOTATIONS AS A TOOL FOR PROVIDING API

Although Java has such syntax instrument as annotations which are a strong part of that programming language, they are often employed by consumers of logic rather than by theirs' writers. A quite good example, it's not hard to discover from Java source code with the @Override, Java compiler processing, the @Autowired commonly used annotation for such framework as Spring, but user-written annotations are uncommon. Although custom annotations are a rare feature of the Java language, stench, like frameworks like Spring, may be a valuable resource for writing code that is easy to comprehend and accomplishes its goals quickly. Annotations in Java are a kind of labels in the code describing the metadata for a function / class / package. For example, the well-known @Override annotation, which means that we are going to override the method of the parent class. Yes, on the one hand, it is possible without it, but if the parents do not have this method, there is a possibility that we wrote the code in vain, because specifically, this method may never be called, and with the @Override annotation, the compiler will tell us that there was not found such a method in the parents something is unclean here. However, Annotations can carry not only the meaning for reliability, but they can also store some data that will then be used. Annotations as well can be described as descriptors included in the program text and are used to store the metadata of the program code required at different stages of the program life cycle. The information stored in annotations can be used by appropriate processors to create the necessary auxiliary files or to mark classes, fields, etc. Please note that the annotation itself does not affect the method override in any way, but it allows you to control the success of the override during compilation or assembly. We have protected a piece of code from an inconspicuous error that would take hours to find in a large program. This is just one of the many uses for annotations.

Keywords: Java; JVM; annotation.