DZone

Creating Annotations in Java

by Justin Albano MVB · Feb. 04, 18 · Java Zone

Annotations are a powerful part of Java, but most times we tend to be the users rather than the creators of annotations. For example, it is not difficult to find Java source code that includes the `@Override` annotation processed by the Java compiler, the `@Autowired` annotation used by the Spring framework, or the `@Entity` annotation used by the Hibernate framework; but rarely do we see custom annotations. While custom annotations are an often-overlooked aspect of the Java language, they can be a very useful asset in developing readable code and just as importantly, useful in understanding how many common frameworks, such as Spring or Hibernate, succinctly accomplish their goals.

In this article, we will cover the basics of annotations, including what annotations are, how they are useful in large-than-academic examples, and how to process them. In order to demonstrate how annotations work in practice, we will create a Javascript Object Notation (JSON) serializer that processes annotated objects and produces a JSON string representing each object. Along the way, we will cover many of the common stumbling blocks of annotations, including the quirks of the Java reflection framework and visibility concerns for annotation consumers. The interested reader can find the source code for the completed JSON serializer on GitHub.

What Are Annotations?

Annotations are decorators that are applied to Java constructs, such as classes, methods, or fields, that associate metadata with the construct. These decorators are benign and do not execute any code in-and-of-themselves, but can be used by runtime frameworks or the compiler to perform certain actions. Stated more formally, the Java Language Specification (JLS), Section 9.7, provides the following definition:

> **An *annotation* is a marker which associates information with a program construct, but has no effect at run time.**

It is important to note the last clause in this definition: Annotations have no effect on a program at runtime. This is not to say that a framework may not change its behavior based on the presence of an annotation at runtime, but that the inclusion of an annotation does not itself change the runtime behavior of a program. While this may appear to be a nuanced distinction, it is a very important one that must be understood in order to grasp the usefulness of annotations

understood in order to grasp the usefulness of annotations.

For example, adding the `@Autowired` annotation to an instance field does not in-and-of-itself change the runtime behavior of a program: The compiler simply includes the annotation at runtime, but the annotation does not execute any code or inject any logic that alters the normal behavior of the program (the behavior expected when the annotation is omitted). Once we introduce the Spring framework at runtime, we are able to gain powerful Dependency Injection (DI) functionality when our program is parsed. By including the annotation, we have instructed the Spring framework to inject an appropriate dependency into our field. We will see shortly (when we create our JSON serializer) that the annotation itself does not accomplish this, but rather, the annotation acts as a marker, informing the Spring framework that we desire a dependency to be injected into the annotated field.

Retention and Target

Creating an annotation requires two pieces of information: (1) a retention policy and (2) a target. A **retention policy** specifies how long, in terms of the program lifecycle, the annotation should be retained for. For example, annotations may be retained during compile-time or runtime, depending on the retention policy associated with the annotation. As of Java 9, there are three standard retention policies, as summarized below:

POLICY	DESCRIPTION
Source	Annotations are discarded by the compiler
Class	Annotations are recorded in the class file generated by the compiler but are not required to be retained by the Java Virtual Machine (JVM) that processes the class file at runtime
Runtime	Annotations are recorded in the class file by the compiler and retained at runtime by the JVM

As we will see shortly, the runtime option for annotation retention is one of the most common, as it allows for Java programs to reflectively access the annotation and execute code based on the presence of an annotation, as well as access the data associated with an annotation. Note that an annotation has *exactly one* associated retention policy.

The **target** of an annotation specifies which Java constructs an annotation can be applied to. For example, some annotations may be valid for methods only, while others may be valid for both classes and fields. As of Java 9, there are eleven standard annotation targets, as summarized in the following table:

TARGET	DESCRIPTION
Annotation Type	Annotates another annotation
Constructor	Annotates a constructor
Field	Annotates a field, such as an instance variable of a class or an enum constant
Local variable	Annotates a local variable

Method	Annotates a method of a class
Module	Annotates a module (new in Java 9)
Package	Annotates a package
Parameter	Annotates a parameter to a method or constructor
Type	Annotates a type, such as a class, interfaces, annotation types, or enum declarations
Type Parameter	Annotates a type parameter, such as those used as formal generic parameters
Type Use	Annotates the use of a type, such as when an object of a type is created using the `new` keyword, when an object is cast to a specified type, when a class implements an interface, or when the type of a throwable object is declared using the throws keyword (for more information, see the Type Annotations and Pluggable Type Systems Oracle tutorial)

For more information on these targets, see Section 9.7.4 of the JLS. It is important to note that *one or more* targets may be associated with an annotation. For example, if the field and constructor targets are associated with an annotation, then the annotation may be used on either fields or constructors. If on the other hand, an annotation only has an associated target of method, then applying the annotation to any construct other than a method results in an error during compilation.

Annotation Parameters

Annotations may also have associated parameters. These parameters may be a primitive (such as `int` or `double`), `String`, class, enum, annotation, or an array of any of the five preceding types (see Section 9.6.1 of the JLS). Associating parameters with an annotation allows for an annotation to provide contextual information or can parameterize a processor of an annotation. For example, in our JSON serializer implementation, we will allow for an optional annotation parameter that specifies the name of a field when it is serialized (or use the variable name of the field by default if no name is specified).

How Are Annotations Created?

For our JSON serializer, we will create a field annotation that allows a developer to mark a field to be included when serializing an object. For example, if we create a car class, we can annotate the fields of the car (such as make and model) with our annotation. When we serialize a car object, the resulting JSON will include make and model keys, where the values represent the value of the make and model fields, respectively. For the sake of simplicity, we will assume that this annotation will be used only for fields of type `String`, ensuring that the value of the field can be directly serialized as a string.

To create such a field annotation, we declare a new annotation using the `@interface` keyword:

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.FIELD)
3  public @interface JsonField {
4      public String value() default "";
   }
```

```
}
```

The core of our declaration is the `public @interface JsonField`, which declares an annotation type with a `public` modifier, allowing our annotation to be used in any package (assuming the package is properly imported if in another module). The body of the annotation declares a single `String` parameter, named `value`, that has a type of `String` and a default value of an empty string.

Note that the variable name `value` has a special meaning: It defines a Single-Element Annotation (Section 9.7.3. of the JLS) and allows users of our annotation to supply a single parameter to the annotation without specifying the name of the parameter. For example, a user can annotate a field using `@JsonField("someFieldName")` and is not *required* to declare the annotation as `@JsonField(value = "someFieldName")`, although the latter may still be used (but it is not required). The inclusion of a default value of empty string allows for the value to be omitted, resulting in `value` holding an empty string if no value is explicitly specified. For example, if a user declares the above annotation using the form `@JsonField`, then the `value` parameter is set to an empty string.

The retention policy and target of the annotation declaration are specified using the `@Retention` and `@Target` annotations, respectively. The retention policy is specified using the `java.lang.annotation.RetentionPolicy` enum and includes constants for each of the three standard retention policies. Likewise, the target is specified using the `java.lang.annotation.ElementType` enum, which includes constants for each of the eleven standard target types.

In summary, we created a public, single-element annotation named `JsonField`, which is retained by the JVM during runtime and may only be applied to fields. This annotation has a single parameter, `value`, of type `String` with a default value of an empty string. With our annotation created, we can now annotate fields to be serialized.

How Are Annotations Used?

Using an annotation requires only that the annotation is placed before an appropriate construct (any valid target for the annotation). For example, we can create a `Car` class using the following class declaration:

```
public class Car {

    @JsonField("manufacturer")
    private final String make;

    @JsonField
    private final String model;

    private final String year;

    public Car(String make, String model, String year) {
        this.make = make;
        this.model = model;
```

```
        this.year = year;
    }

    public String getMake() {
        return make;
    }

    public String getModel() {
        return model;
    }

    public String getYear() {
        return year;
    }

    @Override
    public String toString() {
        return year + " " + make + " " + model;
    }
}
```

This class exercises the two major uses of the `@JsonField` annotation: (1) with an explicit value and (2) with a default value. We could have also annotated a field using the form `@JsonField(value = "someName")`, but this style is overly verbose and does not aid in the readability of our code. Therefore, unless the inclusion of an annotation parameter name *in a single-element annotation* adds to the readability of code, it should be omitted. For annotations with more than one parameter, the name of each parameter is required to differentiate between parameters (unless only one argument is provided, in which case, the argument is mapped to the `value` parameter if no name is explicitly provided).

Given the above uses of the `@JsonField` annotation, we would expect that a `Car` ject is serialized into a JSON string of the form `{"manufacturer":"someMake", "model":"someModel"}` (note, as we will see later, we will disregard the order of the keys– `manufacturer` and `model` –in this JSON string). Before we proceed, it is important to note that adding the `@JsonField` annotations does not change the runtime behavior of the `Car` class. If we compile this class, the inclusion of `@JsonField` annotations does not enhance the behavior of the `Car` class anymore than had we omitted the annotations. These annotations are simply recorded, along with the value of the `value` parameter, in the class file for the `Car` class. Altering the runtime behavior of our system requires that we process these annotations.

How are Annotations Processed?

Processing annotations is accomplished through the Java Reflection Application Programming Interface (API). Sidelining the technical nature of the reflection API for a moment, the reflection API allows us to write code that will inspect the class, methods, fields, etc. of an object. For example, if we create a method

that accepts a `Car` object, we can inspect the class of this object (namely, `Car`) and discover that this class has three fields: (1) `make`, (2) `model`, and (3) `year`. Furthermore, we can inspect these fields to discover if each is annotated with a specific annotation.

Using this capability, we can iterate through each field of the class associated with the object passed to our method and discover which of these fields are annotated with the `@JsonField` annotation. If the field is annotated with the `@JsonField` annotation, we record the name of the field and its value. Once all the fields have been processed, then we can create the JSON string using these field names and values.

Determining the name of the field requires more complex logic than determining the value. If the `@JsonField` includes a provided value for the value parameter (such as `"manufacturer"` in the previous `@JsonField("manufacturer")` use), we will use this provided field name. If the value of the value parameter is an empty string, we know that no field name was explicitly provided (since this is the default value for the value parameter), or else, an empty string was explicitly provided. In either case, we will use the variable name of the field as the field name (for example, `model` in the `private final String model` declaration).

Combining this logic into a `JsonSerializer` class, we can create the following class declaration:

```
public class JsonSerializer {

    public String serialize(Object object) throws JsonSerializeException {

        try {
            Class<?> objectClass = requireNonNull(object).getClass();
            Map<String, String> jsonElements = new HashMap<>();

            for (Field field: objectClass.getDeclaredFields()) {
                field.setAccessible(true);
                if (field.isAnnotationPresent(JsonField.class)) {
                    jsonElements.put(getSerializedKey(field), (String) field.get(obje
                }
            }
            System.out.println(toJsonString(jsonElements));
            return toJsonString(jsonElements);
        }
        catch (IllegalAccessException e) {
            throw new JsonSerializeException(e.getMessage());
        }
    }

    private String toJsonString(Map<String, String> jsonMap) {
        String elementsString = jsonMap.entrySet()
                .stream()
                .map(entry -> "\""  + entry.getKey() + "\":\"" + entry.getValue() + "
```

```
                .collect(joining(","));
        return "{" + elementsString + "}";
    }

    private static String getSerializedKey(Field field) {
        String annotationValue = field.getAnnotation(JsonField.class).value();

        if (annotationValue.isEmpty()) {
            return field.getName();
        }
        else {
            return annotationValue;
        }
    }
}
```

Note that multiple responsibilities have been combined into this class for the sake of brevity. For a refactored version of this serializer class, see this branch in the codebase repository. We also create an exception that will be used to denote if an error has occurred while processing the object supplied to our serialize method:

```
public class JsonSerializeException extends Exception {

    private static final long serialVersionUID = -8845242379503538623L;

    public JsonSerializeException(String message) {
        super(message);
    }
}
```

Although the `JsonSerializer` class appears complex, it consists of three main tasks: (1) finding all fields of the supplied class annotated with the `@JsonField` annotation, (2) recording the field name (or the explicitly provided field name) and value for all fields that include the `@JsonField` annotation, and (3) converting the recorded field name and value pairs into a JSON string.

The line `requireNonNull(object).getClass()` simply checks that the supplied object is not `null` (and throws a `NullPointerException` if it is) and obtains the `Class` object associated with the supplied object. We will use this `Class` object shortly to obtain the fields associated with the class. Next, we create a `Map` of `String`s to `String`s, which will be used store the field name and value pairs.

With our data structures established, we next iterate through each field declared in the class of the supplied object. For each field, we configure the field to suppress Java language access checking when accessing the field. This is a very important step since the fields we annotated are private. In the standard

accessing the field. This is a very important step since the fields we annotated are private. In the standard case, we would be unable to access these fields, and attempting to obtain the value of the private field would result in an `IllegalAccessException` being thrown. In order to access these private fields, we must instruct the reflection API to suppress the standard Java access checking for this field using the `setAccessible` method. The `setAccessible(boolean)` documentation defines the meaning of the supplied boolean flag as follows:

A value of `true` indicates that the reflected object should suppress Java language access checking when it is used. A value of `false` indicates that the reflected object should enforce Java language access checks.

Note that with the introduction of modules in Java 9, using the `setAccessible` method requires that the package containing the class whose private fields will be accessed should be declared open in its module definition. For more information, see this explanation by Michał Szewczyk and Accessing Private State of Java 9 Modules by Gunnar Morling.

After gaining access to the field, we check if the field is annotated with the `@JsonField`. If it is, we determine the name of the field (either through an explicit name provided in the `@JsonField` annotation or the default name, which equals the variable name of the field) and record the name and field value in our previously constructed map. Once all fields have been processed, we then convert the map of field names to field values (`jsonElements`) into a JSON string.

We accomplish by converting the map into a stream of entries (key-value pairs for each entry in the map), mapping each entry to a string of the form `"<fieldName>":"<fieldValue>"`, where `<fieldName>` is the key for the entry and `<fieldValue>` is the value for the entry. Once all entries have been processed, we combine all of these entry strings with a comma. This results in a string of the form `"<fieldName1>":"<fieldValue1>","<fieldName2>":"<fieldValue2>",...`. Once this terminal string has been joined, we surround it with curly braces, creating a valid JSON string.

In order to test this serializer, we can execute the following code:

```
1  Car car = new Car("Ford", "F150", "2018");
2  JsonSerializer serializer = new JsonSerializer();
3  serializer.serialize(car);
```

This results in the following output:

```
1  {"model":"F150","manufacturer":"Ford"}
```

As expected, the `maker` and `model` fields of the `Car` object have been serialized, using the name of the field (or the explicitly supplied name in the case of the `maker` field) as the key and the value of the field as

the value. Note that the order of JSON elements may be reversed from the output seen above. This occurs because there is no definite ordering for the array of declared fields for a class, as stated in the `getDeclaredFields` documentation:

> **The elements in the returned array are not sorted and are not in any particular order.**

Due to this limitation, the order of the elements in the JSON string may vary. To make the order of the elements deterministic, we would have to impose ordering ourselves (such as by sorting the map of field names to field values). Since a JSON object is defined as an *unordered* set of name-value pairs, as per the JSON standard, imposing ordering is unneeded. Note, however, a test case for the serialize method should pass for either `{"model":"F150","manufacturer":"Ford"}` or `{"manufacturer":"Ford","model":"F150"}`.

Conclusion

Java annotations are a very powerful feature in the Java language, but most often, we are the users of standard annotations (such as `@Override`) or common framework annotations (such as `@Autowired`), rather than their developers. While annotations should not be used in place of interfaces or other language constructs that properly accomplish a task in an object-oriented manner, they can greatly simplify repetitive logic. For example, rather than creating a `toJsonString` method within an interface and having all classes that can be serialized implement this interface, we can annotate each serializable field. This takes the repetitive logic of the serialization process (mapping field names to fields values) and places it into a single serializer class. It also decouples the serialization logic from the domain logic, removing the clutter of manual serialization from the conciseness of the domain logic.

While custom annotations are not frequently used in most Java applications, knowledge of this feature is a requirement for any intermediate or advanced user of the Java language. Not only will knowledge of this feature enhance the toolbox of a developer, just as importantly, it will aid in the understanding of the common annotations in the most popular Java frameworks.

Like This Article? Read More From DZone