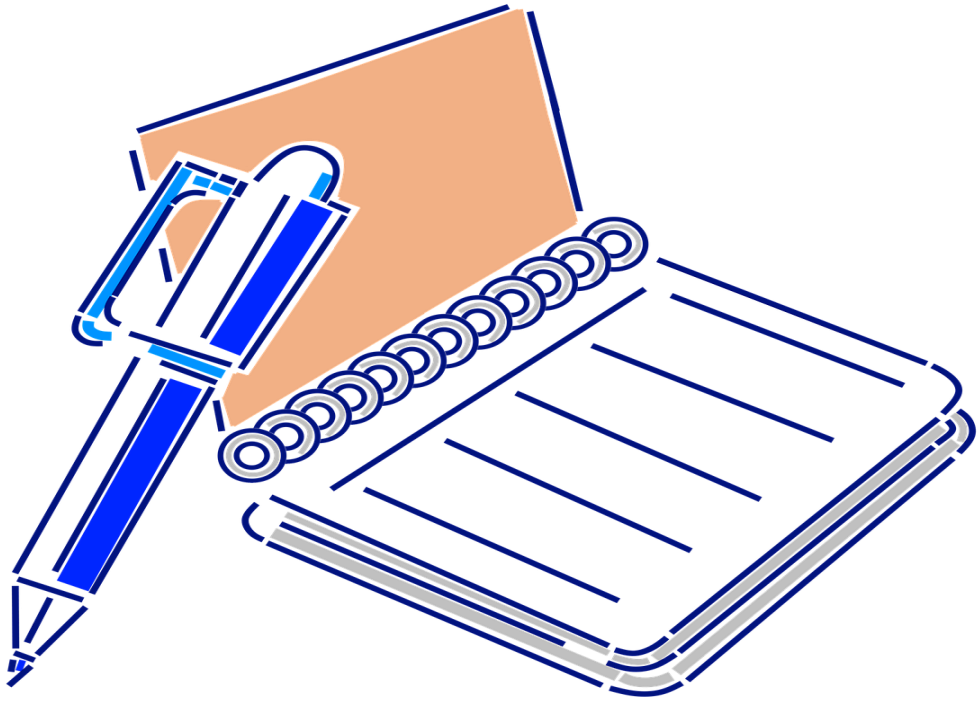

WORKING WITH ANNOTATIONS AND THE REFLECTION API

OBJECT ORIENTED PROGRAMMING I

Sercan Külcü | Object Oriented Programming I | 10.01.2023

Contents

Introduction

Annotations are a powerful feature in Java that allow you to add metadata to your code. They can be used to provide information about classes, methods, fields, and other elements of your code. Annotations can be used for a variety of purposes, including documentation, code generation, and runtime configuration.

Annotations

Annotations are defined using the @ symbol followed by the name of the annotation. Annotations can include parameters, which are specified using parentheses after the annotation name. For example:

```
@MyAnnotation(parameter1 = "value1", parameter2 = 123)
public void myMethod() {
    // ...
}
```

To use annotations, you can either use the pre-defined annotations provided by Java, or you can define your own custom annotations. Custom annotations are defined using the @interface keyword.

Annotations can be processed at compile time or at runtime. To process annotations at compile time, you can use the Java Compiler API or an annotation processing tool like the Annotation Processing Tool (APT). To process annotations at runtime, you can use the Reflection API.

Reflection API

The Reflection API is a powerful feature in Java that allows you to inspect and manipulate objects at runtime. With the Reflection API, you can access information about classes, methods, fields, and other elements of your code.

To use the Reflection API, you need to obtain a Class object that represents the class you want to inspect. You can obtain a Class object using the class literal syntax, for example:

```
Class<MyClass> myClass = MyClass.class;
```

Once you have a Class object, you can use it to access information about the class, such as its methods and fields. You can also use the Reflection API to instantiate objects, invoke methods, and access fields.

Working with Annotations and Reflection

Annotations and the Reflection API can be used together to provide powerful runtime configuration options. For example, you can define custom annotations that specify the configuration of a class or method, and then use the Reflection API to process those annotations and apply the configuration at runtime.

To process annotations using the Reflection API, you can use the getAnnotations() method to obtain an array of annotations applied to a class or method. You can then use the annotation values to perform any required configuration.

For example, consider the following custom annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyAnnotation {
    String value();
}
```

You can apply this annotation to a method in your code, like this:

```
@MyAnnotation("myValue")
public void myMethod() {
    // ...
```

```
}
```

To process this annotation at runtime, you can use the Reflection API to obtain the MyAnnotation object and extract its value:

```
Method method = MyClass.class.getMethod("myMethod");

MyAnnotation annotation =
method.getAnnotation(MyAnnotation.class);

String value = annotation.value();
```

Conclusion

Annotations and the Reflection API are powerful features in Java that allow you to add metadata to your code and manipulate objects at runtime. By mastering these features, you can create more flexible and configurable applications. However, it's important to use these features judiciously and carefully, as they can add complexity and overhead to your code.