Applause from Michelé Clarke, Elvis Chidera, and 119 others

Adrian D. Finlay
@thewiprogrammer. Writer @hackernoon. Code, LOTS of it. Mangos, LOVE THEM! Barbering. Health. Travel. Business. & more! Network w/ me @ adriandavid.me/network
Oct 22, 2017 · 9 min read

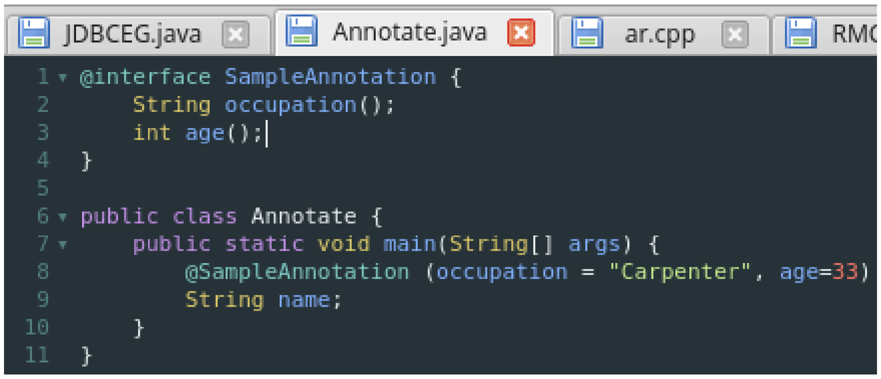
How well do you actually understand Annotations in Java?

Link:Homer Simpson

If you are a web or enterprise developer, which most java developers are, you consume annotations **all the time**. Whether in Spring, JEE, or Struts, you see them a lot. If you make use of unit testing tools like JUnit the same applies. If you do thick client or Android development, you may not see it as often in actual production code but rather your encounters with them will probably be with production tools, build tools, and testing. If you make use of Contexts & Dependency Injection then annotations are your mainstay.

Sure, you consume them. But do you know how to do basic things like **how to define one**? Maybe you do. But in my estimation, for the most part, if you are not developing APIs or deployment tools, you may rarely need to define an annotation. I think Annotations are the Java language element that is more commonly not fully understood. Maybe I'm wrong, but that's just my opinion. Why don't we start here.

Basic Annotation definition



```
1  @interface SampleAnnotation {
2      String occupation();
3      int age();
4  }
5
6  public class Annotate {
7      public static void main(String[] args) {
8          @SampleAnnotation (occupation = "Carpenter", age=33)
9          String name;
10     }
11 }
```

Notice the syntax of the annotation definition with '**@interface**' denoting the type. This is how the compiler recognizes an annotation type. This syntax is because Annotation types are based of off the same plumbing behind the interface. Also notice the method declarations. You don't implement these as you would do in a normal interface. The compiler does. Also, the methods() are (for all concern to the applications developer) treated like fields when using them. Notice Ln. 8. We simply say age = 33 as if age were a ordinary member and not a method.

Although it is not obvious from the source code, the annotation used on Ln. 8 leaves the program unchanged. Recall that annotations are merely metadata, they do not change the nature of a program (at least not directly). Annotations, can signal to development tools to do a certain action based on the type of annotation. One good example of this is the CDI Engine of Java EE. What then happens as a consequence is that the CDI run-time provides X based on X annotation but the actual source file itself, still, does not change.

Restrictions on Annotation Types [1]

- Annotations cannot participate in inheritance.
- Annotations methods may not have arguments.
- Annotations cannot be generic or specify a throws clause.

- Annotations must return: **an enum or primitive type or an annotation, String, or Class object.** They can also return an **array** of these types.

The Default Annotations

The are several annotations used in Java code defined and used across the standard library and third party providers.

The package **java.lang** defined in the Java SE API [2], provides 6 standard annotations. By consequence of being included in java.lang, they are automatically imported in every Java program. They are:

- @Deprecated—"A program element annotated `@Deprecated` is one that programmers are discouraged from using." [2]
- @FunctionalInterface—"An informative annotation type used to indicate that an interface type declaration is intended to be a *functional interface* as defined by the Java Language Specification." [2]
- @Override—"Indicates that a method declaration is intended to override a method declaration in a supertype." [2]
- @SafeVarags—"A programmer assertion that the body of the annotated method or constructor does not perform potentially unsafe operations on its varargs parameter." [2]
- @SupressWarnings—"Indicates that the named compiler warnings should be suppressed in the annotated element (and in all program elements contained in the annotated element)." [2]

The package **java.lang.annotation** as defined in the Java SE API [2], provides 6 standard annotations. They are:

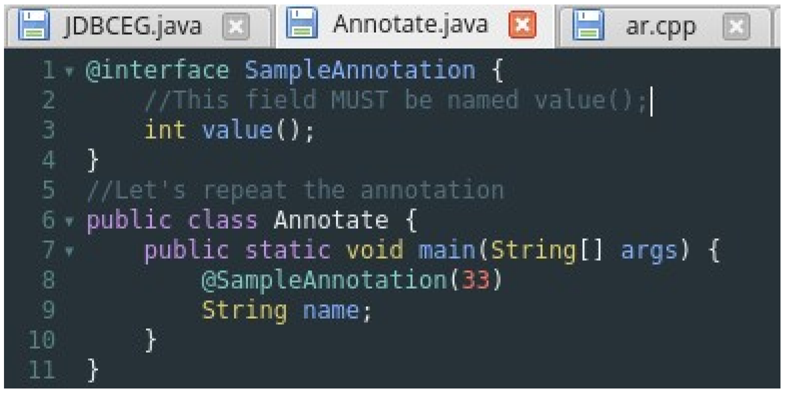
- @Documented—"If the annotation `@Documented` is present on the declaration of an annotation type *A*, then any `@A` annotation on an element is considered part of the element's public contract." [2] **Used to mark another annotation.**
- @Inherited —"Indicates that an annotation type is automatically inherited." [2] **Used to mark another annotation.**

- @Native—“Indicates that a field defining a constant value may be referenced from native code.” [2]
- @Repeatable—“The annotation type `java.lang.annotation.Repeatable` is used to indicate that the annotation type whose declaration it (meta-)annotates is *repeatable*.” [2] **Used to mark another annotation.**
- @Retention— “Indicates how long annotations with the annotated type are to be retained.” [2] **Used to mark another annotation.**
- @Target—“Indicates the contexts in which an annotation type is applicable.” [2] **Used to mark annother annotation.**

I will not go into depth with these annotations. They are, however, the annotations that are commonly seen in standard Java SE development.

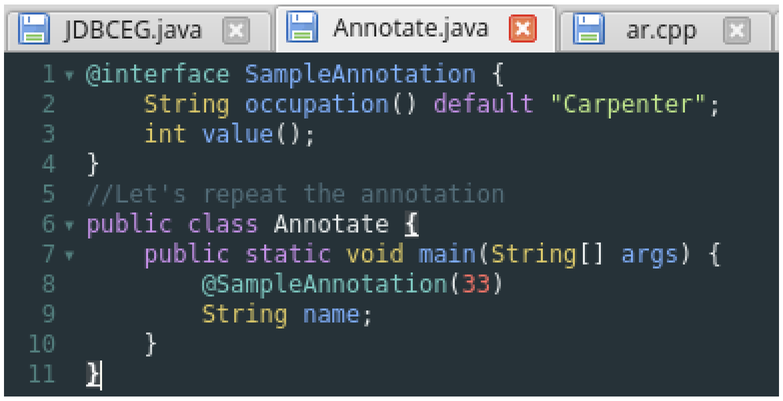
Single-Member Annotations, Default values

Put simply, **single-member annotations** are annotations with only one member. That single member must be named **value()**. Notice how the value is assigned using the parentheses on the annotation similar to a method argument.

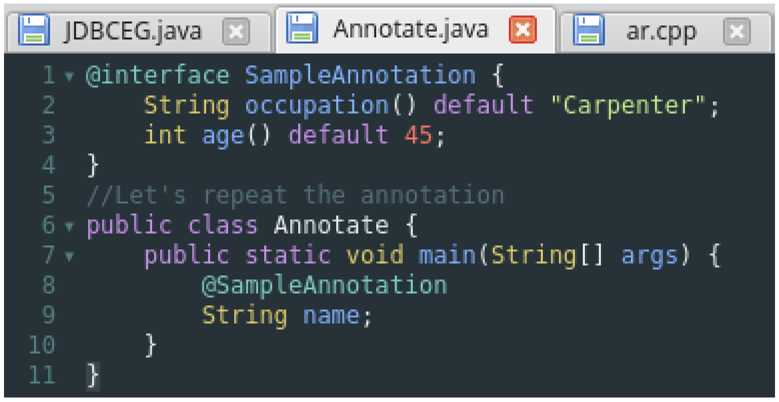


```
@interface SampleAnnotation {
    //This field MUST be named value();
    int value();
}
//Let's repeat the annotation
public class Annotate {
    public static void main(String[] args) {
        @SampleAnnotation(33)
        String name;
    }
}
```

***Default values** in action*



```java
@interface SampleAnnotation {
    String occupation() default "Carpenter";
    int value();
}
//Let's repeat the annotation
public class Annotate {
    public static void main(String[] args) {
        @SampleAnnotation(33)
        String name;
    }
}
```



```java
@interface SampleAnnotation {
    String occupation() default "Carpenter";
    int age() default 45;
}
//Let's repeat the annotation
public class Annotate {
    public static void main(String[] args) {
        @SampleAnnotation
        String name;
    }
}
```

Notice two things. The first is that because occupation() has a default value, we need not specify a value for it when using the @SampleAnnotation annotation, in the first example. In the second example, notice that because needn't specify a value at all.

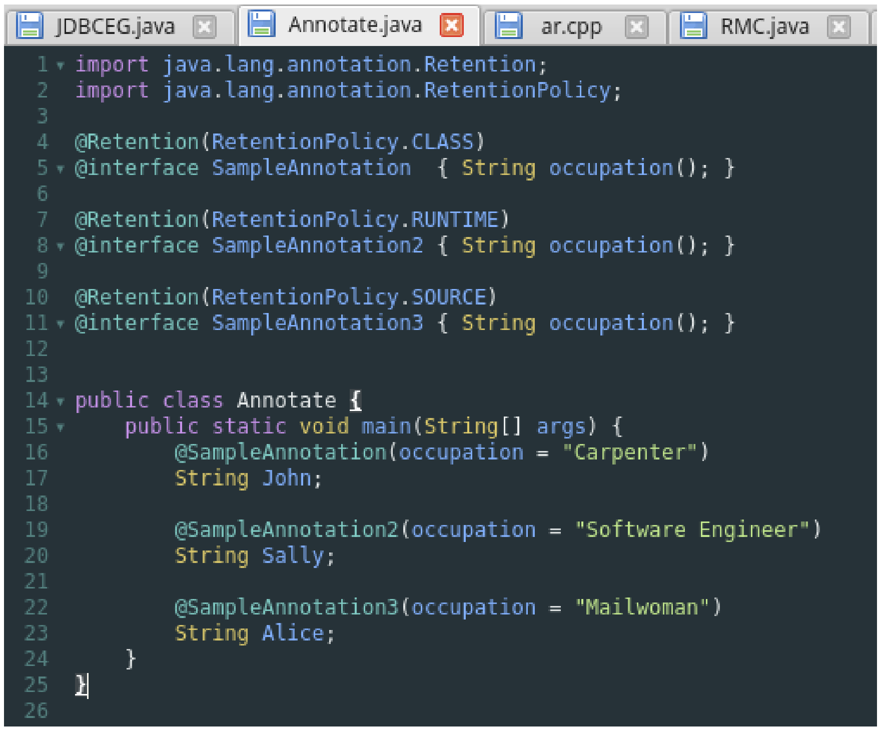
Retention Policies

Retention policies are key concept to understand in using Annotations. There are three retention policies: **CLASS**, **RUNTIME**,& **SOURCE**. [2] They are defined in **java.lang.annotation.RetentionPolicy**. **Annotation retention policies specify how long an Annotation is to be retained**. They are specified with the @Retention annotation. The default retention policy is CLASS. They are, as follows:

- **CLASS**: "Annotations are to be recorded in the class file by the compiler but need not be retained by the VM at run time." [2]
- **RUNTIME:** "Annotations are to be recorded in the class file by the compiler and retained by the VM at run time, so they may be read reflectively." [2]

- **SOURCE**: “Annotations are to be discarded by the compiler.” [2]

An Example



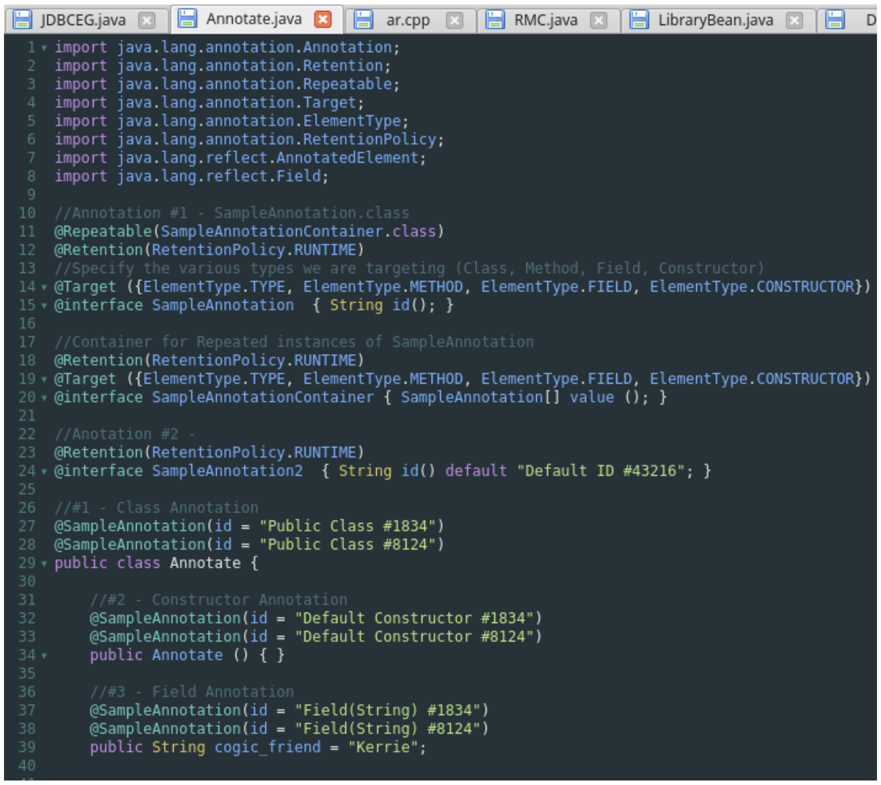
```java
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.CLASS)
@interface SampleAnnotation  { String occupation(); }

@Retention(RetentionPolicy.RUNTIME)
@interface SampleAnnotation2 { String occupation(); }

@Retention(RetentionPolicy.SOURCE)
@interface SampleAnnotation3 { String occupation(); }


public class Annotate {
    public static void main(String[] args) {
        @SampleAnnotation(occupation = "Carpenter")
        String John;

        @SampleAnnotation2(occupation = "Software Engineer")
        String Sally;

        @SampleAnnotation3(occupation = "Mailwoman")
        String Alice;
    }
}

```

Some readers may immediately notice that the ability for an annotation to persist until run-time allows for the use of Reflection to use the data. This is a key ability in the usefulness of annotations. We will briefly touch on this next. Also note that Annotations on **local variable declarations** (not initialized variables, just declarations) are not persisted in Java bytecode (a .class file).

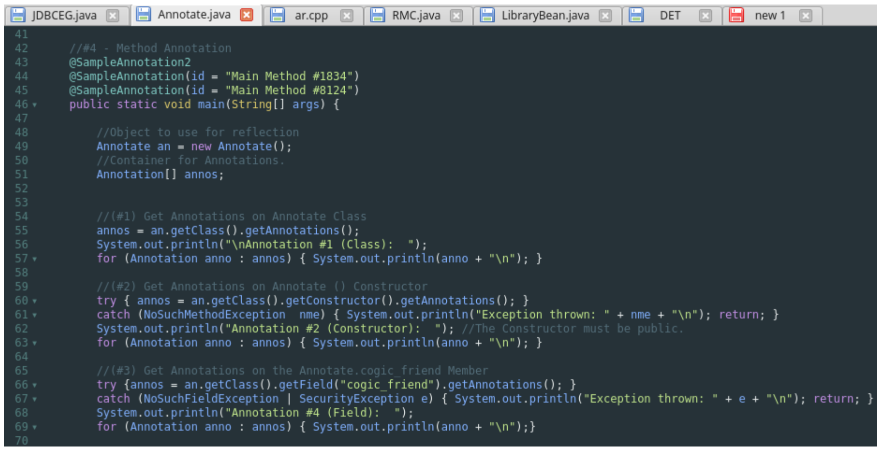
Annotation Information at Runtime—Reflection

As we have just touched on, an annotations data may be obtained at runtime through the use of reflection. While we will not discuss the full extent of what you can do with this, I will provide a brief example demonstrating it’s use. Although we did not make use of it, the AnnotatedElement interface provides some useful methods for performing reflection on Annotations at runtime. In the end, I did not directly use AnnotatedElement and Field(Ln. 7,8) from java.lang.reflect.

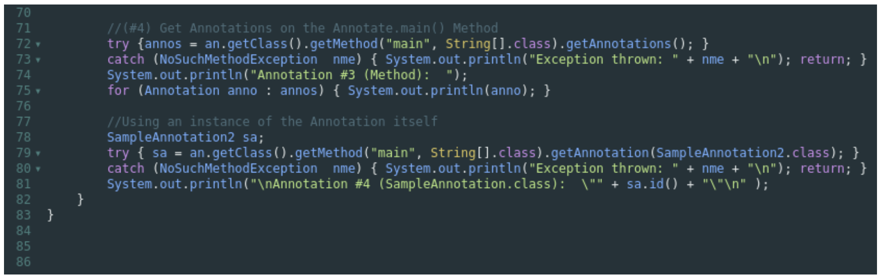
The Code



```java
import java.lang.annotation.Annotation;
import java.lang.annotation.Retention;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
import java.lang.annotation.RetentionPolicy;
import java.lang.reflect.AnnotatedElement;
import java.lang.reflect.Field;

//Annotation #1 - SampleAnnotation.class
@Repeatable(SampleAnnotationContainer.class)
@Retention(RetentionPolicy.RUNTIME)
//Specify the various types we are targeting (Class, Method, Field, Constructor)
@Target ({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD, ElementType.CONSTRUCTOR})
@interface SampleAnnotation  { String id(); }

//Container for Repeated instances of SampleAnnotation
@Retention(RetentionPolicy.RUNTIME)
@Target ({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD, ElementType.CONSTRUCTOR})
@interface SampleAnnotationContainer { SampleAnnotation[] value (); }

//Anotation #2 -
@Retention(RetentionPolicy.RUNTIME)
@interface SampleAnnotation2  { String id() default "Default ID #43216"; }

//#1 - Class Annotation
@SampleAnnotation(id = "Public Class #1834")
@SampleAnnotation(id = "Public Class #8124")
public class Annotate {

    //#2 - Constructor Annotation
    @SampleAnnotation(id = "Default Constructor #1834")
    @SampleAnnotation(id = "Default Constructor #8124")
    public Annotate () { }

    //#3 - Field Annotation
    @SampleAnnotation(id = "Field(String) #1834")
    @SampleAnnotation(id = "Field(String) #8124")
    public String cogic_friend = "Kerrie";

```

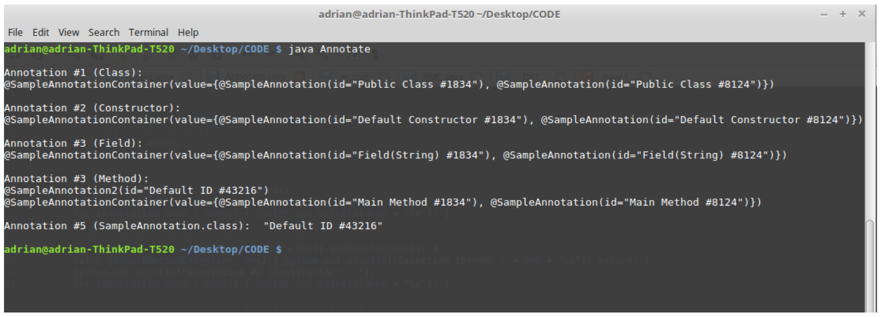


```java

    //#4 - Method Annotation
    @SampleAnnotation2
    @SampleAnnotation(id = "Main Method #1834")
    @SampleAnnotation(id = "Main Method #8124")
    public static void main(String[] args) {

        //Object to use for reflection
        Annotate an = new Annotate();
        //Container for Annotations.
        Annotation[] annos;


        //(#1) Get Annotations on Annotate Class
        annos = an.getClass().getAnnotations();
        System.out.println("\nAnnotation #1 (Class):  ");
        for (Annotation anno : annos) { System.out.println(anno + "\n"); }

        //(#2) Get Annotations on Annotate () Constructor
        try { annos = an.getClass().getConstructor().getAnnotations(); }
        catch (NoSuchMethodException  nme) { System.out.println("Exception thrown: " + nme + "\n"); return; }
        System.out.println("Annotation #2 (Constructor):  "); //The Constructor must be public.
        for (Annotation anno : annos) { System.out.println(anno + "\n"); }

        //(#3) Get Annotations on the Annotate.cogic_friend Member
        try {annos = an.getClass().getField("cogic_friend").getAnnotations(); }
        catch (NoSuchFieldException | SecurityException e) { System.out.println("Exception thrown: " + e + "\n"); return; }
        System.out.println("Annotation #4 (Field):  ");
        for (Annotation anno : annos) { System.out.println(anno + "\n");}

```



```java

        //(#4) Get Annotations on the Annotate.main() Method
        try {annos = an.getClass().getMethod("main", String[].class).getAnnotations(); }
        catch (NoSuchMethodException  nme) { System.out.println("Exception thrown: " + nme + "\n"); return; }
        System.out.println("Annotation #3 (Method):  ");
        for (Annotation anno : annos) { System.out.println(anno); }

        //Using an instance of the Annotation itself
        SampleAnnotation2 sa;
        try { sa = an.getClass().getMethod("main", String[].class).getAnnotation(SampleAnnotation2.class); }
        catch (NoSuchMethodException  nme) { System.out.println("Exception thrown: " + nme + "\n"); return; }
        System.out.println("\nAnnotation #4 (SampleAnnotation.class):  \"" + sa.id() + "\"\n" );
    }
}



```

The Output


adrian@adrian-ThinkPad-T520 ~/Desktop/CODE
File Edit View Search Terminal Help

```
adrian@adrian-ThinkPad-T520 ~/Desktop/CODE $ java Annotate

Annotation #1 (Class):
@SampleAnnotationContainer(value={@SampleAnnotation(id="Public Class #1834"), @SampleAnnotation(id="Public Class #8124")})

Annotation #2 (Constructor):
@SampleAnnotationContainer(value={@SampleAnnotation(id="Default Constructor #1834"), @SampleAnnotation(id="Default Constructor #8124")})

Annotation #3 (Field):
@SampleAnnotationContainer(value={@SampleAnnotation(id="Field(String) #1834"), @SampleAnnotation(id="Field(String) #8124")})

Annotation #3 (Method):
@SampleAnnotation2(id="Default ID #43216")
@SampleAnnotationContainer(value={@SampleAnnotation(id="Main Method #1834"), @SampleAnnotation(id="Main Method #8124")})

Annotation #5 (SampleAnnotation.class):  "Default ID #43216"

adrian@adrian-ThinkPad-T520 ~/Desktop/CODE $
```

What we did was:

1. We made three annotations. SampleAnnotation & it's Container Annotation called SampleAnnotationContainer (for repeated instance of SampleAnnotation)& SampleAnnotation2. We specified an array of values of type ElementType in the target of the SampleAnnotation as well as it's container Annotation. We created RUNTIME annotations.

2. We created a class called Annotate and we annotated the class as well as the class's default constructor, main method, and a field in the class. Only one element was annotated with SampleAnnotation2 which was the main() method.

3. Inside the main() method we instantiated an object of the enclosing class to use for the basis of reflection and an declared an array of type Annotation to hold the annotations.

4. We made five attempts to obtain annotations. The generic step for the first four were to instantiate an array of type Annotation with the annotations on that specific element as provided by the Class API.

5. There are many ways to do it, but we did it as follows: 1) Create an object of the class we are interested in. 2)Call Object.getClass() to get a Class<> object with which we can perform reflection. 3)Call the getConstructor(), getMethod(), getField(), to get a more specific reference to the element with which we want to perform reflection. This step was skipped for the first attempt at reflection, which was on the class itself. 4) Call the Class.getAnnotations() method on the returned object to get the annotations of that object and assign that value to a variable of type Annotation []. 5) We

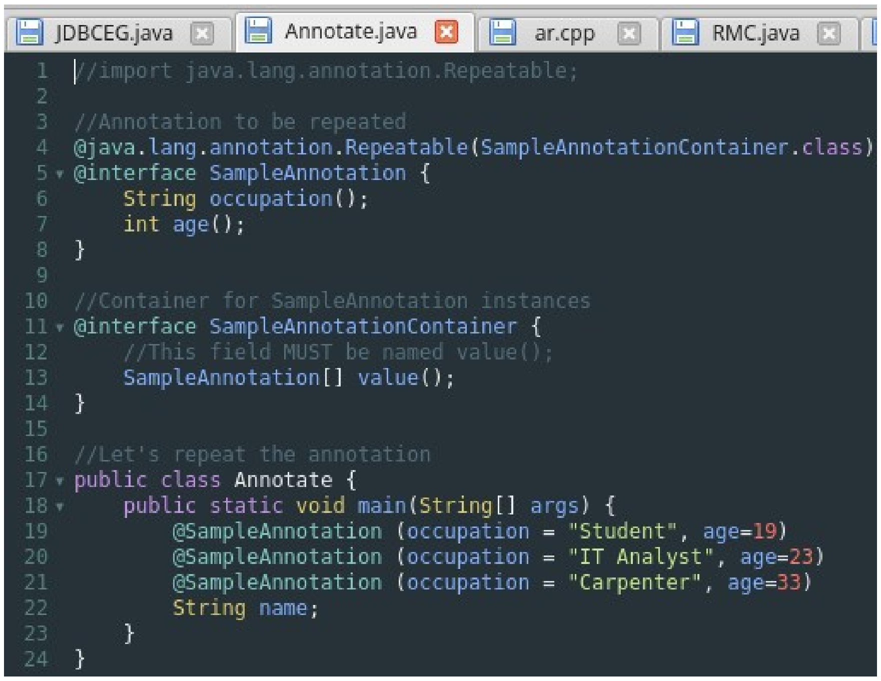
handled exceptions, if necessary. 6)We did some formatting and cycled through the array of type Annotation and printed the elements.

6. We used a reference of our annotation type itself to perform reflection. This was a reference to SampleAnnotation2. We instantiated an instance of SampleAnnotation2 using the same method described above except we called getAnnotation(SampleAnnotation2.class) instead of getAnnotations(). To print the contents we called the method id() of the Annotation. Because we did not specify a value to id() when we used the annotation when we called our object of type SampleAnnotation2 on id() it returned the default value which is the String “Default ID #43216".

Other things you can do with Annotations

There are other things that one can do with annotations and while we will not discuss them in depth here they are worth mentioning so that you may discover them on your own.

> *As it has been alluded to with the presence of the @Repeatable annotation,* ***annotations can be repeated****. For example:*



JDBCEG.java Annotate.java ar.cpp RMC.java

```java
//import java.lang.annotation.Repeatable;

//Annotation to be repeated
@java.lang.annotation.Repeatable(SampleAnnotationContainer.class)
@interface SampleAnnotation {
    String occupation();
    int age();
}

//Container for SampleAnnotation instances
@interface SampleAnnotationContainer {
    //This field MUST be named value();
    SampleAnnotation[] value();
}

//Let's repeat the annotation
public class Annotate {
    public static void main(String[] args) {
        @SampleAnnotation (occupation = "Student", age=19)
        @SampleAnnotation (occupation = "IT Analyst", age=23)
        @SampleAnnotation (occupation = "Carpenter", age=33)
        String name;
    }
}
```

The steps are essentially:
1) Create the annotation to repeat.
2) Mark the annotation to be repeated with @Repeatable(x.class) where x is a container to hold the instances of the annotation that is to be repeated.
3) Create the container for holding the instances of the repeatable annotation
4) Create and array of the annotation that is to be repeated and name the variable value(). It **must** include a method specifically named **value().**
5) Annotate to your hearts content.
6) Call getAnnotation() on the container annotation to get the instances of the annotation instead of calling getAnnotation() on the original annotation itself.

> *As has been discussed earlier, as of JDK 8 one may annotate types. This notion is referred to as **Type Annotations**.*

Type annotations must include "@Target (ElementType.TYPE_USE)" in their definition so the compiler understands it as a Type Annotation. Type annotations are useful with the use of compiler plugins for source code tools, build tools, & development tools. I talk a lot more about this, including how to annotate a Method's receiver (implicit this argument), below.

The Java 8 Feature You Never Knew About

Released on March 18, 2014, Java 8 was generally considered a major feature update, which broug...

medium.com

You can also create **Marker Annotations**. A marker annotation is simply an annotation that does not have any members. @Override, for example, is a marker annotation. It simply let's other users of your code know that you are overriding a superclass method. For example, when overriding Object.toString() in classes you create.

There a number of other subtle features and rules of the API such as that you may not annotate a method return type of type void. I will leave it to you do the discovery.

Knew about all this stuff already? Tell me about your experience in the comments below :)

Want the source? Grab it here.

afinlay5/JavaAnnotations

JavaAnnotations - Gradle source code repository for java source code examples posted on person...

github.com

```
/JavaAnnotations> ./gradlew run


ionContainer(value=[@src.main.java.SampleAnnotation(id=Public Class #
ss #8124)])


ionContainer(value=[@src.main.java.SampleAnnotation(id=Default Constr
ault Constructor #8124)])


ionContainer(value=[@src.main.java.SampleAnnotation(id=Field(String)
ing) #8124)])


ion2(id=Default ID #43216)

ionContainer(value=[@src.main.java.SampleAnnotation(id=Main Method #1
 #8124)])


ion.class):  "Default ID #43216"



ed, 1 up-to-date
/JavaAnnotations> 
```

Interested in Java? Join my Java group on Facebook:

Join My Java Facebook Group

Interested in Java? Check out my Facebook Group: Java Software Development Group!

medium.com

Like my Content? Subscribe to my mailing list:

Works Cited

[1]—Java: The Complete Reference, 9th Edition; Schildt, Herbert

[2]—Java® Platform, Standard Edition & Java Development Kit Version 9 API Specification