Chapter 13 - Annotations

Introducing Annotations

Annotations are all about metadata. That might not sound very exciting at first, but they add a lot of value to the Java language. Or perhaps, better said, they allow you to add a lot of value to your code.

Understanding Metadata

What exactly is metadata? *Metadata* is data that provides information about other data. Imagine our zoo is having a sale on tickets. The *attribute data* includes the price, the expiration date, and the number of tickets purchased. In other words, the attribute data is the transactional information that makes up the ticket sale and its contents.

On the other hand, the *metadata* includes the rules, properties, or relationships surrounding the ticket sales. Patrons must buy at least one ticket, as a sale of zero or negative tickets is silly. Maybe the zoo is having a problem with scalpers, so they add a rule that each person can buy a maximum of five tickets a day. These metadata rules describe information about the ticket sale but are not part of the ticket sale.

Purpose of Annotations

The purpose of an *annotation* is to assign metadata attributes to classes, methods, variables, and other Java types. Let's start with a simple annotation for our zoo: `@ZooAnimal`. Don't worry about how this annotation is defined or the syntax of how to call it just yet; we'll delve into that shortly. For now, you just need to know that annotations start with the at (`@`) symbol and can contain attribute/value pairs called *elements*.

```java
public class Mammal {}
public class Bird {}

@ZooAnimal public class Lion extends Mammal {}

@ZooAnimal public class Peacock extends Bird {}
```

That brings us to our first rule about annotations: *annotations function a lot like interfaces*. In this example, annotations allow us to *mark* a class as a `ZooAnimal` without changing its inheritance structure.

So if annotations function like interfaces, why don't we just use interfaces? While interfaces can be applied only to classes, annotations can be applied to any declaration including classes, methods, expressions, and even other annotations. Also, unlike interfaces, annotations allow us to pass a set of values where they are applied.

That brings us to our second rule about annotations: *annotations establish relationships that make it easier to manage data about our application*.

Sure, we could write applications without annotations, but that often requires creating a lot of extra classes, interfaces, or data files (XML, JSON, etc.) to manage these complex relationships. Worse yet, because these extra classes or files may be defined outside the class where they are being used, we have to do a lot of work to keep the data and the relationships in sync.

an annotation ascribes custom information on the declaration where it is defined. This turns out to be a powerful tool, as the same annotation can often be applied to completely unrelated classes or variables.

There's one final rule about annotations you should be familiar with: *annotations are optional metadata and by themselves do not do anything*. This means you can take a project filled with thousands of annotations and remove all of them, and it will still compile and run, albeit with potentially different behavior at runtime.

While an annotation can be removed from a class and it will still compile, the opposite is not true; adding an annotation can trigger a compiler error. As we will see in this chapter, the compiler validates that annotations are properly used and include all required fields.

Creating an Annotation

Let's say our zoo wants to specify the exercise metadata for various zoo inhabitants using annotations. We use the `@interface` annotation (all lowercase) to declare an annotation. Like classes and interfaces, they are commonly defined in their own file as a top-level type, although they can be defined inside a class declaration like an inner class.

```
public @interface Exercise {}
```

How do we use our new annotation? It's easy. We use the at (`@`) symbol, followed by the type name. In this case, the annotation is `@Exercise`. Then, we apply the annotation to other Java code, such as a class.

Let's apply `@Exercise` to some classes.

```
@Exercise() public class Cheetah {}

@Exercise public class Sloth {}

@Exercise
public class ZooEmployee {}
```

Specifying a Required Element

An *annotation element* is an attribute that stores values about the particular usage of an annotation. To make our previous example more useful, let's change `@Exercise` from a marker annotation to one that includes an element.

```
public @interface Exercise {
   int hoursPerDay();
}

```

The syntax for the `hoursPerDay()` element may seem a little strange at first. It looks a lot like an abstract method, although we're calling it an element (or attribute). Remember, annotations have their roots in interfaces. Behind the scenes, the JVM is creating elements as interface methods and annotations as implementations of these interfaces. Luckily, you don't need to worry about those details; the compiler does that for you.

When declaring an annotation, any element without a default value is considered required. We'll show you how to declare an optional element next.

Providing an Optional Element

For an element to be optional, rather than required, it must include a default value. Let's update our annotation to include an optional value.

```
public @interface Exercise {
   int hoursPerDay();
   int startHour() default 6;
}
```

```
@Exercise(startHour=5, hoursPerDay=3) public class Cheetah {}

@Exercise(hoursPerDay=0) public class Sloth {}

@Exercise(hoursPerDay=7, startHour="8")  // DOES NOT COMPILE
public class ZooEmployee {}

```

There are a few things to unpack here. First, when we have more than one element value within an annotation, we separate them by a comma (`,`). Next, each element is written using the syntax `elementName` `=` `elementValue`. It's like a shorthand for a `Map`. Also, the order of each element does not matter. `Cheetah` could have listed `hoursPerDay` first.

The default value of an annotation cannot be just any value. Similar to `case` statement values, *the default value of an annotation must be a non-* `null` constant expression.

```
    public @interface BadAnnotation {
       String name() default new String("");  // DOES NOT COMPILE
       String address() default "";
       String title() default null;           // DOES NOT COMPILE
    }

```

In this example, `name()` does not compile because it is not a constant expression, while `title()` does not compile because it is `null`. Only `address()` compiles. Notice that while `null` is not permitted as a default value, the empty `String ""` is.

Selecting an Element Type

Similar to a default element value, an annotation element cannot be declared with just any type. It must be a primitive type, a `String`, a `Class`, an enum, another annotation, or an array of any of these types.

Applying Element Modifiers

Like abstract interface methods, annotation elements are implicitly `abstract` and `public`, whether you declare them that way or not. Yep, just like interface variables, annotation variables are implicitly `public`, `static`, and `final`. These constant variables are not considered elements, though. For example, marker annotations can contain constants.

Using Annotations in Declarations

Up until now, we've only been applying annotations to classes and methods, but they can be applied to any Java declaration including the following:

- Classes, interfaces, enums, and modules
- Variables (`static`, instance, local)
- Methods and constructors
- Method, constructor, and lambda parameters
- Cast expressions
- Other annotations

```
1:  @FunctionalInterface interface Speedster {
2:     void go(String name);
3:  }
```

```
4:  @LongEars
5:  @Soft @Cuddly public class Rabbit {
6:     @Deprecated public Rabbit(@NotNull Integer size) {}
7:
8:     @Speed(velocity="fast") public void eat(@Edible String input) {
9:        @Food(vegetarian=true) String m = (@Tasty String) "carrots";
10:
11:       Speedster s1 = new @Racer Speedster() {
12:          public void go(@FirstName @NotEmpty String name) {
13:             System.out.print("Start! "+name);
14:          }
15:       };
16:
17:       Speedster s2 = (@Valid String n) -> System.out.print(n);
18:    }
19: }
```

When applying an annotation to an expression, a cast operation including the Java type is required. On line 9, the expression was cast to `String`, and the annotation `@Tasty` was applied to the type.

In this example, we applied annotations to various declarations, but this isn't always permitted. An annotation can specify which declaration type they can be applied to using the `@Target` annotation. We'll cover this, along with other annotation-specific annotations, in the next part of the chapter.

Creating a *value()* Element

In your development experience, you may have seen an annotation with a value, written without the `elementName`. For example, the following is valid syntax under the right condition:

```
@Injured("Broken Tail") public class Monkey {}

```

This is considered a shorthand or abbreviated annotation notation. What qualifies as *the right condition*? An annotation must adhere to the following rules to be used without a name:

- The annotation declaration must contain an element named `value()`, which may be optional or required.
- The annotation declaration must not contain any other elements that are required.
- The annotation usage must not provide values for any other elements.

Let's create an annotation that meets these requirements.

```
public @interface Injured {
   String veterinarian() default "unassigned";
   String value() default "foot";
   int age() default 1;
}
```

Annotations support a shorthand notation for providing an array that contains a single element. Let's say we have an annotation `Music` defined as follows:

```
public @interface Music {
   String[] genres();
}

```

If we want to provide only one value to the array, we have a choice of two ways to write the annotation. Either of the following is correct:

```
public class Giraffe {
   @Music(genres={"Rock and roll"}) String mostDisliked;
   @Music(genres="Classical") String favorite;
}
```

Many annotation declarations include `@Target` annotation, which limits the types the annotation can be applied to. More specifically, the `@Target` annotation takes an array of `ElementType` enum values as its `value()` element.

Understanding the *TYPE_USE* Value

`ElementType` value	Applies to
`TYPE`	Classes, interfaces, enums, annotations
`FIELD`	Instance and `static` variables, enum values
`METHOD`	Method declarations
`PARAMETER`	Constructor, method, and lambda parameters
`CONSTRUCTOR`	Constructor declarations
`LOCAL_VARIABLE`	Local variables
`ANNOTATION_TYPE`	Annotations
`PACKAGE` *	Packages declared in `package-info.java`
`TYPE_PARAMETER` *	Parameterized types, generic declarations
`TYPE_USE`	Able to be applied anywhere there is a Java type declared or used
`MODULE` *	Modules

While most of the values in Table 13.1 are straightforward, `TYPE_USE` is without a doubt the most complex. The `TYPE_USE` parameter *can be used anywhere there is a Java type*. By including it in `@Target`, it actually includes nearly all the values in Table 13.1 including classes, interfaces, constructors, parameters, and more. There are a few exceptions; for example, it can be used only on a method that returns a value. Methods that return `void` would still need the `METHOD` value defined in the annotation.

Storing Annotations with @*Retention*

annotations *may* be discarded by the compiler or at runtime. We say “may,” because we can actually specify how they are handled using the `@Retention` annotation. This annotation takes a `value()` of the enum `RetentionPolicy`. Table 13.2 shows the possible values, in increasing order of retention.

TABLE 13.2 Values for the @Retention annotation

`RetentionPolicy` value	Description
`SOURCE`	Used only in the source file, discarded by the compiler
`CLASS`	Stored in the `.class` file but not available at runtime (default compiler behavior)
`RUNTIME`	Stored in the `.class` file and available at runtime

For the exam, you should be familiar with the marker annotation `@Documented`. If present, then the generated Javadoc will include annotation information defined on Java types. Because it is a marker annotation, it doesn't take any values; therefore, using it is pretty easy.

```java
// Hunter.java
import java.lang.annotation.Documented;

@Documented public @interface Hunter {}

// Lion.java
@Hunter public class Lion {}
```

Inheriting Annotations with *@Inherited*

Another marker annotation you should know for the exam is `@Inherited`. When this annotation is applied to a class, subclasses will inherit the annotation information found in the parent class.

```java
// Vertebrate.java
import java.lang.annotation.Inherited;

@Inherited public @interface Vertebrate {}

// Mammal.java
@Vertebrate public class Mammal {}

// Dolphin.java
public class Dolphin extends Mammal {}
```

@Repeatable

The `@Repeatable` annotation is used when you want to specify an annotation more than once on a type.

without the `@Repeatable` annotation, an annotation can be applied only once.

declare a `@Repeatable` annotation, you must define a containing annotation type value.

A *containing annotation type* is a separate annotation that defines a `value()` array element. The type of this array is the particular annotation you want to repeat. By convention, the name of the annotation is often the plural form of the repeatable annotation.

Putting all of this together, the following `Risks` declaration is a containing annotation type for our `Risk` annotation:

```java
public @interface Risks {
   Risk[] value();
}
```

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

@Repeatable(Risks.class)
public @interface Risk {
   String danger();
   int level() default 1;
}
```

The following summarizes the rules for declaring a repeatable annotation, along with its associated containing type annotation:

- The repeatable annotation must be declared with `@Repeatable` and contain a value that refers to the containing type annotation.

- The containing type annotation must include an element named `value()`, which is a primitive array of the repeatable annotation type.

Annotation	Marker annotation	Type of `value()`	Default compiler behavior (if annotation not present)
`@Target`	No	Array of `ElementType`	Annotation able to be applied to all locations except `TYPE_USE` and `TYPE_PARAMETER`
`@Retention`	No	`RetentionPolicy`	`RetentionPolicy.CLASS`
`@Documented`	Yes	—	Annotations are not included in the generated Javadoc.
`@Inherited`	Yes	—	Annotations in supertypes are not inherited.
`@Repeatable`	No	Annotation	Annotation cannot be repeated.

Using Common Annotations

Override

The `@Override` is a marker annotation that is used to indicate a method is overriding an inherited method, whether it be inherited from an interface or parent class.

FunctionalInterface

In Chapter 12, we showed you how to create and identify functional interfaces, which are interfaces with exactly one abstract method. The `@FunctionalInterface` marker annotation can be applied to any valid functional interface

@Deprecated

The `@Deprecated` annotation can be applied to nearly any Java declaration, such as classes, methods, or variables.

While this may or may not appear on the exam, the `@Deprecated` annotation does support two optional values: `String since()` and `boolean forRemoval()`. They provide additional information about when the deprecation occurred in the past and whether or not the type is expected to be removed entirely in the future.

```
/**
 * Method to formulate a zoo layout.
 * @deprecated Use ParkPlanner.planPark(String… data) instead.
 */
@Deprecated(since="1.8", forRemoval=true)
public void plan() {}
```

@SuppressWarnings

Applying this annotation to a class, method, or type basically tells the compiler, “I know what I am doing; do not warn me about this.” Unlike the previous annotations, it requires a `String[] value()` parameter. Table 13.4 lists some of the values available for this annotation.

TABLE 13.4 Common *@SuppressWarnings* values

Value	Description
`"deprecation"`	Ignore warnings related to types or methods marked with the `@Deprecated` annotation.
`"unchecked"`	Ignore warnings related to the use of raw types, such as `List` instead of `List<String>`.

@SafeVarargs

The `@SafeVargs` marker annotation indicates that a method does not perform any potential unsafe operations on its varargs parameter. It can be applied only to constructors or methods that cannot be overridden (aka methods marked `private`, `static`, or `final`).

For the exam you don't need to know how to create or resolve unsafe operations, as that can be complex. You just need to be able to identify unsafe operations and know they often involve generics.

You should also know the annotation can be applied only to methods that contain a varargs parameter and are not able to be overridden. For example, the following do not compile:

```
@SafeVarargs
public static void eat(int meal) {}         // DOES NOT COMPILE

@SafeVarargs
protected void drink(String… cup) {}      // DOES NOT COMPILE

@SafeVarargs void chew(boolean… food) {}  // DOES NOT COMPILE

```

The `eat()` method is missing a varargs parameter, while the `drink()` and `chew()` methods are not marked `static`, `final`, or `private`.

Reviewing Common Annotations

Table 13.5 lists the common annotations that you will need to know for the exam along with how they are structured.

TABLE 13.5 Understanding common annotations

Annotation	Marker annotation	Type of `value()`	Optional members
`@Override`	Yes	—	—
`@FunctionalInterface`	Yes	—	—

`@Deprecated`	No	—	`String since()`
			`boolean forRemoval()`

`@SuppressWarnings`	No	`String[]`	—
`@SafeVarargs`	Yes	—	—