# Chapter 13 Annotations

---

**OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:**

- **Annotations**
  - Describe the purpose of annotations and typical usage patterns
  - Apply annotations to classes and methods
  - Describe commonly used annotations in the JDK
  - Declare custom annotations

---

There are some topics you need to know to pass the exam, some that are important in your daily development experience, and some that are important for both. Annotations definitely fall into this last category. Annotations were added to the Java language to make a developer's life a lot easier.

Prior to annotations, adding extra information about a class or method was often cumbersome and required a lot of extra classes and configuration files. Annotations solved this by having the data and the information about the data defined in the same location.

In this chapter, we define what an annotation is, how to create a custom annotation, and how to properly apply annotations. We will also teach you about built-in annotations that you will need to learn for the exam. We hope this chapter increases your understanding and usage of annotations in your professional development experience.

## 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.

As you'll learn in this chapter, annotations provide an easy and convenient way to insert metadata like this into your applications.

---


While annotations allow you to insert rules around data, it does not mean the values for these rules need to be defined in the code, aka "hard-coded." In many frameworks, you can define the rules and relationships in the code but read the values from elsewhere. In the previous example, you could define an annotation specifying a maximum number of tickets but load the value of `5` from a config file or database. For this chapter, though, you can assume the values are defined in the code.

---

## 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*.

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

@ZooAnimal public class Lion extends Mammal {}

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

In this case, the annotation is applied to the `Lion` and `Peacock` classes. We could have also had them extend a class called `ZooAnimal` , but then we have to change the class hierarchy. By using an annotation, we leave the class structure intact.

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.

Consider the following `Veterinarian` class:

```
public class Veterinarian {
   @ZooAnimal(habitat="Infirmary") private Lion sickLion;

   @ZooAnimal(habitat="Safari") private Lion healthyLion;
```

```
    @ZooAnimal(habitat="Special Enclosure") private Lion blindLion;
}
```

This class defines three variables, each with an associated `habitat` value. The `habitat` value is part of the type declaration of each variable, not an individual object. For example, the `healthyLion` may change the object it points to, but the value of the annotation does not. Without annotations, we'd have to define a new `Lion` type for each `habitat` value, which could become cumbersome given a large enough application.

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.

---

**Real World Scenario**

**EXTERNAL METADATA FILES**

Prior to annotations, many early Java enterprise frameworks relied on external XML files to store metadata about an application. Imagine managing an application with dozens of services, hundreds of objects, and thousands of attributes. The data would be stored in numerous Java files alongside a large, ever-growing XML file. And a change to one often required a change to the other.

As you can probably imagine, this becomes untenable very quickly! Many of these frameworks were abandoned or rewritten to use annotations. These days, XML files are still used with Java projects but often serve to provide minimal configuration information, rather than low-level metadata.

Consider the following methods that use a hypothetical `@ZooSchedule` annotation to indicate when a task should be performed.

```
// Lion.java
public class Lion {
   @ZooSchedule(hours={"9am","5pm","10pm"}) void feedLions() {
      System.out.print("Time to feed the lions!");
   }
}

// Peacock.java
public class Peacock {
   @ZooSchedule(hours={"4am","5pm"}) void cleanPeacocksPen() {
      System.out.print("Time to sweep up!");
   }
}
```

These methods are defined in completely different classes, but the interpretation of the annotation is the same. With this approach, the task and its schedule are defined right next to each other. This brings us to our third rule about annotations: *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.

This last rule might seem a little counterintuitive at first, but it refers to the fact that annotations aren't utilized where they are defined. It's up to the rest of the application, or more likely the underlying framework, to enforce or use annotations to accomplish tasks. For instance, marking a

method with `@SafeVarargs` informs other developers and development tools that no unsafe operations are present in the method body. It does not actually prevent unsafe operations from occurring!

---


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.

---

For the exam, you need to know how to define your own custom annotations, how to apply annotations properly, and how to use common annotations. Writing code that processes or enforces annotations is not required for the exam.

---


**Real World Scenario**

**THE SPRING FRAMEWORK**

While there are many platforms that rely on annotations, one of the most recognized and one of the first to popularize using annotations is the Spring Framework, or Spring for short. Spring uses annotations for many purposes, including dependency injection, which is a common technique of decoupling a service and the clients that use it.

In Chapter 17, “Modular Applications,” you'll learn all about Java's built-in service implementation, which uses `module-info` files rather than annotations to manage services. While modules and Spring are both providing dependencies dynamically, they are implemented in a very different manner.

Spring, along with the well-known convention over configuration Spring Boot framework, isn't on the exam, but we recommend professional Java developers be familiar with both of them.

## Creating Custom Annotations

Creating your own annotation is surprisingly easy. You just give it a name, define a list of optional and required elements, and specify its usage. In this section, we'll start with the simplest possible annotation and work our way up from there.

### 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 {}
```

Yes, we use an annotation to create an annotation! The `Exercise` annotation is referred to as a *marker annotation*, since it does not contain any elements. In Chapter 19, "I/O," you'll actually learn about something called a marker interface, which shares a lot of similarities with annotations.

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 {}
```

Oh no, we've mixed animals and zoo employees! That's perfectly fine. There's no rule that our `@Exercise` annotation has to be applied to animals. Like interfaces, annotations can be applied to unrelated classes.

You might have noticed that `Cheetah` and `Sloth` differ on their usage of the annotation. One uses parentheses, `()`, while the other does not. When using a marker annotation, parentheses are optional. Once we start adding elements, though, they are required if the annotation includes any values.

We also see `ZooEmployee` is not declared on the same line as its annotation. If an annotation is declared on a line by itself, then it applies to the next nonannotation type found on the proceeding lines. In fact, this applies when there are multiple annotations present.

```
@Scaley       @Flexible
   @Food("insect") @Food("rodent")      @FriendlyPet
@Limbless public class Snake {}
```

Some annotations are on the same line, some are on their own line, and some are on the line with the declaration of `Snake`. Regardless, all of the annotations apply to `Snake`. As with other declarations in Java, spaces and tabs between elements are ignored.

---


Whether you put annotations on the same line as the type they apply to or on separate lines is a matter of style. Either is acceptable.

---

In this example, some annotations are all lowercase, while others are mixed case. Annotation names are case sensitive. Like class and interface

names, it is common practice to have them start with an uppercase letter, although it is not required.

Finally, some annotations, like `@Food`, can be applied more than once. We'll cover repeatable annotations later in this chapter.

### 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.

Let's see how this new element changes our usage:

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

@Exercise hoursPerDay=0 public class Sloth {}     // DOES NOT COMPILE

@Exercise public class ZooEmployee {}             // DOES NOT COMPILE
```

The `Cheetah` class compiles and correctly uses the annotation, providing a value for the element. The `Sloth` class does not compile because it is missing parentheses around the annotation parameters. Remember, parentheses are optional only if no values are included.

What about `ZooEmployee`? This class does not compile because the `hoursPerDay` field is required. Remember earlier when we said annotations are optional metadata? While the annotation itself is optional, the compiler still cares that they are used correctly.

Wait a second, when did we mark `hoursPerDay()` as required? We didn't. But we also didn't specify a default value either. *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;
}
```

---


In Chapter 12, "Java Fundamentals," we mentioned that the `default` keyword can be used in `switch` statements and interface methods. Yep, they did it again. There is yet another use of the `default` keyword that is unrelated to any of the usages you were previously familiar with. And don't forget package-private access is commonly referred to as default access without any modifiers. Good grief!

---

Next, let's apply the updated annotation to our classes.

```
@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.

We also see that `Sloth` does not specify a value for `startHour` , meaning it will be instantiated with the default value of `6` .

In this version, the `ZooEmployee` class does not compile because it defines a value that is incompatible with the `int` type of `startHour` . The compiler is doing its duty validating the type!

---

#### DEFINING A DEFAULT ELEMENT VALUE

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.

Given this information and our previous `Exercise` annotation, which of the following elements compile?

```
public class Bear {}

public enum Size {SMALL, MEDIUM, LARGE}

public @interface Panda {
   Integer height();
   String[][] generalInfo();
   Size size() default Size.SMALL;
   Bear friendlyBear();
   Exercise exercise() default @Exercise(hoursPerDay=2);
}
```

The `height()` element does not compile. While primitive types like `int` and `long` are supported, wrapper classes like `Integer` and `Long` are not. The `generalInfo()` element also does not compile. The type `String[]` is supported, as it is an array of `String` values, but `String[][]` is not.

The `size()` and `exercise()` elements both compile, with one being an enum and the other being an annotation. To set a default value for `exercise()`, we use the `@Exercise` annotation. Remember, this is the only way to create an annotation value. Unlike instantiating a class, the `new` keyword is never used to create an annotation.

Finally, the `friendlyBear()` element does not compile. The type of `friendlyBear()` is `Bear` (not `Class`). Even if `Bear` were changed to an interface, the `friendlyBear()` element would still not compile since it is not one of the supported types.

### Applying Element Modifiers

Like abstract interface methods, annotation elements are implicitly `abstract` and `public`, whether you declare them that way or not.

```
public @interface Material {}

public @interface Fluffy {
   int cuteness();
   public abstract int softness() default 11;
   protected Material material();  // DOES NOT COMPILE
   private String friendly();      // DOES NOT COMPILE
   final boolean isBunny();        // DOES NOT COMPILE
}
```

The elements `cuteness()` and `softness()` are both considered `abstract` and `public`, even though only one of them is marked as such. The elements `material()` and `friendly()` do not compile because the access modifier conflicts with the elements being implicitly `public`. The element `isBunny()` does not compile because, like abstract methods, it cannot be marked `final`.

### Adding a Constant Variable

Annotations can include constant variables that can be accessed by other classes without actually creating the annotation.

```
public @interface ElectricitySource {
   public int voltage();
   int MIN_VOLTAGE = 2;
   public static final int MAX_VOLTAGE = 18;
}
```

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.

### Reviewing Annotation Rules

We conclude creating custom annotations with Figure 13.1, which summarizes many of the syntax rules that you have learned thus far.

Figure 13.2 shows how to apply this annotation to a Java class. For contrast, it also includes a simple marker annotation `@Alert`. Remember, a marker annotation is one that does not contain any elements.

If you understand all of the parts of these two figures, then you are well on your way to understanding annotations. If not, then we suggest rereading this section. The rest of the chapter will build on this foundation; you will see more advanced usage, apply annotations to other annotations, and learn the built-in annotations that you will need to know for the exam.

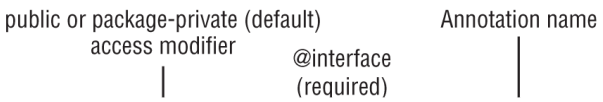


**FIGURE 13.1** Annotation declaration

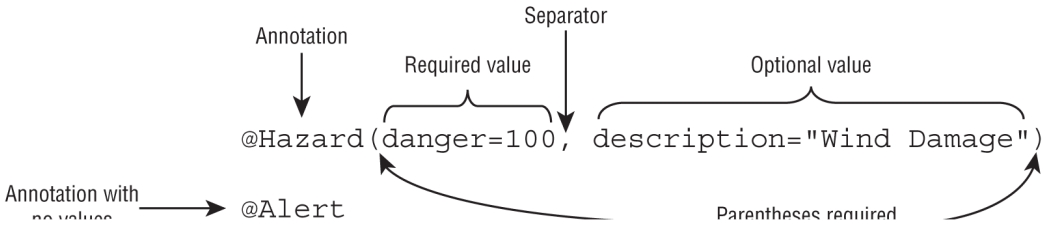


**FIGURE 13.2** Using an annotation

## Applying Annotations

Now that we have described how to create and use simple annotations, it's time to discuss other ways to apply annotations.

### 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

The following compiles, assuming the annotations referenced in it exist:

```
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: }
```

It's a little contrived, we know. Lines 1, 4, and 5 apply annotations to the interface and class declarations. Some of the annotations, like `@Cuddly`, do not require any values, while others, like `@Speed`, do provide values. You would need to look at the annotation declaration to know if these values are optional or required.

Lines 6 and 8 contain annotations applied to constructor and method declarations. These lines also contain annotations applied to their parameters.

Line 9 contains the annotation `@Food` applied to a local variable, along with the annotation `@Tasty` applied to a cast expression.

---


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.

---

Line 11 applies an annotation to the type in the anonymous class declaration, and line 17 shows an annotation in a lambda expression parameter. Both of these examples may look a little odd at first, but they are allowed. In fact, you're more likely to see examples like this on the exam than you are in real life.

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.

### Mixing Required and Optional Elements

One of the most important rules when applying annotations is the following: *to use an annotation, all required values must be provided*. While an annotation may have many elements, values are required only for ones without default values.

Let's try this. Given the following annotation:

```
public @interface Swimmer {
   int armLength = 10;
   String stroke();
   String name();
   String favoriteStroke() default "Backstroke";
}
```

which of the following compile?

```
@Swimmer class Amphibian {}

@Swimmer(favoriteStroke="Breaststroke", name="Sally") class Tadpole {}

@Swimmer(stroke="FrogKick", name="Kermit") class Frog {}

@Swimmer(stroke="Butterfly", name="Kip", armLength=1) class Reptile {}

@Swimmer(stroke="", name="", favoriteStroke="") class Snake {}
```

`Amphibian` does not compile, because it is missing the required elements `stroke()` and `name()`. Likewise, `Tadpole` does not compile, because it is missing the required element `stroke()`.

`Frog` provides all of the required elements and none of the optional ones, so it compiles. `Reptile` does not compile since `armLength` is a constant, not an element, and cannot be included in an annotation. Finally, `Snake` does compile, providing all required and optional values.

### 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;
}
```

This annotation is composed of multiple optional elements. In this example, we gave `value()` a default value, but we could have also made it required. Using this declaration, the following annotations are valid:

```
public abstract class Elephant {
   @Injured("Legs") public void fallDown() {}
   @Injured(value="Legs") public abstract int trip();
   @Injured String injuries[];
}
```

The usage in the first two annotations are equivalent, as the compiler will convert the shorthand form to the long form with the `value()` element

name. The last annotation with no values is permitted because `@Injured` does not have any required elements.

---


Typically, the `value()` of an annotation should be related to the name of the annotation. In our previous example, `@Injured` was the annotation name, and the `value()` referred to the item that was impacted. This is especially important since all shorthand elements use the same element name, `value()`.

---

For the exam, make sure that if the shorthand notation is used, then there is an element named `value()`. Also, check that there are no other required elements. For example, the following annotation declarations cannot be used with a shorthand annotation:

```
public @interface Sleep {
   int value();
   String hours();
}

public @interface Wake {
   String hours();
}
```

The first declaration contains two required elements, while the second annotation does not include an element named `value()`.

Likewise, the following annotation is not valid as it provides more than one value:

```
@Injured("Fur",age=2) public class Bear {}  // DOES NOT COMPILE
```

### Passing an Array of Values

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;
}
```

The first annotation is considered the regular form, as it is clear the usage is for an array. The second annotation is the shorthand notation, where the array braces ( `{}` ) are dropped for convenience. Keep in mind that this is still providing a value for an array element; the compiler is just inserting the missing array braces for you.

This notation can be used only if the array is composed of a single element. For example, only one of the following annotations compiles:

```
public class Reindeer {
   @Music(genres="Blues","Jazz") String favorite;  // DOES NOT COMPILE
   @Music(genres=) String mostDisliked;            // DOES NOT COMPILE
   @Music(genres=null) String other;               // DOES NOT COMPILE
   @Music(genres={}) String alternate;
}
```

The first provides more than one value, while the next two do not provide any values. The last one does compile, as an array with no elements is still

a valid array.

While this shorthand notation can be used for arrays, it does not work for `List` or `Collection`. As mentioned earlier, they are not in the list of supported element types for annotations.

---

COMBINING SHORTHAND NOTATIONS

It might not surprise you that we can combine both of our recent rules for shorthand notations. Consider this annotation:

```
public @interface Rhythm {
   String[] value();
}
```

Each of the following four annotations is valid:

```
public class Capybara {
   @Rhythm(value={"Swing"}) String favorite;
   @Rhythm(value="R&B") String secondFavorite;
   @Rhythm({"Classical"}) String mostDisliked;
   @Rhythm("Country") String lastDisliked;
}
```

The first annotation provides all of the details, while the last one applies both shorthand rules.

---

## Declaring Annotation-Specific Annotations

Congratulations—if you've gotten this far, then you've learned all of the general rules for annotations we have to teach you! From this point on, you'll need to learn about specific annotations and their associated rules for the exam. Many of these rules are straightforward, although some will require memorization.

In this section, we'll cover built-in annotations applied to other annotations. Yes, metadata about metadata! Since these annotations are built into Java, they primarily impact the compiler. In the final section, we'll cover built-in annotations applied to various Java data types.

## Limiting Usage with *@Target*

Earlier, we showed you examples of annotations applied to various Java types, such as classes, methods, and expressions. When defining your own annotation, you might want to limit it to a particular type or set of types. After all, it may not make sense for a particular annotation to be applied to a method parameter or local variable.

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.

### Learning the *ElementType* Values

Table 13.1 shows all of the values available for the `@Target` annotation.

**TABLE 13.1** Values for the `@Target` annotation

| `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 |

* Applying these with annotations is out of scope for the exam.

You might notice that some of the `ElementType` applications overlap. For example, to create an annotation usable on other annotations, you could declare an `@Target` with `ANNOTATION_TYPE` or `TYPE`. Either will work

for annotations, although the second option opens the annotation usage to other types like classes and interfaces.

While you are not likely to be tested on all of these types, you may see a few on the exam. Make sure you can recognize proper usage of them. Most are pretty self-explanatory.

---


You can't add a package annotation to just any package declaration, only those defined in a special file, which must be named `package-info.java`. This file stores documentation metadata about a package. Don't worry, though, this isn't on the exam.

---

Consider the following annotation:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.METHOD,ElementType.CONSTRUCTOR})
public @interface ZooAttraction {}
```

---


Even though the `java.lang` package is imported automatically by the compiler, the `java.lang.annotation` package is not. Therefore, `import` statements are required for many of the examples in the remainder of this chapter.

---

Based on this annotation, which of the following lines of code will compile?

```
1: @ZooAttraction class RollerCoaster {}
2: class Events {
3:    @ZooAttraction String rideTrain() {
4:       return (@ZooAttraction String) "Fun!";
5:    }
6:    @ZooAttraction Events(@ZooAttraction String description) {
7:       super();
8:    }
9:    @ZooAttraction int numPassengers; }
```

This example contains six uses of `@ZooAttraction` with only two of them being valid. Line 1 does not compile, because the annotation is applied to a class type. Line 3 compiles, because it is permitted on a method declaration. Line 4 does not compile, because it is not permitted on a cast operation.

Line 6 is tricky. The first annotation is permitted, because it is applied to the constructor declaration. The second annotation is not, as the annotation is not marked for use in a constructor parameter. Finally, line 9 is not permitted, because it cannot be applied to fields or variables.

**Understanding the *TYPE_USE* Value**

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.

It also allows annotations in places where types are *used*, such as cast operations, object creation with `new`, inside type declarations, etc. These might seem a little strange at first, but the following are valid `TYPE_USE` applications:

```
// Technical.java
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target(ElementType.TYPE_USE)
@interface Technical {}

// NetworkRepair.java
import java.util.function.Predicate;
public class NetworkRepair {
   class OutSrc extends @Technical NetworkRepair {}
   public void repair() {
      var repairSubclass = new @Technical NetworkRepair() {};

      var o = new @Technical NetworkRepair().new @Technical OutSrc();

      int remaining = (@Technical int)10.0;
   }
}
```

For the exam, you don't need to know all of the places `TYPE_USE` can be used, nor what applying it to these locations actually does, but you do need to recognize that they can be applied in this manner if `TYPE_USE` is one of the `@Target` options.

### Storing Annotations with *@Retention*

As you saw in Chapter 7, "Methods and Encapsulation," the compiler discards certain types of information when converting your source code into a `.class` file. With generics, this is known as *type erasure*.

In a similar vein, 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 |

Using it is pretty easy.

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

@Retention(RetentionPolicy.CLASS) @interface Flier {}
@Retention(RetentionPolicy.RUNTIME) @interface Swimmer {}
```

In this example, both annotations will retain the annotation information in their `.class` files, although only `Swimmer` will be available (via reflection) at runtime.

### Generating Javadoc with *@Documented*

When trying to determine what methods or classes are available in Java or a third-party library, you've undoubtedly relied on web pages built with Javadoc. Javadoc is a built-in standard within Java that generates documentation for a class or API.

In fact, you can generate Javadoc files for any class you write! Better yet, you can add additional metadata, including comments and annotations,

that have no impact on your code but provide more detailed and user-friendly Javadoc files.

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.

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

@Documented public @interface Hunter {}

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

In this example, the `@Hunter` annotation would be published with the `Lion` Javadoc information because it's marked with the `@Documented` annotation.

JAVA VS. JAVADOC ANNOTATIONS

Javadoc has its own annotations that are used solely in generating data within a Javadoc file.

```
public class ZooLightShow {

   /**
    * Performs a light show at the zoo.
    *
    * @param      distance   length the light needs to travel.
    * @return     the result of the light show operation.
    * @author     Grace Hopper
    * @since      1.5
    * @deprecated Use EnhancedZooLightShow.lights() instead.
    */
   @Deprecated(since="1.5") public static String perform(int distance)
      return "Beginning light show!";
   }
}
```

Be careful not to confuse Javadoc annotations with the Java annotations. Take a look at the `@deprecated` and `@Deprecated` annotations in this example. The first, `@deprecated`, is a Javadoc annotation used inside a comment, while `@Deprecated` is a Java annotation applied to a class. Traditionally, Javadoc annotations are all lowercase, while Java annotations start with an uppercase letter.

## 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.

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

```
@Inherited public @interface Vertebrate {}

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

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

In this example, the `@Vertebrate` annotation will be applied to both `Mammal` and `Dolphin` objects. Without the `@Inherited` annotation, `@Vertebrate` would apply only to `Mammal` instances.

### Supporting Duplicates with *@Repeatable*

The last annotation-specific annotation you need to know for the exam is arguably the most complicated to use, as it actually requires creating two annotations. The `@Repeatable` annotation is used when you want to specify an annotation more than once on a type.

Why would you want to specify twice? Well, if it's a marker annotation with no elements, you probably wouldn't. Generally, you use repeatable annotations when you want to apply the same annotation with different values.

Let's assume we have a repeatable annotation `@Risk`, which assigns a set of risk values to a zoo animal. We'll show how it is used and then work backward to create it.

```
public class Zoo {
   public static class Monkey {}

   @Risk(danger="Silly")
   @Risk(danger="Aggressive",level=5)
   @Risk(danger="Violent",level=10)
   private Monkey monkey;
}
```

Next, let's define a simple annotation that implements these elements:

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

Now, as written, the `Zoo` class does not compile. Why? Well, the `Risk` annotation is missing the `@Repeatable` annotation! That brings us to our first rule: *without the `@Repeatable` annotation, an annotation can be applied only once*. So, let's add the `@Repeatable` annotation.

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

@Repeatable  // DOES NOT COMPILE
public @interface Risk {
   String danger();
   int level() default 1;
}
```

This code also does not compile, but this time because the `@Repeatable` annotation is not declared correctly. It requires a reference to a second annotation. That brings us to our next rule: *to 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:

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

Finally, we go back to our original `Risk` annotation and specify the containing annotation class:

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

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

With these two annotations, our original `Zoo` class will now compile. Notice that we never actually use `@Risks` in our `Zoo` class. Given the declaration of the `Risk` and `Risks` annotations, the compiler takes care of applying the annotations for us.

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.

Once you understand the basic structure of declaring a repeatable annotation, it's all pretty convenient.

#### REPEATABLE ANNOTATIONS VS. AN ARRAY OF ANNOTATIONS

Repeatable annotations were added in Java 8. Prior to this, you would have had to use the `@Risks` containing annotation type directly:

```
@Risks({
   @Risk(danger="Silly"),
   @Risk(danger="Aggressive",level=5),
   @Risk(danger="Violent",level=10)
})
private Monkey monkey;
```

With this implementation, `@Repeatable` is not required in the `Risk` annotation declaration. The `@Repeatable` annotation is the preferred approach now, as it is easier than working with multiple nested statements.

---

## Reviewing Annotation-Specific Annotations

We conclude this part of the chapter with Table 13.3, which shows the annotations that can be applied to other annotations that might appear on the exam.

**TABLE 13.3** Annotation-specific annotations

| 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. |

Prior to this section, we created numerous annotations, and we never used any of the annotations in Table 13.3. So, what did the compiler do? Like implicit modifiers and default no-arg constructors, the compiler auto-inserted information based on the lack of data.

The default behavior for most of the annotations in Table 13.3 is often intuitive. For example, without the `@Documented` or `@Inherited` annotation, these features are not supported. Likewise, the compiler will report an error if you try to use an annotation more than once without the `@Repeatable` annotation.

The `@Target` annotation is a bit of a special case. When `@Target` is not present, an annotation can be used in any place except `TYPE_USE` or `TYPE_PARAMETER` scenarios (cast operations, object creation, generic declarations, etc.).

---

**WHY DOESN'T *@TARGET*'S DEFAULT BEHAVIOR APPLY TO ALL TYPES?**

We learn from Table 13.3 that to use an annotation in a type use or type parameter location, such as a lambda expression or generic declaration, you must explicitly set the `@Target` to include these values. If an annotation is declared without the `@Target` annotation that includes these values, then these locations are prohibited.

One possible explanation for this behavior is backward compatibility. When these values were added to Java 8, it was decided that they would have to be explicitly declared to be used in these locations.

That said, when the authors of Java added the `MODULE` value in Java 9, they did not make this same decision. If `@Target` is absent, the annotation is permitted in a module declaration by default.

---

## Using Common Annotations

For the exam, you'll need to know about a set of built-in annotations, which apply to various types and methods. Unlike custom annotations that you might author, many of these annotations have special rules. In fact, if they are used incorrectly, the compiler will report an error.

Some of these annotations (like `@Override`) are quite useful, and we recommend using them in practice. Others (like `@SafeVarargs`), you are likely to see only on a certification exam. For each annotation, you'll need to understand its purposes, identify when to use it (or not to use it), and know what elements it takes (if any).

### Marking Methods with *@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. From Chapter 8, "Class Design," you should know that the overriding method must have the same signature, the same or broader access modifier, and a covariant return type, and not declare any new or broader checked exceptions.

Let's take a look at an example:

```
public interface Intelligence {
   int cunning();
}
public class Canine implements Intelligence {
   @Override public int cunning() { return 500; }
   void howl() { System.out.print("Woof!"); }
}
public class Wolf extends Canine {
   @Override
   public int cunning() { return Integer.MAX_VALUE; }
   @Override void howl() { System.out.print("Howl!"); }
}
```

In this example, the `@Override` annotation is applied to three methods that it inherits from the parent class or interface.

During the exam, you should be able to identify anywhere this annotation is used incorrectly. For example, using the same `Canine` class, this `Dog` class does not compile:

```
public class Dog extends Canine {
   @Override
   public boolean playFetch() { return true; }  // DOES NOT COMPILE
   @Override void howl(int timeOfDay) {}        // DOES NOT COMPILE
}
```

The `playFetch()` method does not compile, because there is no inherited method with that name. In the `Dog` class, `howl()` is an overloaded

method, not an overridden one. While there is a `howl()` method defined in the parent class, it does not have the same signature. It is a method overload, not a method override.

Removing both uses of the `@Override` annotation in the `Dog` class would allow the class to compile. Using these annotations is not required, but using them incorrectly is prohibited.

---


The annotations in this section are entirely optional but help improve the quality of the code. By adding these annotations, though, you help take the guesswork away from someone reading your code. It also enlists the compiler to help you spot errors. For example, applying `@Override` on a method that is not overriding another triggers a compilation error and could help you spot problems if a class or interface is later changed.

---

### Declaring Interfaces with *@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. For example, our previous `Intelligence` example was actually a functional interface.

```
@FunctionalInterface public interface Intelligence {
   int cunning();
}
```

The compiler will report an error, though, if applied to anything other than a valid functional interface. From what you learned in Chapter 12, which of the following declarations compile?

```
@FunctionalInterface abstract class Reptile {
   abstract String getName();
}

@FunctionalInterface interface Slimy {}

@FunctionalInterface interface Scaley {
   boolean isSnake();
}

@FunctionalInterface interface Rough extends Scaley {
   void checkType();
}

@FunctionalInterface interface Smooth extends Scaley {
   boolean equals(Object unused);
}
```

The `Reptile` declaration does not compile, because the `@FunctionalInterface` annotation can be applied only to interfaces. The `Slimy` interface does not compile, because it does not contain any abstract methods. The `Scaley` interface compiles, as it contains exactly one abstract method.

The `Rough` interface does not compile, because it contains two abstract methods, one of which it inherits from `Scaley`. Finally, the `Smooth` interface contains two abstract methods, although since one matches the signature of a method in `java.lang.Object`, it does compile.

Like we saw with the `@Override` annotation, removing the `@FunctionalInterface` annotation in the invalid declarations would allow the code to compile. Review functional interfaces in Chapter 12 if you had any trouble with these examples.


If you are declaring a complex interface, perhaps one that contains `static`, `private`, and `default` methods, there's a simple test you can perform to determine whether it is a valid functional interface. Just add the `@FunctionalInterface` annotation to it! If it compiles, it is a functional interface and can be used with lambda expressions.

---

### Retiring Code with *@Deprecated*

In professional software development, you rarely write a library once and never go back to it. More likely, libraries are developed and maintained over a period of years. Libraries change for external reasons, like new business requirements or new versions of Java, or internal reasons, like a bug is found and corrected.

Sometimes a method changes so much that we need to create a new version of it entirely with a completely different signature. We don't want to necessarily remove the old version of the method, though, as this could cause a lot of compilation headaches for our users if the method suddenly disappears. What we want is a way to notify our users that a new version of the method is available and give them time to migrate their code to the new version before we finally remove the old version.

With those ideas in mind, Java includes the `@Deprecated` annotation. The `@Deprecated` annotation is similar to a marker annotation, in that it can be used without any values, but it includes some optional elements. The `@Deprecated` annotation can be applied to nearly any Java declaration, such as classes, methods, or variables.

Let's say we have an older class `ZooPlanner`, and we've written a replacement `ParkPlanner`. We want to notify all users of the older class to switch to the new version.

```
/**
 * Design and plan a zoo.
 * @deprecated Use ParkPlanner instead.
 */
@Deprecated
public class ZooPlanner { … }
```

That's it! The users of the `ZooPlanner` class will now receive a compiler warning if they are using `ZooPlanner`. In the next section, we'll show how they can use another annotation to ignore these warnings.

---

#### ALWAYS DOCUMENT THE REASON FOR DEPRECATION

Earlier, we discussed `@Deprecated` and `@deprecated`, the former being a Java annotation and the latter being a Javadoc annotation. Whenever you deprecate a method, you should add a Javadoc annotation to instruct users on how they should update their code.

For the exam, you should know that it is good practice to document why a type is being deprecated and be able to suggest possible alternatives.

---

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() {}
```

Note that the `@Deprecated` annotation does not allow you to provide any suggested alternatives. For that, you should use the Javadoc annotation.

---


When reviewing the Java JDK, you may encounter classes or methods that are marked deprecated, with the purpose that developers migrate to a new implementation. For example, the constructors of the wrapper classes (like `Integer` or `Double`) were recently marked `@Deprecated`, with the Javadoc note that you should use the factory method `valueOf()` instead. In this case, the advantage is that an immutable value from a pool can be reused, rather than creating a new object. This saves memory and improves performance.

---

### Ignoring Warnings with *@SuppressWarnings*

One size does not fit all. While the compiler can be helpful in warning you of potential coding problems, sometimes you need to perform a particular operation, and you don't care whether or not it is a potential programming problem.

Enter `@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>`. |

The annotation actually supports a lot of other values, but for the exam, you only need to know the ones listed in this table. Let's try an example:

```
import java.util.*;

class SongBird {
   @Deprecated static void sing(int volume) {}
   static Object chirp(List<String> data) { return data.size(); }
}

public class Nightingale {
   public void wakeUp() {
      SongBird.sing(10);
   }
   public void goToBed() {
      SongBird.chirp(new ArrayList());
   }
   public static void main(String[] args) {
      var birdy = new Nightingale();
      birdy.wakeUp();
      birdy.goToBed();
   }
}
```

This code compiles and runs but produces two compiler warnings.

```
Nightingale.java uses or overrides a deprecated API.
Nightingale.java uses unchecked or unsafe operations.
```

The first warning is because we are using a method `SongBird.sing()` that is deprecated. The second warning is triggered by the call to `new ArrayList()`, which does not define a generic type. An improved implementation would be to use `new ArrayList<String>()`.

Let's say we are absolutely sure that we don't want to change our `Nightingale` implementation, and we don't want the compiler to bother us anymore about these warnings. Adding the `@SuppressWarnings` annotation, with the correct values, accomplishes this.

```
@SuppressWarnings("deprecation") public void wakeUp() {
   SongBird.sing(10);
}

@SuppressWarnings("unchecked") public void goToBed() {
   SongBird.chirp(new ArrayList());
}
```

Now our code compiles, and no warnings are generated.

---


You should use the `@SuppressWarnings` annotation sparingly. Oftentimes, the compiler is correct in alerting you to potential coding problems. In some cases, a developer may use this annotation as a way to ignore a problem, rather than refactoring code to solve it.

---

### Protecting Arguments with *@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`).

Let's review varargs for a minute. A varargs parameter is used to indicate the method may be passed zero or more values of the same type, by providing an ellipsis ( ... ). In addition, a method can have at most one varargs parameter, and it must be listed last.

Returning to `@SafeVargs`, the annotation is used to indicate to other developers that your method does not perform any unsafe operations. It basically tells other developers, “Don't worry about the varargs parameter; I promise this method won't do anything bad with it!” It also suppresses unchecked compiler warnings for the varargs parameter.

In the following example, `thisIsUnsafe()` performs an unsafe operation using its varargs parameter:

```
1:  import java.util.*;
2:
3:  public class NeverDoThis {
4:     final int thisIsUnsafe(List<Integer>… carrot) {
5:        Object[] stick = carrot;
6:        stick[0] = Arrays.asList("nope!");
7:        return carrot[0].get(0);  // ClassCastException at runtime
8:     }
9:     public static void main(String[] a) {
10:       var carrot = new ArrayList<Integer>();
11:       new NeverDoThis().thisIsUnsafe(carrot);
12:    }
13: }
```

This code compiles, although it generates two compiler warnings. Both are related to type safety.

```
[Line 4]  Type safety: Potential heap pollution via varargs
   parameter carrot
[Line 11] Type safety: A generic array of List<Integer> is created
   for a varargs parameter
```

We can remove both compiler warnings by adding the `@SafeVarargs` annotation to line 4.

```
3:     @SafeVarargs final int thisIsUnsafe(List<Integer>… carrot) {
```

Did we actually fix the unsafe operation? No! It still throws a `ClassCastException` at runtime on line 7. However, we made it so the compiler won't warn us about it anymore.

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 | — | — |

Some of these annotations have special rules that will trigger a compiler error if used incorrectly, as shown in Table 13.6.

**TABLE 13.6** Applying common annotations

| Annotation | Applies to | Compiler error when |
|---|---|---|
| `@Override` | Methods | Method signature does not match the signature of an inherited method |
| `@FunctionalInterface` | Interfaces | Interface does not contain a single abstract method |
| `@Deprecated` | Most Java declarations | — |
| `@SuppressWarnings` | Most Java declarations | — |
| `@SafeVarargs` | Methods, constructors | Method or constructor does not contain a varargs parameter or is applied to a method not marked `private`, `static`, or `final` |

While none of these annotations is required, they do improve the quality of your code. They also help prevent you from making a mistake.

Let's say you override a method but accidentally alter the signature so that the compiler considers it an overload. If you use the `@Override` annotation, then the compiler will immediately report the error, rather than finding it later during testing.

JAVABEAN VALIDATION

This chapter covered only the annotations you need to know for the exam, but there are many incredibly useful annotations available.

If you've ever used JavaBeans to transmit data, then you've probably written code to validate it. While this can be cumbersome for large data structures, annotations allow you to mark `private` fields directly. The following are some useful `javax.validation` annotations:

- `@NotNull` : Object cannot be `null`
- `@NotEmpty` : Object cannot be `null` or have size of `0`
- `@Size(min=5,max=10)` : Sets minimum and/or maximum sizes
- `@Max(600)` and `@Min(-5)` : Sets the maximum or minimum numeric values
- `@Email` : Validates that the email is in a valid format

These annotations can be applied to a variety of data types. For example, when `@Size` is applied to a `String` , it checks the number of characters in the `String` . When applied to an array or `Collection` , it checks the number of elements present.

Of course, using the annotations is only half the story. The service receiving or processing the data needs to perform the validation step. In some frameworks like Spring Boot, this can be performed automatically by adding the `@Valid` annotation to a service parameter.

## Summary

In this chapter, we taught you everything you need to know about annotations for the exam. Ideally, we also taught you how to create and use custom annotations in your daily programming life. As we mentioned early on, annotations are one of the most convenient tools available in the Java language.

For the exam, you need to know the structure of an annotation declaration, including how to declare required elements, optional elements, and constant variables. You also need to know how to apply an annotation properly and ensure required elements have values. You should also be familiar with the two shorthand notations we discussed in this chapter. The first allows you to drop the `elementName` under certain conditions. The second allows you to specify a single value for an array element without the array braces ( `{}` ).

You need to know about the various built-in annotations available in the Java language. We sorted these into two groups: annotations that apply to other annotations and common annotations. The annotation-specific annotations provide rules for how annotations are handled by the compiler, such as specifying an inheritance or retention policy. They can also be used to disallow certain usage, such as using a method-targeted annotation applied to a class declaration.

The second set of annotations are common ones that you should know for the exam. Many, like `@Override` and `@FunctionalInterface` , are quite useful and provide other developers with additional information about your application.

## Exam Essentials

- **Be able to declare annotations with required elements, optional elements, and variables.** An annotation is declared with the `@interface` type. It may include elements and `public static final` constant variables. If it does not include any elements, then it is a marker annotation. Optional elements are specified with a `default` keyword and value, while required elements are those specified without one.
- **Be able to identify where annotations can be applied.** An annotation is applied using the at ( `@` ) symbol, followed by the annotation name. Annotations must include a value for each required element and can be applied to types, methods, constructors, and variables.

They can also be used in cast operations, lambda expressions, or inside type declarations.

- **Understand how to apply an annotation without an element name.** If an annotation contains an element named `value()` and does not contain any other elements that are required, then it can be used without the *`elementName`*. For it to be used properly, no other values may be passed.
- **Understand how to apply an annotation with a single-element array.** If one of the annotation elements is a primitive array and the array is passed a single value, then the annotation value may be written without the array braces (`{}`).
- **Apply built-in annotations to other annotations.** Java includes a number of annotations that apply to annotation declarations. The `@Target` annotation allows you to specify where an annotation can and cannot be used. The `@Retention` annotation allows you to specify at what level the annotation metadata is kept or discarded. `@Documented` is a marker annotation that allows you to specify whether annotation information is included in the generated documentation. `@Inherited` is another marker annotation that determines whether annotations are inherited from super types. The `@Repeatable` annotation allows you to list an annotation more than once on a single declaration. It requires a second containing type annotation to be declared.
- **Apply common annotations to various Java types.** Java includes many built-in annotations that apply to classes, methods, variables, and expressions. The `@Override` annotation is used to indicate that a method is overriding an inherited method. The `@FunctionalInterface` annotation confirms that an interface contains exactly one abstract method. Marking a type `@Deprecated` means that the compiler will generate a depreciation warning when it is referenced. Adding `@SuppressWarnings` with a set of values to a declaration causes the compiler to ignore the set of specified warnings. Adding `@SafeVarargs` on a constructor or `private`, `static`, or `final` method instructs other developers that no unsafe operations will be performed on its varargs parameter. While all of these annota-

tions are optional, they are quite useful and improve the quality of code when used.

## Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. What modifier is used to mark that an annotation element is required?
   A. `optional`
   B. `default`
   C. `required`
   D. `*`
   E. None of the above
2. Which of the following lines of code do not compile? (Choose all that apply.)

```
1: import java.lang.annotation.Documented;
2: enum Color {GREY, BROWN}
3: @Documented public @interface Dirt {
4:    boolean wet();
5:    String type() = "unknown";
6:    public Color color();
7:    private static final int slippery = 5;
8: }
```

   A. Line 2
   B. Line 3
   C. Line 4
   D. Line 5
   E. Line 6
   F. Line 7
   G. All of the lines compile.
3. Which built-in annotations can be applied to an annotation declaration? (Choose all that apply.)

A. `@Override`
B. `@Deprecated`
C. `@Document`
D. `@Target`
E. `@Repeatable`
F. `@Functional`

4. Given an automobile sales system, which of the following information is best stored using an annotation?
   A. The price of the vehicle
   B. A list of people who purchased the vehicle
   C. The sales tax of the vehicle
   D. The number of passengers a vehicle is rated for
   E. The quantity of models in stock
5. Which of the following lines of code do not compile? (Choose all that apply.)

```
1: import java.lang.annotation.*;
2: class Food {}
3: @Inherited public @interface Unexpected {
4:    public String rsvp() default null;
5:    Food food();
6:    public String[] dessert();
7:    final int numberOfGuests = 5;
8:    long startTime() default 0L;
9: }
```

   A. Line 3
   B. Line 4
   C. Line 5
   D. Line 6
   E. Line 7
   F. Line 8
   G. All of the lines compile.
6. Which annotations, when applied independently, allow the following program to compile? (Choose all that apply.)

```
import java.lang.annotation.*;
@Documented @Deprecated
public @interface Driver {
   int[] directions();
   String name() default "";
}
_____________________ class Taxi {}
```

A. `@Driver`
B. `@Driver(1)`
C. `@Driver(3,4)`
D. `@Driver({5,6})`
E. `@Driver(directions=7)`
F. `@Driver(directions=8,9)`
G. `@Driver(directions={0,1})`
H. None of the above

7. Annotations can be applied to which of the following? (Choose all that apply.)
   A. Class declarations
   B. Constructor parameters
   C. Local variable declarations
   D. Cast operations
   E. Lambda expression parameters
   F. Interface declarations
   G. None of the above
8. Fill in the blanks with the correct answers that allow the entire program to compile. (Choose all that apply.)

```
@interface FerociousPack {
   ____________________;           // m1
}

@Repeatable(____________________)  // m2
public @interface Ferocious {}

@Ferocious @Ferocious class Lion {}
```

A. `Ferocious value()` on line `m1`.

B. `Ferocious[] value()` on line `m1`.

C. `Object[] value()` on line `m1`.

D. `@FerociousPack` on line `m2`.

E. `FerociousPack` on line `m2`.

F. `FerociousPack.class` on line `m2`.

G. None of the above. The code will not compile due to its use of the `Lion` class.

9. What properties must be true to use an annotation with an element value, but no element name? (Choose all that apply.)

A. The element must be named `values()`.

B. The element must be required.

C. The annotation declaration must not contain any other elements.

D. The annotation must not contain any other values.

E. The element value must not be array.

F. None of the above

10. Which statement about the following code is correct?

```
import java.lang.annotation.*;
@Target(ElementType.TYPE) public @interface Furry {
   public String[] value();
   boolean cute() default true;
}
class Bunny {
   @Furry("Soft") public static int hop() {
      return 1;
   }
}
```

A. The code compiles without any changes.

B. The code compiles only if the type of `value()` is changed to a `String` in the annotation declaration.

C. The code compiles only if `cute()` is removed from the annotation declaration.

D. The code compiles only if `@Furry` includes a value for `cute()`.

E. The code compiles only if `@Furry` includes the element name for `value`.

F. The code compiles only if the value in `@Furry` is changed to an array.

G. None of the above

11. What properties of applying `@SafeVarargs` are correct? (Choose all that apply.)

A. By applying the annotation, the compiler verifies that all operations on parameters are safe.

B. The annotation can be applied to `abstract` methods.

C. The annotation can be applied to method and constructor declarations.

D. When the annotation is applied to a method, the method must contain a varargs parameter.

E. The annotation can be applied to method and constructor parameters.

F. The annotation can be applied to `static` methods.

12. Which of the following lines of code do not compile? (Choose all that apply.)

```
1: import java.lang.annotation.*;
2: enum UnitOfTemp { C, F }
3: @interface Snow { boolean value(); }
4: @Target(ElementType.METHOD) public @interface Cold {
5:    private Cold() {}
6:    int temperature;
7:    UnitOfTemp unit default UnitOfTemp.C;
8:    Snow snow() default @Snow(true);
9: }
```

A. Line 4

B. Line 5

C. Line 6

D. Line 7

E. Line 8

F. All of the lines compile.

13. Which statements about an optional annotation are correct? (Choose all that apply.)
    A. The annotation declaration always includes a default value.
    B. The annotation declaration may include a default value.
    C. The annotation always includes a value.
    D. The annotation may include a value.
    E. The annotation must not include a value.
    F. None of the above
14. Fill in the blanks: The _____ annotation determines whether annotations are discarded at runtime, while the _____ annotation determines whether they are discarded in generated Javadoc.
    A. `@Target` , `@Deprecated`
    B. `@Discard` , `@SuppressWarnings`
    C. `@Retention` , `@Generated`
    D. `@Retention` , `@Documented`
    E. `@Inherited` , `@Retention`
    F. `@Target` , `@Repeatable`
    G. None of the above
15. What statement about marker annotations is correct?
    A. A marker annotation does not contain any elements or constant variables.
    B. A marker annotation does not contain any elements but may contain constant variables.
    C. A marker annotation does not contain any required elements but may include optional elements.
    D. A marker annotation does not contain any optional elements but may include required elements.
    E. A marker annotation can be extended.
16. Which options, when inserted into the blank in the code, allow the code to compile without any warnings? (Choose all that apply.)

```
import java.util.*;
import java.lang.annotation.*;
public class Donkey {
   _____________________
   public String kick(List… t) {
```

```
            t[0] = new ArrayList();
            t[0].add(1);
            return (String)t[0].get(0);
        }
    }
```

A. `@SafeVarargs`
B. `@SafeVarargs("unchecked")`
C. `@Inherited`
D. `@SuppressWarnings`
E. `@SuppressWarnings("ignore")`
F. `@SuppressWarnings("unchecked")`
G. None of the above

17. What motivations would a developer have for applying the `@FunctionalInterface` annotation to an interface? (Choose all that apply.)

A. To allow the interface to be used in a lambda expression
B. To provide documentation to other developers
C. To allow the interface to be used as a method reference
D. There is no reason to use this annotation.
E. To trigger a compiler error if the annotation is used incorrectly

18. Which of the following lines of code do not compile? (Choose all that apply.)

```
1: @interface Strong {
2:    int force(); }
3: @interface Wind {
4:    public static final int temperature = 20;
5:    Boolean storm() default true;
6:    public void kiteFlying();
7:    protected String gusts();
8:    Strong power() default @Strong(10);
9: }
```

A. Line 2
B. Line 4
C. Line 5

D. Line 6

E. Line 7

F. Line 8

G. All of the lines compile.

19. Which annotations can be added to an existing method declaration but could cause a compiler error depending on the method signature? (Choose all that apply.)

A. `@Override`

B. `@Deprecated`

C. `@FunctionalInterface`

D. `@Repeatable`

E. `@Retention`

F. `@SafeVarargs`

20. Given the `Floats` annotation declaration, which lines in the `Birch` class contain compiler errors? (Choose all that apply.)

```
// Floats.java
import java.lang.annotation.*;
@Target(ElementType.TYPE_USE)
public @interface Floats {
   int buoyancy() default 2;
}

// Birch.java
1: import java.util.function.Predicate;
2: interface Wood {}
3: @Floats class Duck {}
4: @Floats
5: public class Birch implements @Floats Wood {
6:    @Floats(10) boolean mill() {
7:       Predicate<Integer> t = (@Floats Integer a) -> a> 10;
8:       return (@Floats) t.test(12);
9:    } }
```

A. Line 3

B. Line 4

C. Line 5

D. Line 6

E. Line 7

F. Line 8

G. None of the above. All of the lines compile without issue.

21. Fill in the blanks: The _____ annotation determines what annotations from a superclass or interface are applied, while the _____ annotation determines what declarations an annotation can be applied to.

A. `@Target`, `@Retention`

B. `@Inherited`, `@ElementType`

C. `@Documented`, `@Deprecated`

D. `@Target`, `@Generated`

E. `@Repeatable`, `@Element`

F. `@Inherited`, `@Retention`

G. None of the above

22. Which annotation can cancel out a warning on a method using the `@Deprecated` API at compile time?

A. `@FunctionalInterface`

B. `@Ignore`

C. `@IgnoreDeprecated`

D. `@Retention`

E. `@SafeVarargs`

F. `@SuppressWarnings`

G. None of the above

23. The `main()` method in the following program reads the annotation `value()` of `Plumber` at runtime on each member of `Team`. It compiles and runs without any errors. Based on this, how many times is `Mario` printed at runtime?

```
import java.lang.annotation.*;
import java.lang.reflect.Field;
@interface Plumber {
   String value() default "Mario";
}

public class Team {
```

```
   @Plumber("") private String foreman = "Mario";
   @Plumber private String worker = "Kelly";
   @Plumber("Kelly") private String trainee;

   public static void main(String[] args) {
      var t = new Team();
      var fields = t.getClass().getDeclaredFields();
      for (Field field : fields)
         if(field.isAnnotationPresent(Plumber.class))
            System.out.print(field.getAnnotation(Plumber.class)
               .value());
   }
}
```

A. Zero

B. One

C. Two

D. Three

E. The answer cannot be determined until runtime.

24. Which annotations, when applied independently, allow the following program to compile? (Choose all that apply.)

```
public @interface Dance {
   long rhythm() default 66;
   int[] value();
   String track() default "";
   final boolean fast = true;
}
class Sing {
   _______________ String album;
}
```

A. `@Dance(77)`

B. `@Dance(33, 10)`

C. `@Dance(value=5, rhythm=2, fast=false)`

D. `@Dance(5, rhythm=9)`

E. `@Dance(value=5, rhythm=2, track="Samba")`

F. `@Dance()`

G. None of the above

25. When using the `@Deprecated` annotation, what other annotation should be used and why?
    A. `@repeatable` , along with a containing type annotation
    B. `@retention` , along with a location where the value should be discarded
    C. `@deprecated` , along with a reason why and a suggested alternative
    D. `@SuppressWarnings` , along with a cause
    E. `@Override` , along with an inherited reference

Support Sign Out

©2022 O'REILLY MEDIA, INC. TERMS OF SERVICE PRIVACY POLICY