CHAPTER 1

■ ■ ■

Annotations

QUESTIONS AND EXCERCISES

1. What are annotations? How do you declare them?

   Answer:

   Annotations are types in Java, used to associate information to the declarations of program elements or type uses in a Java program.

   The `interface` keyword preceded by the `@` sign is used to declare an annotation type. The general syntax is to declare an annotation is as follows:

```
[modifiers] @interface <annotation-type-name> {
    // Annotation type body goes here
}
```

   The following is a declaration for an interface named `Version`. The `Version` interface contains two elements named `major` and `minor`.

```
public @interface Version {
    int major();
    int minor();
}
```

2. What are meta-annotations?

   Answer:

   Meta-annotations are annotations used to annotate other annotations.

3. What is the difference between an annotation type and annotation instances?

Answer:

An annotation type is a type like an interface. It can be used as variable type or even be implemented. An annotation instance is an instance of an annotation type.

4. Can you inherit an annotation type from another annotation type?

Answer:

No. An annotation type cannot inherit from another annotation type. Every annotation type implicitly inherits from the `java.lang.annotation.Annotation` interface,

5. What are marker annotations? Describe their use. Name two marker annotations available in Java SE API.

Answer:

A marker annotation type is an annotation type that does not declare any elements, not even one with a default value. Two of annotation type available in the Java SE API are `java.lang.Override` and `java.lang.Deprecated`.

6. Name the annotation type whose instances are used to annotate an overridden method. What is the fully qualified name of this annotation type?

Answer:

`Override` and `java.lang.Override`

7. What are the allowed return types for methods in an annotation type declaration?

Answer:

The return type of a method declared in an annotation type must be one of the following types:

- Any primitive type: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`
- `java.lang.String`

- `java.lang.Class`
- An `enum` type
- An annotation type
- An array of any of the above mentioned type, for example, `String[]`, `int[]`, etc. The return type cannot be a nested array. For example, you cannot have a return type of `String[][]` or `int[][]`.

8. Declare an annotation type named `Table`. It contains one `String` element named `name`. The sole element does not have any default value. This annotation must be used only on classes. Its instances should be available at runtime.

Solution:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Table {
     String name();
}
```

9. What is wrong with the following annotation type declaration?

```
public @interface Version extends BasicVersion {
    int extended();
}
```

Answer:

An annotation type cannot inherit from another annotation type. `Version` cannot extend `BasicVersion`.

10. What is wrong with the following annotation type declaration?

```
public @interface Author {
    void name(String firstName, String lastName);
}
```

Answer:

There are two problems with this declaration:

- Return type `void` is not allowed in an annotation type declaration.
- Method declarations cannot specify any format parameters in an annotation declaration.

11. Briefly describe the use of the following built-in meta-annotations: `Target`, `Retention`, `Inherited`, `Documented`, `Repeatable`, and `Native`.

Answer:

`Target`: Specifies the context in which the annotation type can be used. For ex., method, type, etc.

`Retention`: Specifies how an annotation instance of an annotation type should be retained by Java, source code only, class file only or class file & runtime.

`Inherited`: Indicates that its instances are inherited by a subclass declaration. It is effective only for class declaration.

`Documented`: Indicates that the Javadoc tool will generate documentation for all of its instances.

`Repeatable`: Allows repetition of an annotation in the same context.

12. Declare an annotation type named `ModuleOwner`, which contains one element `name`, which is of the `String` type. The instances of the `ModuleOwner` type should be retained only in the source code and they should be used only on module declarations.

Solution:

```
@Target(ElementType.MODULE)
@Retention(RetentionPolicy.SOURCE)
public @interface ModuleOwner {
     String name();
}
```

13. Declare a repeatable annotation type named `Author`. It contains two elements of `String` type: `firstName` and `lastName`. This annotation can be used on types,

methods, and constructors. Its instances should be available at runtime. Name the containing annotation type for the `Author` annotation type as `Authors`.

Solution:

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Authors.class)
public @interface Author {
    String firstName();
    String lastName();
}

@Target({ElementType.TYPE, ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface Authors {
    Author[] value();
}
```

14. What annotation type do you use to deprecate your APIs? Describe all the elements of such an annotation type.

Answer:

The `java.lang.Deprecated` annotation type is used to deprecate APIs. It has two elements:

- `String since() default "";`
- `boolean forRemoval() default false;`

The `since` element specifies the version in which the annotated API element became deprecated.

The `forRemoval` element indicates that the annotated API element is subject to removal in a future release and you should migrate away from the API.

15. What annotation type do you use to annotate a functional interface?

Answer:

`java.lang.FunctionalInterface`

16. How do you annotate a package?

Answer:

The annotated package declaration is placed in a file named `package-info.java`. The following contents in a `package-info.java` file for a package named `com.jdojo.test` annotates the package with a `@Version` annotation.

```
// package-info.java
@Version(major=1, minor=0)
package com.jdojo.test;
```

17. Create an annotation type named `Owner`. It should have one element, `name`, of the `String` type. Its instances should be retained at runtime. It should be repeatable. It should be used only on types, methods, constructors, and modules. Create a module named `jdojo.annotation.test` and create a class named `Test` in the `com.jdojo.annotation.exercises` package. Add a constructor and a method to the class. Annotate the class, its module, constructor, and method with the `Owner` annotation type. Add a `main()` method to the `Test` class and write code to access and print the details of these instances of the `Owner` annotation.

Solution:

```
// module-info.java
@Owner(“Owner Test Module”)
module jdojo.annotation.test {
}


// package-info.java
@Owner(name = "Owner Test Package")
package com.jdojo.annotation.exercises;


// Owner.java
package com.jdojo.annotation.exercises;

import java.lang.annotation.ElementType;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.MODULE, ElementType.TYPE, ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Owners.class)
public @interface Owner {
    String name();
}
```

```
// Owners.java
package com.jdojo.annotation.exercises;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.MODULE, ElementType.TYPE, ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface Owners {
    Owner[] value();
}


// Test.java
package com.jdojo.annotation.exercises

import java.lang.annotation.Annotation;
import java.lang.reflect.AnnotatedElement;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

@Owner(“Owner Test Class”)
public class Test {
    @Owner(“Owner Test Constructor”)
    public Test {
    }

    @Owner(“Owner Test Method”)
    public void testMethod() {
    }

     public static void main(String[] args) {
        Class cls = Test.class;
        Package p = cls.getPackage();
        System.out.println("Annotations for package: " + p.getName());
        printAnnotations(p);

        System.out.println("Annotations for class: " + cls.getName());
        printAnnotations(cls);

        System.out.println("Annotations for constructor: " );
        Constructor[] conList = cls.getConstructors();
        for (Constructor con : conList)
            printAnnotations(con);

        Method[] methodList = cls.getDeclaredMethods();
        for (Method method : methodList) {
            System.out.println("Annotations for method: " + method.getName());
            printAnnotations(method);
        }
```

```
    }

    private static void printAnnotations(AnnotatedElement anEle) {
        Annotation[] allAnns = anEle.getAnnotations();
        for (Annotation ann : allAnns) {
             System.out.println("\t" + ann);
        }
    }
}
```

18. Consider the following declaration of an annotation type named `Status`:

```
public @interface Status {
    boolean approved() default false;
    String approvedBy();
}
```

Later you need to add another element to the `Status` annotation type. Modify the declaration of the annotation to include a new element named `approvedOn`, which is of the `String` type. The new element will contain a date in ISO format whose default value may be set to `"1900-01-01"`.

Solution:

Updated the `Status` annotation declaration as follows::

```
public @interface Status {
    boolean approved() default false;
    String approvedBy();
    String approvedOn() default "1900-01-01";
}
```

19. Consider the declaration of the following annotation type named `LuckyNumber`:

```
public @interface LuckyNumber {
    int[] value() default {19};
}
```

Which of the following uses of the `LuckyNumber` annotation type is/are invalid? Explain your answer.

```
a) @LuckyNumber
b) @LuckyNumber({})
c) @LuckyNumber(10)
```

```
d) @LuckyNumber({8, 10, 19, 28, 29, 26})
e) @LuckyNumber(value={8, 10, 19, 28, 29, 26})
f) @LuckyNumber(null)
```

Answer:

```
a) @LuckyNumber: Valid. Uses default value
b) @LuckyNumber({}): Valid. Setting empty int[]
c) @LuckyNumber(10): Valid. Passing valid int value (which will be added to int[])
d) @LuckyNumber({8, 10, 19, 28, 29, 26}): Valid. Setting valid int[]
e) @LuckyNumber(value={8, 10, 19, 28, 29, 26}): Valid. Setting valid int[] to value
f) @LuckyNumber(null): Invalid. Cannot use null reference as a value
```

20. Given a `LuckyNumber` annotation type, is the following variable declaration valid?

```
LuckNumber myLuckNumber = null;
```

Solution:

Yes, this is a valid variable declaration because an *annotation type* is a type like an interface.

21. Consider the following declaration for a `jdojo.annotation.exercises` module:

```
module jdojo.annotation.exercises {
    exports com.jdojo.annotation.exercises;
}
```

The module exists since version 1.0. The module has been deprecated and will be removed in the next version. Annotate the module declaration to reflect these pieces of information.

Solution:

```
@Deprecated(since="1.0", forRemoval=true)
module jdojo.annotation.exercises {
    exports com.jdojo.annotation.exercises;
}
```