Annotations 25

Chapter Topics

- Understanding the role of annotations to specify metadata in the source code
- Declaring annotation types: marker annotation type, single-element annotation type, and multi-element annotation type
- Understanding the implications of declaring a `value()` element and default values in annotation types
- Applying annotations as instantiations of annotation type declarations with notations for normal, marker, and single-element annotations
- Understanding how the `value()` element and default values in an annotation type declaration influence its application in the source code
- Applying predefined meta-annotations in annotation type declarations to specify the retention (`@Retention`), the target (`@Target`), the inheritance (`@Inherited`), the documentation (`@Documented`), and the repeatability (`@Repeatable`) of an annotation type declaration
- Writing code to demonstrate use of standard predefined annotations to validate overriding (`@Override`) and functional interfaces (`@FunctionalInterface`), annotate deprecated code (`@Deprecated`), and suppress warnings (`@SuppressWarnings` , `@SafeVarargs`)
- Writing code to use the Reflection API to process annotations in a program at runtime

Java SE 17 Developer Supplementary Topic

[12.2] Use Annotations such as Override, Functionalnterface, Deprecated, SuppressWarnings, and SafeVarargs.	§25.5, p. 1577

- *Note that this is a supplementary objective in the description of the Java SE 17 Developer exam.*

Java SE 11 Developer Exam Objectives

[13.1] Create, apply, and process annotations	§25.1, p. 1557 *to* §25.6, p. 1587

Annotations allow information in the form of metadata to be added to the source code. Annotations provide information about the program, but are not considered to be a part of the program. Syntactically, annotations are a special kind of interfaces that are attached to various program elements in the source code, such as classes, variables, and methods. Tools can then process these annotations at the source level or process the class files into which they are compiled. Without such processing, the annotations do not actually do anything. The benefit of using annotations is only realized when appropriate tools are used to process them and analyze the metadata they represent. There are some existing annotations that are processed by the Java compiler, and even more annotation support is provided by various frameworks like Java EE/Jakarta and the Spring Framework.

It is important to note that annotations are optional and do not change the logic of the program. Augmenting the source code with annotations does not change the bytecode generated for the program. Thus the presence of annotations has no effect on the execution of the program by the JVM, only that they may be present in the class file.

This chapter covers declaring annotation types and applying annotations and utilizing the predefined annotations provided by the Java SE platform. It also provides a taste of processing annotations using the Reflection API.

25.1 Basics of Annotations

Support for annotations is provided by the Annotation API, which is located in the `java.lang.annotation` package. The contents of this package are used to declare annotation types—as we shall see in this chapter.

As subpackages are not automatically imported, the types in the subpackage `java.lang.annotation` of the `java.lang` package must be explicitly imported in order to access them by their simple names—for example, by including a type-import-on-demand statement:

Click here to view code image

```
import java.lang.annotation.*;
```

Annotations are types that are a special kind of interfaces, analogous to enum types that are a special kind of class. An annotation must be declared before it can be used: An *annotation type declaration* must be defined, and when *applied* to a program element, an *annotation* designates a specific invocation of an annotation type. The `java.lang.annotation.Annotation` interface is a common interface implicitly extend by all annotation types, analogous to all classes implicitly extending the `Object` class. However, an annotation type declaration cannot explicitly extend the `Annotation` interface. Typically, an annotation type is declared as a top-level type, but can be nested like static member types. An annotation type declaration is compiled as any other Java type declaration, resulting in a class file containing its bytecode.

Click here to view code image

```
// Annotation type declaration with the name Tag.
@interface Tag {}              // Declaration always specified with
                               // at-sign (@) preceding the keyword interface.
_____________________________________________________________________________
// Applying annotation @Tag which denotes the annotation type declaration Tag.
@Tag class Gizmo {            // Always applied with at-sign (@) preceding
                              // the annotation type name.
  @Tag void start() {}        // Another invocation of the annotation type Tag.
}
```

The code above shows the minimalistic declaration of an annotation type named `Tag` and its application to a class named `Gizmo`. What is important to note is that an annotation type declaration is specified with the at-sign (`@`) preceding the keyword `interface`, and when applied to a program element, the at-sign (`@`) precedes the annotation type name, and each application of the annotation to any program element denotes a different invocation of the annotation type. The rest of this chapter will elaborate on defining annotation type declarations and applying annotations to various program elements in the source code. The last section provides an introduction to *annotation processing*.

25.2 Declaring Annotation Types

An *annotation type declaration* specifies a new *annotation type*, and has a lot in common with an interface type. General syntax of an annotation type declaration is shown below.

Click here to view code image

```
meta-annotations access_modifier @interface annotation_name // Annot type header
{ // Annotation type body with zero or more member declarations:

  annotation_type_element_declarations

  constant_variable_declarations
```

```
  enum_declarations
  class_declarations
  interface_declarations
}
```

We will use the annotation type declaration `TaskInfo` below to explain the syntax of an annotation type declaration. The `TaskInfo` annotation type allows meta-information to be specified about tasks on program elements: a description of the task, people the task is assigned to, and the priority given to the task. The `TaskInfo` annotation type is meant to illustrate declaring annotation types, and is in no way meant to replace task management tools.

Click here to view code image

```
import java.lang.annotation.Target;                // (1)
import java.lang.annotation.ElementType;           // (2)
import java.lang.annotation.Retention;             // (3)
import java.lang.annotation.RetentionPolicy;       // (4)

@Target({ElementType.TYPE, ElementType.METHOD})    // (5) Meta-annotation
@Retention(RetentionPolicy.RUNTIME)                // (6) Meta-annotation
public @interface TaskInfo {                       // (7)
  String taskDesc();                               // (8) Annotation type element
  String[] assignedTo();                           // (9) Annotation type element
  TaskPriority priority() default TaskPriority.NORMAL;  // (10) Annot type element

  public enum TaskPriority { LOW, NORMAL, HIGH };       // (11) Nested enum type

  public static final String LOG_FILE = "./logs/Tasks.log"; // (12) const decl.
}
```

The *meta-annotations* specified in the annotation type header are annotations that are applied to the annotation type declaration to specify certain aspects of the declaration—that is, they allow metadata to be associated with the annotation type declaration (**p. 1567**). In the code above, these meta-annotations are defined by classes in the `java.lang.annotation` package, which are imported at (1) and (3), together with auxiliary enum types at (2) and (4). The meta-annotation `@Target` at (5) specifies which program elements the annotation type can be applied to. The targets specified for the `TaskInfo` annotation type are any type declaration (`Element-Type.TYPE`) and methods (`ElementType.METHOD`). The `@Retention` meta-annotation specifies the retention policy: whether invocations of the annotation should be retained with the program element in the source file or in the class file, or made available at runtime. Applications of the `TaskInfo` annotation type on program elements will be available at runtime (`RetentionPolicy.RUNTIME`).

The *access modifier* that can be specified in the annotation type header is `public`, or there is no access modifier to indicate package accessibility—the same as with a normal interface. The `TaskInfo` annotation type is declared `public`.

Like any normal interface, an annotation type declaration is implicitly abstract, but seldom explicitly specified as such. Note also that an annotation type declaration cannot be generic and the extends clause is not permitted—in contrast to a normal interface.

The *simple name* of an annotation type shares the same namespace as simple names of normal classes and interfaces in a package—that is, a simple name of an annotation type should not conflict with any type declaration having the same simple name in the package.

The *annotation type body* can contain zero or more *member declarations*. These member declarations are analogous to the member declarations in a normal interface when it comes to constant variable declarations (which are implicitly public, static, and final) and nested type member declarations—but default, static, and private methods are not permitted. For example, the `TaskInfo` type annotation declaration declares a nested enum type named `TaskPriority` at (11) and a constant variable named `LOG_FILE` at (12).

In the rest of this section, we take a closer look at declaration of annotation type elements in the body of an annotation type, declared at (8), (9), and (10), as values provided for these annotation type elements when an annotation is applied to a program element determine the metadata that is associated with the program element.

Declaring Annotation Type Elements

The body of an *annotation type declaration* may contain declarations of *annotation type elements*. The syntax of an annotation type element declaration is shown below.

Click here to view code image

```
element_modifiers element_type method_name() default constant_expression;
```

A partial declaration of the `TaskInfo` annotation type is repeated below, showing the declaration of the three annotation type elements at (8), (9), and (10).

Click here to view code image

```
// ...
public @interface TaskInfo {                    // (7)
  String taskDesc();                            // (8) Required annotation element
  String[] assignedTo();                        // (9) Required annotation element
  TaskPriority priority() default TaskPriority.NORMAL;  // (10) Optional
                                                        // annotation element
  // ...
}
```

Each annotation type element specifies a *method declaration*, resembling the abstract method declaration in a normal interface. Each annotation type element is usually referred to as an *element* of the annotation type. The elements declared in an annotation type declaration constitute the *attributes* of the annotation type.

The *method declaration* in an element has a *return type* and a *method name*, and is *always* specified with an *empty formal parameter list*, as exemplified by the method declarations in the elements at (8), (9), and (10) in the `TaskInfo` annotation type declaration. Note that no formal parameters, type parameters, or a throws clause can be specified for a method declaration in an element. The name and the return type of the method declared in an element are referred to as *the name and the type of the element*, respectively. Only specific types can be declared as the type of an element (p. 1561).

Annotation types are characterized by the number of elements declared in their declaration.

- *A marker annotation type* is an annotation type with no elements, analogous to a marker interface. A marker annotation is typically used to indicate a specific purpose that does not require any elements. Earlier in this chapter we have seen the annotation type `Tag`, which is a marker annotation type. The notation for applying such an annotation type to program elements is straightforward: an at-sign (`@`) preceding the annotation name (p. 1565)—for example, `@Tag`. Ample examples of marker annotation types can be found throughout this chapter.
  Click here to view code image

  ```
  @interface Tag {} // No element type declarations.
  ```

- *A single-element annotation type*, as the name suggests, is an annotation type with a single element. By convention, the method of a single-element annotation type is declared as `value()`. Various shorthand notations are offered when applying such annotation types to program elements (p. 1565).
  Click here to view code image

```
@interface SecurityLevel {              // Single-element type declaration
   int value();
}
```

- *A multi-element annotation type*, as the name suggests, is an annotation type with more than one element. The `TaskInfo` annotation type is a multi-element annotation type, as it declares three elements. At the most, one method of an element in a multi-element annotation type can be declared as `value()`. Under certain conditions, a shorthand notation can be used when applying an annotation type declaration that has a `value()` element (p. 1563).

Element Modifiers for Annotation Type Elements

The modifiers allowed for an annotation type element are analogous to the modifiers that can be specified for an abstract method declaration in a normal interface. The only *access modifier* that can be specified for an annotation type element is `public`; otherwise, the accessibility is implicitly public. The modifier `abstract` is also implied, and typically not specified in the annotation type element declaration. The elements of the `TaskInfo` annotation type are implicitly `public` and `abstract`.

Element Type of Annotation Type Elements

The *element type* of an annotation type element can be any one of the following:

- A primitive type
- A `String`
- The class `Class`
- An enum type
- An annotation type
- A one-dimensional array whose element type is one of the above types

The code below illustrates declaring the element type of an annotation type element. An example of each type that is allowed as the return type of an annotation type element is shown, together with the specification of a default value for each annotation type element.

Click here to view code image

```
public @interface MultiElementAnnotationType {

  public enum Priority { LOW, NORMAL, HIGH };

  public int certificationLevel()     default 1;                        // int
  String date()                       default "2021-01-11";   // String
  Class<? extends PrettyPrinter> pp()
                          default AdvancedPrettyPrinter.class;          // type Class
  Priority priorityLevel()              default Priority.NORMAL;       // enum Priority
  Tag annotate()                        default @Tag;                  // Annotation type
  int[] value()                         default {10, 20, 30};          // Array, int[]
}
// Auxiliary classes:
class PrettyPrinter {}
class AdvancedPrettyPrinter extends PrettyPrinter {}
```

The declaration of the `ProblematicAnnotationType` annotation type below shows examples of illegal declarations of annotation type elements.

Click here to view code image

```
@interface ProblematicAnnotationType {
  StringBuilder message();              // Illegal return type.
  int[][] voting();                     // Only one-dimensional array allowed.
  String value;                         // Missing parentheses.
```

```
  private Thread.State state();         // Only public can be specified.
}
```

The code below shows an annotation which is used as an element type within another annotation. This relationship between annotations is sometimes described as a *contained annotation type* (1) and a *containing annotation type* (2). This relationship is further explored when we discuss repeatable annotations (**p. 1575**).

Click here to view code image

```
public @interface MusicMeta {             // (1) Contained annotation type
  String value();
}
public @interface ArtistMeta {            // (2) Containing annotation type
  MusicMeta value();                      // Annotation type from (1).
}
```

Defaults for Annotation Type Elements

An annotation type element declaration can declare a *default value* for an element using the optional `default` clause in the annotation type element declaration, as shown in the declaration of the `MultiElementAnnotationType` earlier. The default value is specified in the `default` clause as a *constant expression*—that is, the operands of the expression are all constants so that the compiler is able to determine its value at compile time. The constant expression must not be `null`. The declaration of the `Refactor` annotation type below shows examples of illegal default values.

Click here to view code image

```
@interface Refactor {
  int id()          default (int) (Math.random() * 10);// Not a const expression.
  String value()    default null;                      // Cannot be null.
  String deadline() default new String("2021-01-11");  // Not a const expression.
  String[] team()   default new String[] {"VJ", "PT"}; // Not a const expression.
}
```

An element that specifies a default value is called an *optional annotation element*. An element that does *not* specify a `default` clause is called a *required annotation element*. As we shall see when an annotation is applied to a program element, specifying a value for an optional annotation element can be omitted, in which case the default value in the declaration of the annotation type element is used. For a required annotation element, a value for the element *must* be specified when the annotation is applied to a program element. We see that the declaration of the `TaskInfo` annotation type below has two required annotation elements, at (8) and at (9), and one optional annotation element at (10).

Click here to view code image

```
// ...
public @interface TaskInfo {                    // (7)
  String taskDesc();                            // (8) Required annotation element

  String[] assignedTo();                        // (9) Required annotation element
  TaskPriority priority() default TaskPriority.NORMAL; // (10) Optional
                                                //      annotation element
  // ...
}
```

25.3 Applying Annotations

Once an annotation type has been compiled, it can be applied to program elements. An annotation is an application of an annotation type to a program element, and we refer to the annotation as being of that type. Which program elements an annotation can be applied to is typically determined by the meta-annotation `@Target` specified in the declaration of the annotation type (p. 1569). Each annotation refers to a specific invocation of an annotation type. An annotation is always specified with the at-sign (`@`) preceding the name of the annotation type, and usually provides values for the elements of the annotation type.

Multiple annotations of different types can be applied to a program element. Repeatable annotations are covered later (p. 1575).

Annotations can be characterized according to the *notation* that can be used when applied to a program element. We distinguish between three kinds of annotations:

- *Normal annotations*
- *Marker annotations*
- *Single-element annotations*

The last two annotations are special cases of normal annotations. Choosing which annotation to use depends on knowing what annotation type elements are declared in an annotation type declaration. The different kinds of annotations are discussed in this section.

Applying Normal Annotations

When a *normal annotation* is applied to a program element, in addition to the name of the annotation type being preceded by the at-sign (`@`), it can optionally specify a comma-separated list of element-value pairs enclosed in parentheses, `()`. Each pair has the syntax *element* `=` *value*. For each pair, the *element* and the *value* must be compatible with the method name and the return type of a method specified in an annotation type element.

The normal annotation for the annotation type `TaskInfo` (p. 1558) is applied to a class below, where each element-value pair associates a value with an element of the annotation type `TaskInfo`.

Click here to view code image

```
@TaskInfo(                                                  // Normal annotation
   taskDesc = "Class for monitoring nuclear reactor activity",       // Required
   assignedTo = {"Tom", "Dick", "Harriet"},                          // Required
   priority = TaskInfo.TaskPriority.HIGH                             // Optional
   )
class NuclearPlant {}
```

A normal annotation must contain element-value pairs for all required annotation elements in the annotation type—that is, for elements that do not specify a default value. Specifying the element-value pairs for optional annotation elements is obviously optional.

The order of the element-value pairs is irrelevant, but usually they are specified in the same order that the elements have in the annotation type declaration. Note also that a `null` value in an element-value pair will result in a compile-time error.

In the normal annotation below, the optional annotation element `priority` in the annotation type `TaskInfo` is omitted.

Click here to view code image

```
@TaskInfo(                                                          // Normal
annotation
   taskDesc = "Class for monitoring nuclear reactor activity",       // Required
```

```
   assignedTo = {"Tom", "Dick", "Harriet"}                          // Required
   )
class NuclearPlant {}
```

The above normal annotation is then equivalent to the normal annotation below, where the default value `TaskInfo.TaskPriority.NORMAL` of the optional annotation element `priority` is implied.

Click here to view code image

```
@TaskInfo(                                                          // Normal
annotation
   taskDesc = "Class for monitoring nuclear reactor activity",      // Required
   assignedTo = {"Tom", "Dick", "Harriet"},                         // Required
   priority = TaskInfo.TaskPriority.NORMAL                          // Implied
   )
class NuclearPlant {}
```

An *array element initializer*, $\{v_1, \ldots, v_n\}$, can be used to specify the values of array elements for annotation elements whose type is an array type. The curly brackets can be omitted when specifying the value for a *single-element array-valued element-value pair*—that is, where the type of the annotation type element is an array type, but only a single array element is specified in the annotation.

Click here to view code image

```
@TaskInfo(
    priority = TaskInfo.TaskPriority.LOW,
    taskDesc = "Start nuclear reactor",
    assignedTo = "Harriet"               // Single-element array-valued element
    )
class NuclearPlant {}
```

Applying Marker Annotations

A *marker annotation* is a shorthand notation typically for use with marker annotation types, which have no elements in their type declaration.

Click here to view code image

```
// Marker annotation type
@interface Tag {}                       // No element type declarations
```

The marker annotation of annotation type `Tag` is applied to a class as shown below, with just the name of the annotation type preceded by the at-sign (`@`)—that is, the parentheses are omitted.

Click here to view code image

```
@Tag class Gizmo {}                     // Marker annotation
```

The normal annotation of annotation type `Tag` is applied to a class, where the empty list of element-value pairs is explicitly specified:

Click here to view code image

```
@Tag() class Gizmo {}                   // Normal annotation
```

A marker annotation can also be used if all elements are optional annotation elements in the annotation type declaration—that is, they specify default values.

Click here to view code image

```
// Annotation type declaration where all elements have default values.
@interface Option {
  Color color()  default Color.WHITE;   // Optional annotation element
  Size size()    default Size.M;        // Optional annotation element
  enum Color {RED, WHITE, BLUE}
  enum Size  {S, M, L, XL}
}
```

A marker annotation of annotation type `Option` is applied to the class below:

Click here to view code image

```
@Option                                                 // Marker annotation
class Item {}
```

The above marker annotation is equivalent to the following normal annotation:

Click here to view code image

```
@Option(color = Option.Color.WHITE, size = Option.Size.M)  // Normal annotation
class Item {}
```

Applying Single-Element Annotations

A *single-element annotation* is a shorthand notation typically for use with single-element annotation types, whose only annotation type element is declared as a `value()` element. Single-element annotation will not work if the element in the single-element annotation type is not declared as a `value()` element. However, there is no requirement that the single element should be either required or optional. Note that there is no restriction on declaring a `value()` element for any annotation type, but that alone does not guarantee that single-element annotation can be used.

Click here to view code image

```
// Single-element annotation type
@interface Author {
  String value();                           // Single annotation type element
}
```

A single-element annotation of annotation type `Author` is applied to the class below, where only the value of the single `value()` element is specified:

Click here to view code image

```
@Author("Tom")                              // Single-element annotation
class Connection {}
```

The above single-element annotation is equivalent to the following normal annotation:

Click here to view code image

```
@Author(value = "Tom")                      // Normal annotation
class Connection {}
```

Single-element annotation can also be used if one element is declared as a `value()` element and *all other elements are optional annotation elements* in the annotation type declaration—that is, they specify default values.

Click here to view code image

```
//Annotation type declaration with a value() element and other elements with
default values.
@interface ExtraOption {
  int value();                              // Required annotation element
  Color color()  default Color.WHITE;       // Optional annotation element
  Size size()    default Size.M;            // Optional annotation element
  enum Color {RED, WHITE, BLUE}
  enum Size  {S, M, L, XL}
}
```

A single-element annotation of annotation type `ExtraOption` is applied to the class below:

Click here to view code image

```
@ExtraOption(10)                                  // Single-element annotation
class ItemV2 {}
```

The above single-element annotation is equivalent to the following normal annotation:

Click here to view code image

```
@ExtraOption(value=10, color=ExtraOption.Color.WHITE,      // Normal annotation
             size=ExtraOption.Size.M)
class ItemV3 {}
```

Following is an example of using *array-valued single-element annotation*—that is, where the element type of the single annotation type element is an array type:

Click here to view code image

```
// Array-valued single-element annotation type declaration:
@interface Problems {
  String[] value();                               // Array-type value() element
}
@Problems({"Code smell", "Exception not caught"})  // Array-valued single-element
class ItemV4 {}                                   // annotation

@Problems(value = {"Code smell", "Exception not caught"})  // Normal annotation
class ItemV5 {}
```

Using a *single-element array-valued single-element annotation*—that is, where the element type of the single `value()` element is an array type, but only a single array element is specified in the annotation—is shown below.

Click here to view code image

```
@Problems("Code smell")   // Single-element array-valued single-element annotation
class ItemV6 {}

@Problems({"Code smell"}) // Single-element array-valued single-element annotation
class ItemV7 {}

@Problems(value = "Code smell")      // Normal annotation
class ItemV8 {}
```

```
@Problems(value = {"Code smell"})    // Normal annotation
class ItemV9 {}
```

25.4 Meta-Annotations

Meta-annotations are annotations that can be applied on an annotation type declaration to specify various aspects of the annotation when this annotation type is applied in the source code. We discuss the purpose and usage of predefined meta-annotations when declaring an annotation type, as they have implications on how the annotation will be handled when applied in the source code.

`@Retention`: Specifies how this annotation is retained when applied, whether in the source file, in the class file, or at runtime.

`@Target`: Specifies to which contexts in the source code this annotation can be applied.

`@Inherited`: Indicates that all subclasses of a superclass that this annotation is applied to will inherit this annotation.

`@Repeatable`: Indicates that this annotation can be used multiple times in the same context.

`@Documented`: Indicates that this annotation should be included in the documentation generated by a tool like `javadoc`.

The `@Retention` Meta-Annotation

In the declaration of an annotation type, it is possible to specify how long the annotation will be retained, meaning whether it should be in the source file, in the class file, or available at runtime.

In fact, each annotation is associated with one of three retention policies, controlled by the meta-annotation type `java.lang.annotation.Retention` and the *retention policy constants* defined by the enum type `java.lang.annotation.RetentionPolicy` shown in **Table 25.1** in increasing order of retention.

The meta-annotation type `Retention` is a *single-element meta-annotation type* that specifies a `value()` element of enum type `RetentionPolicy`. Defining `value()` elements is covered elsewhere in this chapter (**p. 1560**).

Click here to view code image

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(value = RetentionPolicy.RUNTIME)   // Retention: In source code,
                                              // in class file, at runtime.
public @interface MusicMeta {}
```

The annotation type `MusicMeta` above needs to be retained at runtime so that it is possible to extract any information that is specified by this annotation. The `Retention-Policy.RUNTIME` enum constant provides the longest retention. If used in the source code, the `@MusicMeta` annotation will be recorded in the class file, and when loaded at runtime, it will be available through reflection. However, no provision has been made as yet in the `MusicMeta` annotation type declaration to provide any information. It is added to the annotation type declaration by specifying annotation type elements (**p. 1560**).

Click here to view code image

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

```
@Retention(value = RetentionPolicy.SOURCE) // Retention: Only in the source code.
public @interface Override {}
```

The code above shows the declaration of the `Override` annotation type. This builtin annotation type when applied in the source code can be used by the compiler (p. 1578). Its declaration specifies the `RetentionPolicy.SOURCE`, meaning that the annotation will only be retained in the source code. In other words, it will not be recorded in the class file or be available at runtime.

The annotation type declarations shown above do not contain any annotation type elements. They are *marker annotation types*. The purpose of a marker annotation type is to indicate the presence of a feature, rather than convey any specific information. As the `MusicMeta` annotation type declaration stands, it is possible to discover at runtime that this annotation is used on a particular program element and take the appropriate action, whatever that might be. Use of the `@Override` marker annotation in the source code alerts the compiler to check that the criteria for overriding is satisfied by the method on which this annotation is applied, but the annotation is not recorded in the class file.

Table 25.1 *Retention Policy Values for the* `@Retention` *Meta-Annotation*

Constants defined by the `RetentionPolicy` enum type	Description
`SOURCE`	This retention policy implies that the annotation is only retained in the source code and is discarded by the compiler—that is, it is not recorded in the class file. The purpose of this annotation is to provide information for the compiler—for example, to validate source code, detect design errors, and suppress compiler warnings.
`CLASS`	This retention policy implies that the annotation is recorded in the class file and thus retained by the compiler, but it is not intended to be used at runtime. That it, it need not be loaded by the JVM at runtime. The purpose of this annotation is to provide information for compile-time and deployment-time processing. It can be used by tools that process information to generate additional code—for example, deployment descriptors such as XML files used by Java EE/Jakarta. `RetentionPolicy.CLASS` is the default retention policy if no retention policy is specified on the annotation type declaration.
`RUNTIME`	This retention policy results in the annotation being recorded in the class file by the compiler. It is retained by the JVM and can be read at runtime via reflection.

The `@Target` Meta-Annotation

An annotation type declaration can specify the contexts in which the annotation is applicable. This is controlled by the meta-annotation type `java.lang.annotation.Target` and the *element type constants* defined by the enum type `java.lang.annotation.Element-Type`, shown in Table 25.2. The meta-annotation type `Target` is an *array-valued single-element meta-annotation type*—that is, it has a single `value()` element of type `ElementType[]` to specify contexts in which the annotation is applicable, meaning multiple `ElementType` values can be specified to indicate multiple contexts.

Table 25.2 *The* `ElementType` *Values for the* `@Target` *Meta-Annotation*

	Constants defined by the `ElementType` enum	Description
1.	`ANNOTATION_TYPE`	The annotation can be applied to annotation type declarations. `@Tag @interface Status {}`
2.	`CONSTRUCTOR`	The annotation can be applied to constructor declarations. `@Tag public Item() {}`

3.	`FIELD`	The annotation can be applied to field declarations, such as instance and static variables, or enum constants.
		Click here to view code image
		`@Tag public static final int YEAR = 2021;`
		`enum Size { @Tag S, @Tag M, @Tag L, @Tag XL }`

4.	`LOCAL_VARIABLE`	The annotation can be applied to local variable declarations, including loop variables of `for` statements and resource variables of `try`-with-resources statements.
		Click here to view code image
		`@Tag String localBarber = "Director's Cut";`
		`for (@Tag int i : intArray) {}`
		`try (@Tag var resource = new CloseableResource()) {}`

5.	`METHOD`	The annotation can be applied to method declarations (including elements of annotation types).
		Click here to view code image
		`@Tag boolean passed() { return true; }`
		`@Tag String[] value();`

6.	`MODULE`	The annotation can be applied to a module declaration in a `module-info.java` source file. `@Tag module com.passion {}`
7.	`PACKAGE`	The annotation can be applied to a package declaration in a source file. Best practice is to specify package-related annotations in a file named `package-info.java`. Click here to view code image `@Tag package com.passion.logic;`

	Constants defined by the `ElementType` enum	Description

8.	`PARAMETER`	The annotation can be applied to formal and exception parameter declarations in constructors, methods, lambda expressions, and `catch` blocks.
		Click here to view code image
		`double circleArea(@Tag double r) { return Math.PI*r*r; }`
		`Predicate<String> p = (@Tag var str) -> str.length() < 10;`
		`try {} catch (@Tag Exception ex) {}`

9.	`TYPE`	The annotation can be applied to type declarations: class, interface (including annotation type), and enum declarations.
		Click here to view code image
		`@Tag class Gizmo {}`
		`@Tag interface Reparable {}`
		`@Tag enum Direction {LEFT, RIGHT}`
		`@Tag @interface Validate {}`

10.	`TYPE_PARAMETER`	The annotation can be applied to type parameter declarations of generic classes, interfaces, methods, and constructors.
		Click here to view code image
		`class Box<@Tag E> {}`
		`interface Eatable<@Tag E> {}`
		`<@Tag E extends Comparable<E>> void comparison(E e) {}`
		`<@Tag E> Item(E e) {}`

11.	`TYPE_USE`	The annotation can be applied in any type context in declarations and expressions where a type can be used.
		Click here to view code image
		`class X extends @Tag Y implements @Tag IZ {}`
		`java.lang. @Tag String strOp() { return "ok"; }`
		`@Tag int compute() { return (@Tag int) 3.14; }`
		`java.lang. @Tag Integer iRef1 = 100;`
		`@Tag Integer iRef2 = 100;`
		`void passItOn() throws @Tag Exception {}`
		`@Tag int[] array1;  // Annotates primitive type int`
		`int @Tag [] array2;  // Annotates array type int[]`
		`int @Tag [][] array3;  // Annotates array type int[][]`

The target values of the `ElementType` constants in **Table 25.2** are categorized by the two contexts where annotations can be applied: *declaration context* and *type context*.

The target values of the `ElementType` constants from 1 to 10 in **Table 25.2** indicate *declaration contexts*—that is, a context where a program element is declared. Annotations with these target values are called *declaration annotations*. For example, an annotation type declaration with the target value `ElementType.CONSTRUCTOR` is only applicable to a constructor declaration.

Click here to view code image

```
@Target(ElementType.CONSTRUCTOR)
@interface Tag {}
———————————————————————
class Item {
  @Tag public Item() {}       // Declaration context. Annotates constructor Item.
}
```

However, an annotation type declaration with the target value `ElementType.PARAMETER` is applicable to a parameter declaration in contexts where such parameter declarations are permitted. Note that several target values can be specified in the `@Target` meta-annotation.

Click here to view code image

```
@Target({ElementType.CONSTRUCTOR, ElementType.PARAMETER})
@interface Tag {}
———————————————————————-
double circleArea(@Tag double r) { // Declaration context. Annotates parameter r.
  return 5.0;
}
```

Note also that the context of the target value `ElementType.ANNOTATION_TYPE` and the context of the target value `ElementType.TYPE` are overlapping, as the latter not only includes annotation type declarations, but also other reference type declarations (class, interface, and enum).

The target value `ElementType.TYPE_USE` indicates *type contexts*—that is, where a type is used in a declaration or an expression. Annotations with this target value are called *type annotations*.

Click here to view code image

```
@Target(ElementType.TYPE_USE)
@interface Tag {}
———————————————————————-
java.lang. @Tag Integer iRef1 = 100;        // (1) Type context. Annotates Integer.
@Tag Integer iRef2 = 100;                   // (2) Type context. Annotates Integer.
@Tag java.lang.Integer iRef3 = 100;        // (3) Declaration context.
                                                Compile-time error!
@Tag int compute() { return (@Tag int) 3.14; }// (4) Type context is int type.
@Tag void compute(int i) {}                   // (5) No return type specified.
                                              //     Compile-time error!
```

In both (1) and (2) above, `@Tag` is a type annotation on the type `Integer`, as this type is used in the declaration of the variables. The code at (3) does not compile, as the type annotation `@Tag` is used in a declaration context. Note how the position of the annotation changes the context of the annotation at (1) and (3). At (4), the type annotation `@Tag` is on the return type `int` of the non-`void` method. As the method at (4) is a `void` method, there is no return type that is applicable for the type annotation `@Tag`, so the compiler flags an error.

Consider the following annotation type with the target value `ElementType.FIELD`:

Click here to view code image

```
@Target(ElementType.FIELD)
@interface Tag {}
———————————————————————
java.lang. @Tag Integer iRef1 = 100; // (1a) Type context. Compile-time error!
@Tag Integer iRef2 = 100;              // (2a) Declaration context. Annotates iRef2.
@Tag java.lang.Integer iRef3 = 100;   // (3a) Declaration context. Annotates iRef3.
```

At both (2a) and (3a), the annotation `@Tag` is a declaration annotation on the field, as these are field declarations. The code at (1a) does not compile, as the declaration annotation `@Tag` is

used in a type context.

An annotation can be both a declaration and a type annotation, as illustrated by the code below.

Click here to view code image

```
@Target({ElementType.TYPE_USE, ElementType.FIELD})
@interface Tag {}
_____________________________________
java.lang. @Tag Integer iRef1 = 100; // (1b) Type context. Annotates Integer.
@Tag Integer iRef2 = 100;            // (2b) Declaration and type context.
@Tag java.lang.Integer iRef3 = 100;   // (3b) Declaration context. Annotates iRef3.
```

An `ElementType` value can only occur once in the `value()` element of the meta-annotation `@Target`. The following specification of the `value()` element of the meta-annotation `@Target` will not compile.

Click here to view code image

```
@Target(value = {ElementType.METHOD, ElementType.FIELD,
                 ElementType.METHOD})       // Compile-time error!
@interface Tag {}
```

By default (when the `@Target` meta-annotation is not present on the annotation type declaration), the annotation can be applied in any declaration context *except type parameter declarations*. In addition, the annotation is *not permitted in any type contexts*.

Click here to view code image

```
@interface Tag {}               // No target specified.
_____________________________________________________________________________
@Tag class Box<@Tag E> {        // Class declaration: OK.
                                // Type parameter: Compile-time error!
  <@Tag T> void doIt(T t) {}    // Type parameter: Compile-time error!
  void doTask() throws @Tag Exception {}     // Type context: Compile-time error!
  int iTod(double d) { return (@Tag int) d; }// Type context: Compile-time error!
  @Tag boolean flag;                         // Field declaration. OK.
}
```

An empty `@Target(value = {})` annotation is allowed, shown at (1) below, but it does not allow the annotation to be directly applied to any program element, as shown at (3) below. However, this annotation can still be used as the type of an element in other annotation type declarations, as shown at (2), which can be applied normally, as shown at (4).

Click here to view code image

```
@Target({})                                 // (1) Empty target.
@interface Exclusive {}

@interface ExtraExclusive {
  Exclusive value() default @Exclusive;     // (2) Annotation as element type.
}
_____________________________________________________________________________-
@Exclusive class Titanium {}                // (3) Compile-time error!
@ExtraExclusive class Krypton {}            // (4) OK.
```

The annotation type declaration below shows that the `MusicMeta` type annotation is applicable to any type (class, interface, and enum), method, or field.

Click here to view code image

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

@Retention(RetentionPolicy.RUNTIME)
@Target(value = {ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
public @interface MusicMeta {}
```

The `@MusicMeta` annotation is applied below on a class at (1), on a field at (2), and on a method at (3), as permitted by the annotation type declaration shown above. Not surprisingly, applying it on a local variable at (4) results in a compile-time error.

Click here to view code image

```
@MusicMeta public class Composition {         // (1) On class
  @MusicMeta private String description;      // (2) On field
  @MusicMeta public void play() {             // (3) On method
//  @MusicMeta int volume;                    // (4) Compile-time error!
  }
}
```

The `@Inherited` Meta-Annotation

By default, an annotation applied on a class is not inherited by its subclasses. However, this can be changed by applying the meta-annotation `@Inherited` on the declaration of the annotation type. Such an annotation will automatically be inherited by subclasses when this annotation is present in their superclass. It has no effect if the annotation is applied to program elements other than classes. Note also that a class cannot inherit annotations from interfaces it implements, nor can a subinterface inherit annotations from superinterfaces it extends, regardless of the fact that these annotations are marked with the meta-annotation `@Inherited`.

The meta-annotation `@Inherited` is of type `java.lang.annotation.Inherited`, which is a *marker meta-annotation type*.

Click here to view code image

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

@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface Playable {}
_______________________________________
@MusicMeta
@Playable
public class Composition {}
_______________________________________
public class Song extends Composition {}
```

In the code above, the annotations `@MusicMeta` and `@Playable` are applied to the class `Composition`, but only the annotation `@Playable` is inherited by the subclass `Song` because only the `@Playable` annotation is applied to the meta-annotation `@Inherited` in its type declaration. This means that the `Composition` class has two annotations, but the subclass `Song` has only one, namely the inherited annotation `@Playable`.

The `@Documented` Meta-Annotation

By default, annotations are not processed by the `javadoc` tool when generating API documentation from the source code, unless the declaration of the annotation type is marked with the

meta-annotation `@Documented`. This meta-annotation is of type `java.lang.annotation.Documented` which is a *marker meta-annotation type*.

Click here to view code image

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented                              // (1)
public @interface Pending {}
_____________________________________
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Verified {}          // (2)
_____________________________________
@Documented
@Verified
public class Gizmo {}                    // (3)
```

In the code above, the declaration of the annotation type `Pending` is marked with the meta-annotation `Documented` at (1), but this is not the case for the declaration of the annotation type `Verified` at (2). Both annotations are applied to the class `Gizmo` at (3). As a result, the generated Javadoc documentation for the class `Gizmo` will only show the presence of the `@Pending` annotation, but not the `@Verified` annotation. Thus the `@Documented` annotation gives annotation developers full control over the process of including annotations in the generated API documentation.

For another example of using the meta-annotation `@Documented`, see the discussion about the `@Deprecated` annotation (**Example 25.1**, **p. 1580**).

The `@Repeatable` Meta-Annotation

There are situations where we would like to apply the same annotation multiple times on the same program element. This makes more sense if the annotation is not a marker annotation and we want to apply the annotation with different values.

The declaration of the `Choice` annotation type below declares two annotation type elements that specify the color and the size.

Click here to view code image

```
import java.lang.annotation.*;
enum Size {S, M, L, XL}
@Retention(RetentionPolicy.RUNTIME)
@interface Choice {
  String color();
  Size size() default Size.L;
}
```

We wish to apply the `@Choice` annotation multiple times to the class `Item` to indicate different combinations of color and size. However, the code below will not compile.

Click here to view code image

```
@Choice(color="Green", size=Size.S)                    // Compile-time error!
@Choice(color="Yellow", size=Size.XL)                  // Compile-time error!
@Choice(color="Red", size=Size.M)                      // Compile-time error!
@Choice(color="White")                                 // Compile-time error!
class Item {}
```

The compiler complains that the `@Choice` annotation is not *repeatable*. The declaration of the *Choice* annotation type must be declared with the meta-annotation `@Repeatable`, which is a *single-element meta-annotation* having the following declaration:

Click here to view code image

```
@Documented
@Retention(RUNTIME)
@Target(ANNOTATION_TYPE)
public @interface Repeatable {
  Class<? extends Annotation> value();  // Indicates the containing annot type
}
```

In order to apply the `@Repeatable` meta-annotation on an annotation type, the value of the *containing annotation type* must be supplied—that is, the `Class` object representing the containing annotation type in its single-element type declaration must be specified. Keep in mind that all annotation types implement the `Annotation` interface.

We can augment the declaration of the `Choice` annotation with the meta-annotation `@Repeatable` that specifies the containing annotation type `Choices` (which has not been declared as yet).

Click here to view code image

```
import java.lang.annotation.*;
enum Size {S, M, L, XL}
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Choices.class)    // @Repeatable specifies the container
                              // annotation type Choices.
@interface Choice {
  String color();
  Size size() default Size.L;
}
```

The *containing annotation type* (`Choices`) of a *repeatable annotation type* (`Choice`) must declare a `value()` method whose return type is *an array of the repeatable annotation type* (`Choice[]`).

Click here to view code image

```
@Retention(RetentionPolicy.RUNTIME)
@interface Choices {
  Choice[] value();             // Return value is an array of Choice.
}
```

The upshot of the setup described above is that the multiple applications of a *repeatable annotation* (`@Choice`) will be stored in the *array of the repeatable annotation type* (`Choice[]`) declared in the *containing annotation type* (`@Choices`).

The multiple applications of the `@Choice` annotation on the class `Item` above will now compile. We can call the method `AnnotationPrinter.printAllAnnotations()` in **Example 25.5**, **p. 1592**, with the `Class<Item>` object `Item.class` to print the annotations on the class `Item`. (*Output shown below has been edited to fit on the page.*)

Click here to view code image

```
Annotations for 'class Item':
  @Choices(value={@Choice(size=S, color="Green"),
                  @Choice(size=XL, color="Yellow"),
                  @Choice(size=M, color="Red"),
```

```
                    @Choice(size=L, color="White")})
...
```

A *Containing Annotation Type* (*CAT*) for a *Repeatable Annotation Type* (*RAT*) must satisfy the following conditions:

- *RAT* must specify *CAT* `.class` as the value of the single element of its meta-annotation `@Repeatable`.
- *CAT* must declare a `value()` annotation type element whose type is *RAT* `[]`.
- Any additional annotation type elements declared in *CAT* must have a default value.

Click here to view code image

```
@Retention(RetentionPolicy.RUNTIME)
@interface Choices {
  Choice[] value();                  // Return value is an array of Choice.
  double minPrice() default 1.00;    // Must specify a default value.
}
```

- The retention policy for *CAT* is at least as long as the retention policy for *RAT*. For example, if *RAT* has retention policy `RUNTIME`, the retention policy for *CAT* cannot be `CLASS` or `SOURCE`. However, if *RAT* has retention policy `CLASS`, the retention policy for *CAT* can be `CLASS` or `RUNTIME`.
- *RAT* is applicable to at least the same kind of program elements as *CAT*—that is, the list of targets to which *RAT* can be applied cannot be shorter than the list of targets to which *CAT* can be applied. For example, if *CAT* is applicable to the targets `METHOD` and `FIELD`, then *RAT* must be applicable to at least these targets.
- If *RAT* has the meta-annotation `@Documented`, then so must *CAT*.
- If *RAT* has the meta-annotation `@Inherited`, then so must *CAT*.

In addition, it is a compile-time error if the meta-annotation `@Repeatable` in the repeatable annotation type does not specify a containing annotation type for the repeatable annotation type.

Click here to view code image

```
@Repeatable(Choices.class)        // Compile-time error since invalid return type
                                  // at (1).
@interface Choice {
  String color();

  Size size() default Size.L;
}

@Retention(RetentionPolicy.RUNTIME)
@interface Choices {
  String[] value();                // (1) Invalid return type.
}
```

25.5 Selected Standard Annotations

The Java SE Platform API provides several predefined annotation types in the `java.lang` package that can readily be used in the source code. Selected predefined annotations are discussed in this section, but they should already be familiar as they have been previously used in this book. The reader is encouraged to consult the Java SE Platform API documentation for further details.

It is important to note that the standard annotations presented here are all optional, but programmers are encouraged to use them as they improve the quality of the code. If used, the compiler can aid in verifying certain aspects of the code at compile time.

The `@Override` Annotation

The marker annotation `@Override` can be used to indicate that *the annotated method in a subtype must override an inherited method from a supertype*. If that is not the case, the compiler issues an error.

Click here to view code image

```
public class A {
  public void doThings() {}
}
public class B extends A {
  @Override
  public void dothings() {}         // Wrong method name.
}
```

Without the `@Override` annotation, the class `B` above will compile successfully even though the method `dothings()` does not really override the method `doThings()` in the superclass `A`. With the `@Override` annotation, the compiler will flag an error that the method does not fulfill the criteria for overriding methods—in this case, the method names do not match.

Basically, the `@Override` annotation when applied to a method helps to catch any errors in an attempt to override some method in its supertype at compile time, ensuring that the overriding method satisfies the criteria for method overriding (§5.1, p. 196).

A common mistake is to inadvertently overload a method from the supertype when the intention is to override it. A classic example is the `equals()` method from the `Object` class that has the header:

Click here to view code image

```
public boolean equals(Object obj)
```

Instead, the method declaration at (1) is declared in the `Gizmo` class below, leading to subtle bugs in the code:

Click here to view code image

```
class Gizmo {
  public boolean equals(Gizmo obj) { // (1) Overloaded, as parameter doesn't match.
    // ...
  }
}
```

Best practices advocate that the `@Override` annotation is always used when overriding a method.

The API of the `Override` annotation type from the `java.lang` package is shown below:

Click here to view code image

```
@Target(ElementType=METHOD)
@Retention(RetentionPolicy=SOURCE)
public @interface Override
```

Note that the `Override` annotation type can only be applied to a method. Its retention policy is `SOURCE`, meaning it is discarded by the compiler once the code is validated—in other words, it is not recorded in the class file and therefore not available at runtime.

The `@FunctionalInterface` Annotation

The marker annotation `@FunctionalInterface` is designed to ensure that a given interface is indeed functional—that is, it contains exactly one abstract method that is either specified or inherited by the interface; otherwise, it will report an *error*.

Click here to view code image

```
@FunctionalInterface
public interface FIX {       // Compiles without errors.One abstract method.
  void doThings();
}

@FunctionalInterface
public interface FIXIt extends FIX {  // Compiles without errors. Abstract method
}                                     // inherited.

@FunctionalInterface
public interface FIY {       // Compile-time error! More than one abstract method.
  void doThings();
  void doOtherThings();
}

@FunctionalInterface
public interface FIZ {       // Compile-time error! No abstract method.
}
```

Without the `@FunctionalInterface` annotation, the interfaces above will compile successfully as they are all valid interface declarations. However, the presence of the `@FunctionalInterface` annotation will ensure that the interface has exactly one abstract method. For more details on declaring functional interfaces, see §13.1, p. 675.

It is not mandatory that all interfaces that have exactly one abstract method should be marked with the `@FunctionalInterface` annotation. For example, the `Comparable<E>` interface qualifies as a functional interface but is not marked with the `@Functional-Interface` annotation in the `java.lang` package API. Typically, only those interfaces whose implementation is provided by lambda expressions or method references are marked with this annotation. However, whether marked with this annotation or not, the compiler will treat interfaces with exactly one abstract method as functional interfaces.

The API of the `FunctionalInterface` annotation type from the `java.lang` package is shown below:

Click here to view code image

```
@Documented
@Retention(RUNTIME)
@Target(TYPE)
public @interface FunctionalInterface
```

Note that the annotation `@FunctionalInterface` can only be applied to a target that is a type declaration (`ElementType.TYPE`)—in particular, *only to an interface declaration*; otherwise, the compiler will flag an error. Its retention policy is `RUNTIME` , and therefore it is recorded in the class file and available at runtime. The `javadoc` tool will include the annotation in the documentation it generates, as the annotation has the meta-annotation `@Documented` .

The `@Deprecated` Annotation

The annotation `@Deprecated` is designed to discourage the use of certain declarations that were previously allowed. The deprecated code can be declarations of methods, constructors,

fields, local variables, packages, modules, parameters, classes, interfaces, and enums, to which this annotation is applied.

It is typically used to indicate that certain code is considered legacy and should no longer be used for various reasons: API design changes, deficiencies in legacy implementations, availability of a better alternative approach that could be more stable, have better performance, and be less likely to cause errors or unwanted side effects. The purpose of the `@Deprecated` annotation is to caution developers that certain code is obsolete and is considered to be removed from future versions. However, the annotation does not prevent such obsolete code from being used in a program. Instead, the compiler can produce a deprecation warning when compiling code that utilizes declarations that have been marked as deprecated.

Although strictly not required, developers are strongly advised to document the reason for deprecating a program element using the Javadoc annotation `@deprecated` in a Javadoc comment. Usually such documentation should describe a reason why certain code has been deprecated and suggest an alternative replacement API, as well as point out any differences in the way this suggested API works or is supposed to be used.

There are two annotation type elements defined by the `Deprecated` annotation type:

Click here to view code image

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

The value of the `since()` element in the `Deprecated` annotation type is used to indicate the Java version when deprecation first occurred. The `boolean` value of the `for-Removal()` element, when set to `true`, indicates the intention to remove the code marked with this annotation in future versions—called *terminally deprecated*. If the value is `false`, the intent is to discourage the use of the deprecated program element, but it does not indicate an imminent intent to remove it—called *ordinarily deprecated*. As both annotation type elements are defined with default values, they can be omitted in the annotation, but it is strongly advised to provide this information for deprecated code.

Some IDEs show the name of the deprecated program element by a strikethrough, as shown in **Example 25.1**. The compiler can provide more details of deprecated code usage if the `-deprecation` or `-Xlint:deprecation` option is used to compile the code, as seen from the compiler output in **Example 25.1**. The warnings relate to the usage of the deprecated method `initAuthentication()` and the deprecated `static` field `SIGN_IN_ATTEMPTS`, respectively.

Example 25.1 *Using the* `@Deprecated` *Annotation*

Click here to view code image

```
class LoginHandler {
  /**
   * The field SIGN_IN_ATTEMPTS has now been deprecated,
   * and replaced by more appropriate name.
   * @deprecated Use ACCOUNT_LOCKOUT_THRESHOLD instead.
   */
  @Deprecated(since = "10", forRemoval = false)         // Ordinarily deprecated
  public static final int SIGN_IN_ATTEMPTS = 4;         // static field.
  public static final int ACCOUNT_LOCKOUT_THRESHOLD = 4;

  /**
   * The following method has now been deprecated.
   * @deprecated Use initTwoFactorAuthentication() instead
   */
  @Deprecated(since = "10", forRemoval = true)          // Terminally deprecated
  public void initAuthentication() { /* ...*/ }         // instance method.
```

```
  public void initTwoFactorAuthentication() { /* ... */ }
}

public class UserLogin {
  public static void main(String[] args) {
    LoginHandler lh = new LoginHandler();
    lh.initAuthentication();                          // Removal warning
    System.out.println(LoginHandler.SIGN_IN_ATTEMPTS);  // Deprecation warning
  }
}
```

Compiling the program:

Click here to view code image

```
>javac -deprecation UserLogin.java
UserLogin.java:24: warning: [removal] initAuthentication() in LoginHandler has
been deprecated and marked for removal
    lh.initAuthentication();                          // Removal warning
      ^

UserLogin.java:25: warning: [deprecation] SIGN_IN_ATTEMPTS in LoginHandler has
been deprecated
    System.out.println(LoginHandler.SIGN_IN_ATTEMPTS);  // Deprecation warning
                                   ^
2 warnings
```

The following example of a deprecated method can be found in the `Object` class API:

Click here to view code image

```
@Deprecated(since="9")
protected void finalize() throws Throwable
```

The example above shows that the use of the `finalize()` method is discouraged and appropriate documentation provides reasons for it, as well as suggests alternative approaches to implementing its functionality. Starting from Java 9, the compiler produces *a deprecation warning* when compiling classes that invoke or override this method.

Other examples to note are the constructors of the wrapper classes which are now deprecated (§8.3, p. 429).

The API of the `Deprecated` annotation type from the `java.lang` package is shown below:

Click here to view code image

```
@Documented
@Retention(RUNTIME)
@Target({CONSTRUCTOR,FIELD,LOCAL_VARIABLE,METHOD,PACKAGE,MODULE,PARAMETER,TYPE})
public @interface Deprecated
```

The primary use of the `@Deprecated` annotation is to mark code that is deprecated so that its use is discouraged. Its retention policy is `RUNTIME` so that it is retained by the compiler, making it possible to dynamically discover whether some code is deprecated through the use of the Reflection API. The contexts in which it can be used are practically all declarations.

As the declaration of the `Deprecated` annotation type specifies the meta-annotation `@Documented`, its use will also be documented by the `javadoc` tool as mentioned earlier. It is recommended to use this annotation in conjunction with the Javadoc annotation `@deprecated` in a Javadoc comment. A screenshot showing partial documentation generated for the deprecated method `initAuthentication()` of the `Login-Handler` class is shown in **Figure 25.1**.

Figure 25.1 *API Documentation of a Deprecated Method*

The `@SuppressWarnings` Annotation

The annotation `@SuppressWarnings` can be used to suppress different kinds of warnings from the compiler in code that would otherwise result in these warnings being issued (§11.13, p. 623).

The `SuppressWarnings` annotation type is an *array-valued single-element annotation type*—that is, it defines a `value()` element of type `String[]`. It is used to indicate which kinds of warnings should be suppressed. Typical kinds of warnings that can be specified for the `value` array are `"unchecked"` and `"deprecation"`. *Unchecked warnings* caution about mixing legacy and generic code that might result in heap pollution (§11.13, p. 630). *Deprecation warnings* result from usage of deprecated code (p. 1580). Different Java compilers may provide different kinds of warnings that can be suppressed.

The `@SuppressWarnings` annotation can be applied at different nested levels of a program construct—for example, when it is applied to a class, it will suppress warnings of a specific kind for the entire class. It is advisable to apply the `@SuppressWarnings` annotation at the deepest nested level possible, such as a specific method, or a variable that may be the cause of the compiler warnings that a programmer wishes to suppress. Specifying the same kind of warning multiple times is permissible, but only the first occurrence of the name is applied and any other occurrences of this name are ignored.

In the code below, suppressing unchecked warnings in the method is redundant, as that is already the case since its class also suppresses unchecked warnings. However, deprecation warnings are only suppressed for the annotated method in the example below. Note also that the compiler will only suppress unchecked warnings relating to the declaration of the method, but it will *not* suppress any unchecked warnings at the call sites for this method, in contrast to the `@SafeVarargs` annotation (p. 1585).

Click here to view code image

```
@SuppressWarnings("unchecked")
public class PhoneCenter {

  @SuppressWarnings({"unchecked", "deprecation"})
  public void callLandline() {}

}
```

Example 25.2 illustrates suppressing warnings with the `@SuppressWarnings` annotation. The intention is to suppress both unchecked and deprecation warnings. To show the potential problems in the code, it is first complied without the `@Suppress-Warnings` annotation using

appropriate compiler options. The compiler output shows two unchecked warnings and two deprecation warnings in the `undisciplined-Method()` and the overridden `finalize()` method, respectively. With these issues in the code, it is hardly recommended to suppress the warnings using the `@Suppress-Warnings` annotation; doing so will suppress them, but it will not solve the potential problems in the code.

Example 25.2 *Using the* `@SuppressWarnings` *Annotation*

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;

@SuppressWarnings(value={"unchecked", "deprecation"})  // (1)
public class ATSuppressWarnings {
  /** Mixing legacy code and generic code. */
  public void undisciplinedMethod() {
    List wordList1 = new ArrayList<String>();  // Assigning parameterized type
                                               // to raw type.
    List<String> wordList2 = wordList1;        // (2) Unchecked conversion
    wordList1.add("911");                      // (3) Unchecked call
    wordList2.add("119");                      // OK
  }

  /** Overriding and using a deprecated method. */
  @Override
  public void finalize() throws Throwable { // (4) Overriding a deprecated method.
    super.finalize();                       // (5) Usage of deprecated method.
  }
}
```

Compiling without the `@SuppressWarnings` annotation at (1) is shown below:

Click here to view code image

```
>javac -Xlint:unchecked -deprecation ATSuppressWarnings.java
ATSuppressWarnings.java:9: warning: [unchecked] unchecked conversion
    List<String> wordList2 = wordList1;        // (2) Unchecked conversion
                             ^
  required: List<String>
  found:    List
ATSuppressWarnings.java:10: warning: [unchecked] unchecked call to add(E) as a
member of the raw type List
    wordList1.add("911");                      // (3) Unchecked call
                 ^
  where E is a type-variable:
    E extends Object declared in interface List
ATSuppressWarnings.java:16: warning: [deprecation] finalize() in Object has been
deprecated
  public void finalize() throws Throwable { // (4) Overriding a deprecated method.
              ^
ATSuppressWarnings.java:17: warning: [deprecation] finalize() in Object has been
deprecated
    super.finalize();                       // (5) Usage of deprecated method.
          ^
4 warnings
```

Suppressing warnings is generally considered to be potentially dangerous, as this might hide potential problems within the code. For example, suppressing deprecation warnings may result in a programmer not noticing that deprecated code is in use and may lead to the program not compiling in future versions of Java because the deprecated code has been removed. Regardless, the `@SuppressWarnings` annotation should not be used unless it has been manually verified that it is safe to do so. Any warnings not suppressed must always be looked into.

The API of the `SuppressWarnings` annotation type from the `java.lang` package is shown below:

Click here to view code image

```
@Target({TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,LOCAL_VARIABLE,MODULE})
@Retention(SOURCE)
public @interface SuppressWarnings
```

The annotation is primarily used for suppressing various warnings that the compiler would otherwise issue. Note that the annotation `@SuppressWarnings` can be applied to declarations of class, interface, enum type, fields, methods, parameters, constructors, local variables, and modules. Its retention policy is `SOURCE` and thus it is discarded by the compiler once the code is validated—in other words, it is not recorded in the class file and therefore not available at runtime.

The `@SafeVarargs` Annotation

The marker annotation `@SafeVarargs` is used to instruct the compiler to suppress unchecked warnings that would otherwise be issued if the declaration or the invocation of a variable arity method or constructor has a variable arity parameter of a non-reifiable element type, as this can lead to heap pollution (§11.13, p. 630).

It is entirely the responsibility of the programmer to ensure that the variable arity method or constructor is well formed to prevent *heap pollution* before using the `@SafeVarargs` annotation to suppress the warnings. Note that the `@SafeVarargs` annotation suppresses any unchecked warnings both for the declaration and all call sites of the variable arity method or constructor.

In order to prevent heap pollution occurring in a subclass, a variable arity method cannot be overridden. The compiler will only allow this annotation on a method that is either `static`, `private`, or `final`. Constructors are not a problem in this regard as they cannot be overridden.

Example 25.3 below illustrates using the `@SafeVarargs` annotation. It is compiled with and without the `@SafeVarargs` annotation at (1). The output without the @`Safe-Varargs` annotation shows that two warnings are generated: the first one where the method `printList()` is declared at (2) and the second one at the call site for this method at (3). Since the variable arity method `printList()` is well formed, there is no risk in using the annotation.

Example 25.3 *Using the* `@SafeVarargs` *Annotation*

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;
public class SafeVarargsTest{
  @SafeVarargs                                          // (1)
  private static void printList(List<String>... toys) { // (2) List<String>[]
    for (List<String> toy : toys) {
      System.out.println(toy);
    }
  }

  public static void main(String[] args) {
    List<String> tList = new ArrayList<String>();
    tList.add("vaporizer"); tList.add("slime gun");
    printList(tList);                                   // (3) new List<String>[]
  }
}
```

Without `@SafeVarargs` at (1):

Click here to view code image

```
>javac SafeVarargsTest.java
Note: SafeVarargsTest.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

>javac -Xlint:unchecked SafeVarargsTest.java
SafeVarargsTest.java:5: warning: [unchecked] Possible heap pollution from parame-
terized vararg type List<String>
  private static void printList(List<String>... toys) { // (2) List<String>[]
                                                 ^

SafeVarargsTest.java:14: warning: [unchecked] unchecked generic array creation for
varargs parameter of type List<String>[]
    printList(tList);                                 // (3) new List<String>[]
              ^

2 warnings
```

With `@SafeVarargs`:

Click here to view code image

```
>javac SafeVarargsTest.java
>java SafeVarargsTest
[vaporizer, slime gun]
```

The API of the `SafeVarargs` annotation type from the `java.lang` package is shown below:

```
@Documented
@Retention(RUNTIME)
@Target({CONSTRUCTOR,METHOD})
public @interface SafeVarargs
```

The primary use of the `@SafeVarargs` marker annotation is to suppress unchecked warnings alerting to potential heap pollution that can result from a variable arity parameter of a non-reifiable element type in a variable arity method or constructor.

The `@SafeVarargs` marker annotation has the `RUNTIME` retention policy and is thus not discarded by the compiler, making it possible to dynamically discover it through the use of the Reflection API. It can be used to annotate the declaration of a constructor or a method—the compiler ensures that the method is either `static`, `private`, or `final`. Its use will also be documented by the `javadoc` tool.

Table 25.3 provides a summary of selected standard annotations.

Table 25.3 *Summary of Selected Standard Annotations*

Standard annotation	Kind of annotation	Retention policy	Element type values of target
`@Override`	*Marker annotation*	`SOURCE`	`METHOD` (*which must be an instance or abstract metho*

Standard annotation	Kind of annotation	Retention policy	Element type values of target
`@FunctionalInterface`	*Marker annotation*	`RUNTIME`	`TYPE` (*which must be an interface*)

`@Deprecated`	*Two elements*:
	`String since() default "";`
	`boolean forRemoval() default false;`
		`RUNTIME`	`CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, MODULE, PARAMETER, TYPE`

`@SuppressWarnings`	*Single-element annotation*: `String[] value();`	`SOURCE`	`CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, MODULE, PARAMETER, TYPE`
`@SafeVarargs`	*Marker annotation*	`RUNTIME`	`CONSTRUCTOR, METHOD (which must be static, private, or f`

Other Annotations

Many other annotations used in various contexts are defined in the language. Additional annotations may be provided with various extended APIs that are not part of the core Java language, but nevertheless are often used in practical programming, especially in server-side Java applications deployed in the Java Enterprise Edition or Micro Profile environments. Examples include the Java Persistence (JPA), Container Dependency Injection (CDI), and Bean Validation APIs. These APIs go beyond the Java SE Edition and are therefore not on the Java SE certification exam. However, they are essential for developing real-world Java applications.

25.6 Processing Annotations

It is possible to discover annotations on program elements at runtime using the Reflection API found in the `java.lang.reflect` package. As this topic is beyond the scope of this book, we provide a brief introduction to the Reflection API—in particular, how to discover annotations on a class and its members.

Figure 25.2 shows a few selected classes and interfaces from the Reflection API. At runtime, an instance of the class `Class<T>` represents the type `T` in a Java program. Classes, interfaces, enums, annotations, and primitive types are all represented by `Class` objects at runtime. For example, the class `NuclearPlant` is represented by a `Class` object which can be referenced by `NuclearPlant.class` and whose type is `Class<NuclearPlant>` at compile time to ensure type-safety before type erasure removes the type parameter.

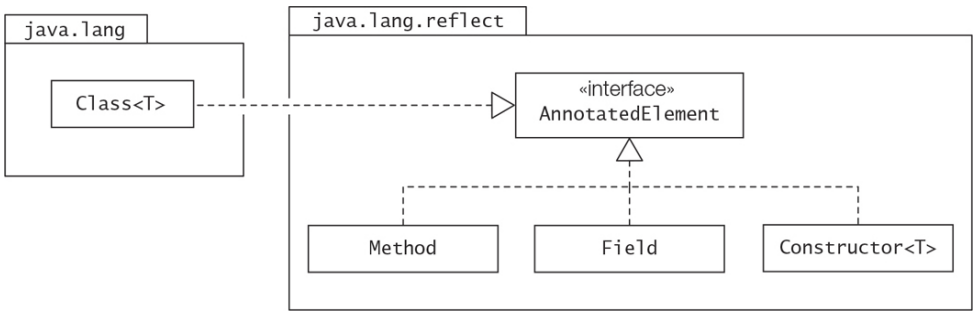

Figure 25.2 *Selected Types from the Reflection API*

A `Class<T>` object representing a class can be queried for members that are declared in the class:

Click here to view code image

```
Constructor<?>[] cons = classobj.getDeclaredConstructors();
Method[] methods      = classobj.getDeclaredMethods();
Field[] fields        = classobj.getDeclaredFields();
```

The classes `Class<T>`, `Method`, `Field`, and `Constructor<T>` implement the `java.lang.reflect.AnnotatedElement` interface, as shown in **Figure 25.2**. The method `getDeclaredAnnotations()` of the `AnnotatedElement` interface can be used to obtain the annotations applied to an annotated element. All annotation types implicitly implement the `java.lang.annotation.Annotation` interface. The method `getDeclared-Annotations()` of the `Annotation` interface returns an array of `Annotation` containing the annotations applied to an annotated element. Using this method, we can obtain the annotations on a class or any member declared in the class.

Click here to view code image

```
Annotation[] annotations = annotatedElement.getDeclaredAnnotations();
```

If we are interested in a particular annotation on an annotated element, we can use the `getDeclaredAnnotationsByType()` method of the `AnnotatedElement` interface. The code below

returns all `@TaskInfo` annotations applied on the `NuclearPlant` class. Since the `@TaskInfo` annotation is not repeatable, the array returned will have at most one `TaskInfo` object, which can be queried by calling the methods declared in the `TaskInfo` type declaration.

Click here to view code image

```
TaskInfo[] tias = NuclearPlant.class.getDeclaredAnnotationsByType(TaskInfo.class);
```

Of course, in order for an annotation to be discoverable at runtime, its retention policy must be `RUNTIME`.

In **Example 25.4**, the class `NuclearPlant` and its members are marked with annotations which have previously been declared in this chapter: `@Pending` and `@TaskInfo`. In addition, the standard annotations `@Deprecated` and `@Override` are also applied to some methods.

Example 25.4 *Processing Annotations*

Click here to view code image

```
@Pending
@TaskInfo(
    priority   = TaskInfo.TaskPriority.HIGH,
    taskDesc   = "Class for running a nuclear reactor.",
    assignedTo = {"Tom", "Dick", "Harriet"}
    )
public class NuclearPlant {

  @Pending
  public NuclearPlant() {}

  @Deprecated(forRemoval = true, since = "8")
  public boolean outOfProduction;

  @Deprecated(since = "10")
  public void notInUse() {}

  @Pending
  @TaskInfo(
      taskDesc   = "Procedure for nuclear reactor shutdown",
      assignedTo = {"Tom", "Harriet"}
      )
  public void shutDownNuclearReactor() {}

  @TaskInfo(
      priority   = TaskInfo.TaskPriority.LOW,
      taskDesc   = "Exchange nuclear rods",
      assignedTo = {"Tom", "Dick"}
      )
  public void changeNuclearRods() {}

  @TaskInfo(
      priority   = TaskInfo.TaskPriority.LOW,
      taskDesc   = "Adjust nuclear fuel",
      assignedTo = {"Harriet"}
      )
  public void adjustNuclearFuel() {}

  @TaskInfo(
      taskDesc   = "Start nuclear reactor",
      assignedTo = "Dick"
      )
  public void startNuclearReactor() {}

  @Pending
  @Override
```

```
  public String toString() {
    return "TBD";
  }
}
```

Click here to view code image

```
import static java.lang.System.out;

import java.lang.reflect.AnnotatedElement;
import java.util.Arrays;
import java.util.stream.Stream;

public class TaskInfoAnnotationProcessor {

  public static void printTaskInfoAnnotation(AnnotatedElement... elements) {// (1)
    Stream.of(elements)                                                     // (2)
      .filter(ae -> ae.isAnnotationPresent(TaskInfo.class))                 // (3)
      .peek(ae -> out.printf("%s annotation for '%s':%n",                   // (4)
          TaskInfo.class.getName(), ae))
      .flatMap(ae -> Stream.of(
                      ae.getDeclaredAnnotationsByType(TaskInfo.class)))    // (5)
      .forEach(a -> {                                                       // (6)
        out.printf("  Task description: %s%n", a.taskDesc());
        out.printf("  Priority: %s%n", a.priority());
        out.printf("  Assigned to: %s%n", Arrays.toString(a.assignedTo()));
      });
  }

  public static void main(String[] args) {
    Class<?> classobj = NuclearPlant.class;                                 // (7)
    printTaskInfoAnnotation(classobj);                                      // (8)
    printTaskInfoAnnotation(classobj.getDeclaredMethods());                 // (9)
  }
}
```

Output from the program:

Click here to view code image

```
TaskInfo annotation for 'class NuclearPlant':
  Task description: Class for running a nuclear reactor.
  Priority: HIGH
  Assigned to: [Tom, Dick, Harriet]
TaskInfo annotation for 'public void NuclearPlant.shutDownNuclearReactor()':
  Task description: Procedure for nuclear reactor shutdown
  Priority: NORMAL
  Assigned to: [Tom, Harriet]
TaskInfo annotation for 'public void NuclearPlant.changeNuclearRods()':
  Task description: Exchange nuclear rods
  Priority: LOW
  Assigned to: [Tom, Dick]
TaskInfo annotation for 'public void NuclearPlant.adjustNuclearFuel()':
  Task description: Adjust nuclear fuel
  Priority: LOW
  Assigned to: [Harriet]
TaskInfo annotation for 'public void NuclearPlant.startNuclearReactor()':
  Task description: Start nuclear reactor
  Priority: NORMAL
  Assigned to: [Dick]
```

In **Example 25.4**, the method `printTaskInfoAnnotation()` at (1) in the `TaskInfoAnnotationProcessor` class can be used to specifically discover `TaskInfo` annotations on program elements. Its parameter is an array of `AnnotatedElement`—that is, it con-

tains program elements that are presumably marked with annotations. A stream of `AnnotatedElement` to process each annotated element is created at (2) from the array of `AnnotatedElement` that is passed as an argument to the method. This stream is filtered at (3) to discard any annotated element that is not marked with the `TaskInfo` annotation. A header is printed at (4) to indicate which annotated element is being processed for `TaskInfo` annotation. At (5) the method `getDeclaredAnnotationsByType()` returns an array of `TaskInfo`, containing any `TaskInfo` annotations applied to the current annotated element. This array of `TaskInfo` is flattened to create a stream of `TaskInfo` annotations. The element values of a `TaskInfo` annotation are extracted at (6) by calling the methods specified in the annotation type elements declared in the `TaskInfo` annotation type.

The method `printTaskInfoAnnotation()` is called at (8) and (9) to process `TaskInfo` annotations for the class `NuclearPlant` and its methods, respectively. The program output shows the result of running this annotation processor on the `NuclearPlant` class.

In **Example 25.5**, the class `AnnotationPrinter` implements a more general annotation processor that prints all annotations applied on a class and its members. The numbers below refer to the numbered lines in the code in **Example 25.5**.

(1) The `printAllAnnotations()` method is passed the `Class` object of the class whose annotations should be printed.
(2) through (5): The method `printAnnotatedElements()` is successively called to print any annotations on the class, its constructors, its methods, and its fields.
(6) The method `printAnnotatedElements()` accepts elements that implement the `AnnotatedElement` interface. Recall that this is the case for the `Class<T>`, `Constructor<T>`, `Method`, and `Field` classes (**Figure 25.2**).
(7) A stream of `AnnotatedElement` is created.
(8) The name of the `AnnotatedElement` is printed.
(9) The method `getDeclaredAnnotations()` returns an array with the annotations applied to an `AnnotatedElement`, which is flattened into a stream of `Annotation`.
(10) The text representation of an `Annotation` is printed.

The program output shows the results of running the annotation processor on the `NuclearPlant` class. It is worth running this annotation processor on classes that we have seen in this chapter, keeping in mind that any annotation applied in the code must have `RetentionPolicy.RUNTIME` in order for it to be processed at runtime. For instance, the `@Override` annotation on the `toString()` method of the `NuclearPlant` class is not in the program output, as it has `RetentionPolicy.SOURCE`.

Example 25.5 *Annotation Processor*

Click here to view code image

```
import java.lang.annotation.Annotation;
import java.lang.reflect.AnnotatedElement;
import java.util.stream.Stream;

public class AnnotationPrinter {

  public static void printAllAnnotations(Class<?> classobj) {             // (1)
    printAnnotatedElements(classobj);                                     // (2)
    printAnnotatedElements(classobj.getDeclaredConstructors());           // (3)
    printAnnotatedElements(classobj.getDeclaredMethods());                // (4)
    printAnnotatedElements(classobj.getDeclaredFields());                 // (5)
  }

  public static void printAnnotatedElements(AnnotatedElement... elements) {// (6)
    Stream.of(elements)                                                   // (7)
      .peek(ae -> System.out.printf("Annotations for \'%s\':%n", ae))     // (8)
      .flatMap(ae -> Stream.of(ae.getDeclaredAnnotations()))              // (9)
      .forEach(a -> System.out.println("  " + a));                       // (10)
```

```
  }
}
```

Click here to view code image

```
public class AnnotationClient {
  public static void main(String[] args) {
    AnnotationPrinter.printAllAnnotations(NuclearPlant.class);
  }
}
```

Output from the program (*edited to fit on the page*):

Click here to view code image

```
Annotations for 'class NuclearPlant':
  @Pending()
  @TaskInfo(priority=HIGH, taskDesc="Class for running a nuclear reactor.",
            assignedTo={"Tom", "Dick", "Harriet"})
Annotations for 'public NuclearPlant()':
  @Pending()
Annotations for 'public void NuclearPlant.shutDownNuclearReactor()':
  @Pending()
  @TaskInfo(priority=NORMAL, taskDesc="Procedure for nuclear reactor shutdown",
            assignedTo={"Tom", "Harriet"})
Annotations for 'public void NuclearPlant.notInUse()':
  @java.lang.Deprecated(forRemoval=false, since="10")
Annotations for 'public void NuclearPlant.changeNuclearRods()':
  @TaskInfo(priority=LOW, taskDesc="Exchange nuclear rods",
            assignedTo={"Tom", "Dick"})
Annotations for 'public void NuclearPlant.adjustNuclearFuel()':
  @TaskInfo(priority=LOW, taskDesc="Adjust nuclear fuel", assignedTo={"Harriet"})
Annotations for 'public void NuclearPlant.startNuclearReactor()':
  @TaskInfo(priority=NORMAL, taskDesc="Start nuclear reactor",
            assignedTo={"Dick"})
Annotations for 'public java.lang.String NuclearPlant.toString()':
  @Pending()
Annotations for 'public boolean NuclearPlant.outOfProduction':
  @java.lang.Deprecated(forRemoval=true, since="8")
```

Review Questions

25.1 Which of the following statements are true about annotations? Select the two correct answers.

a. Annotations are compiled as classes.

b. Annotations are used to provide metadata for Java program elements.

c. An annotation cannot be applied to other annotations.

d. An annotation cannot be an element type in another annotation.

25.2 Which of the following statements is true about the value of `ElementType` that can be specified for the meta-annotation `@Target` in an annotation type declaration— that is, the target of an annotation type?

Select the one correct answer.

a. `ElementType.TYPE` means the annotation type cannot be applied to annotations.

b. `ElementType.TYPE_PARAMETER` means the annotation type cannot be applied to generic types.

c. `ElementType.FIELD` means the annotation type can be applied to constants.

d. `ElementType.METHOD` means the annotation type can be applied to constructors.

25.3 Given the following code:

```
@Test3Annotation          // (1)
public class Test3 { }
```

Which of the following annotation types are compatible with the annotation applied at (1)?

Select the two correct answers.

a.

Click here to view code image

```
public @interface Test3Annotation { }
```

b.

Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.*;
@Target(PACKAGE)
public @interface Test3Annotation { }
```

c.

Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.*;
@Target({FIELD,TYPE})
public @interface Test3Annotation { }
```

d.

Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.*;
@Target({PACKAGE,TYPE_USE}) public
@interface Test3Annotation { }
```

e. None of the above

25.4 Given the following code:

Click here to view code image

```
@interface Test4Annotation {
  int[] value() default {1,2,3};
}
```

and

Click here to view code image

```
// (1) INSERT ANNOTATION HERE
public class Test4 { }
```

Which of the following are incorrect ways to apply `@Test4Annotation` at (1)? Select the two correct answers.

a. `@Test4Annotation(value=7)`

b. `@Test4Annotation(value=7,8,9)`

c. `@Test4Annotation({7,8,9})`

d. `@Test4Annotation`

e. `@Test4Annotation()`

f. `@Test4Annotation(8)`

g. `@Test4Annotation(8,9)`

25.5 Given the following code:

Click here to view code image

```
@Test5Annotation
public class Test5<@Test5Annotation(value="a") T> {
  @Test5Annotation({"a","b"})
  private T var;
  public Test5(T var) {
  this.var = var;
  }
}
```

Which of the following annotation types is compatible with the annotation `@Test5Annotation`?
Select the one correct answer.

a.

Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({CONSTRUCTOR, FIELD, TYPE})
@interface Test5Annotation {
  String[] value() default {"x"};
}
```

b.

Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({METHOD, FIELD, TYPE_PARAMETER, TYPE})
@interface Test5Annotation {
```

```
  String[] value() default "x";
}
```

c.

Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({CONSTRUCTOR, FIELD, TYPE_PARAMETER, TYPE})
@interface Test5Annotation {
  String[] values() default {"x"};
}
```

d.

Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({METHOD, FIELD, TYPE_PARAMETER})
@interface Test5Annotation {
  String[] values() default "x";
}
```

25.6 Which of the following statements is true about annotations being reflected in the class documentation generated by the `javadoc` tool?

Select the one correct answer.

a. Annotations applied at the class level are reflected in the class documentation.

b. Annotations applied at the method level are reflected in the class documentation.

c. Annotations applied to public methods are reflected in the class documentation.

d. Annotations are not reflected in the class documentation by default.

25.7 Which of the following annotation types are correctly declared? Select the two correct answers.

a.

Click here to view code image

```
public @interface Location {
  String value() default null;
  int[] coordinates default 1;
}
```

b.

Click here to view code image

```
public @interface Location {
  String value() default "London";
  int[] coordinates() default 1;
}
```

c.

Click here to view code image

```
public @interface Location {
  String value();
  int[] coordinates default {1,1};
}
```

d.

Click here to view code image

```
public @interface Location {
  String value() default null;
  int[] coordinates default 1;
}
```

e.

Click here to view code image

```
public @interface Location {
  String value();
  int[] coordinates() default {1,1};
}
```

25.8 Given the following code:

Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({FIELD, CONSTRUCTOR, TYPE})
public @interface Descriptor {
  String value();
  String[] details() default "";
}
```

Which of the following applications of this annotation type are valid? Select the three correct answers.

a.

Click here to view code image

```
@Descriptor("Music to play")
public interface Playable { }
```

b.

Click here to view code image

```
public enum Style {
  @Descriptor(value="Rock music", details={"listen"})
  ROCK,
  @Descriptor(value="POP music", details="dance")
  POP,
  @Descriptor(value="Jazz music", details={"listen", "dance"})
  JAZZ,
  @Descriptor("Classic music")
```

```
  CLASSIC;
}
```

c.

Click here to view code image

```
public class Music {
   @Descriptor(value="Music to play")
   Music m = new Music();
}
```

d.

Click here to view code image

```
@Descriptor("Music to play",details={"listen","dance"})
public class Music {
  Music m = new Music();
}
```

e.

Click here to view code image

```
public class Player {
  @Descriptor("Music to play")
  public static void main(String[] args) {
    Music m = new Music();
  }
}
```

f.

Click here to view code image

```
public class Player {
  public static void main(String[] args) {
     @Descriptor(value="Music to play",details={"listen","dance"})
     new Music();
  }
}
```

25.9 Given the following code:

Click here to view code image

```
import java.lang.annotation.*;
@Repeatable(Container.class)
public @interface Containee { int value(); }
```

and

Click here to view code image

```
public @interface Container {
  // (1) INSERT CODE HERE
}
```

Which of the following code fragments inserted at (1) will allow the annotation types to compile?

Select the one correct answer.

a.

```
String name();
  Containee[] value();
```

b.

```
String name() default "x";
    Containee[] value();
```

c.

```
String name() default "x";
    Containee[] values();
```

d.

Click here to view code image

```
String name() default "x";
    Containee value() default @Containee;
```

e.

Click here to view code image

```
String name() default "x";
    Containee[] values() default {@Containee(1), @Containee(2)};
```

f.

Click here to view code image

```
String name();
    Containee[] values() default {@Containee(1), @Containee(2)};
```

25.10 Given the following code:

Click here to view code image

```
@Folder("/data")
@Folder(value="/tmp", temp=true)
public class Storage { }
```

Which of the following declarations of the `Folder` and `Folders` annotation types is valid? Select the one correct answer.

a.

Click here to view code image

```
import java.lang.annotation.Repeatable;
@Repeatable(Folders.class)
public @interface Folder {
  String value();
  boolean temp() default false;
}
```

and

Click here to view code image

```
public @interface Folders {
  Folder[] paths();
}
```

b.

Click here to view code image

```
import java.lang.annotation.Repeatable;
@Repeatable(Folders.class)
public @interface Folder {
  String value();
  boolean temp();
}
```

and

Click here to view code image

```
public @interface Folders {
  Folder[] value();
}
```

c.

Click here to view code image

```
import java.lang.annotation.Repeatable;
@Repeatable(Folders.class)
public @interface Folder {
  String value();
  boolean temp() default false;
}
```

and

```
public @interface Folders {
  Folder[] value();
}
```

d.

Click here to view code image

```
import java.lang.annotation.Repeatable;
@Repeatable(Folders.class)
public @interface Folder {
  String path();
```

```
    boolean temp() default true;
  }
```

and

```
  public @interface Folders {
    Folder[] value();
  }
```