Annotations

Annotations

Annotations have a number of uses, among them:

- Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.
- Compile-time and deployment-time processing — Software tools can process annotation information to generate cod
- Runtime processing — Some annotations are available to be examined at runtime.

This section explains where annotations can be used, how to apply annotations, what predefined annotation types are ava

The Format of an Annotation

In its simplest form, an annotation looks like the following:

```
@Entity
```

The at sign character (@) indicates to the compiler that what follows is an annotation. In the following example, the annot

```
@Override
void mySuperMethod() { ... }
```

The annotation can include *elements*, which can be named or unnamed, and there are values for those elements:

```
@Author(
    name = "Benjamin Franklin",
    date = "3/27/2003"
)
class MyClass { ... }
```

or

```
@SuppressWarnings(value = "unchecked")
void myMethod() { ... }
```

If there is just one element named `value`, then the name can be omitted, as in:

```
@SuppressWarnings("unchecked")
void myMethod() { ... }
```

If the annotation has no elements, then the parentheses can be omitted, as shown in the previous `@Override` example.

It is also possible to use multiple annotations on the same declaration:

```
@Author(name = "Jane Doe")
@EBook
```

```
class MyClass { ... }
```

If the annotations have the same type, then this is called a repeating annotation:

```
@Author(name = "Jane Doe")
@Author(name = "John Smith")
class MyClass { ... }
```

Repeating annotations are supported as of the Java SE 8 release. For more information, see the section Repeating Annot

The annotation type can be one of the types that are defined in the `java.lang` or `java.lang.annotation` packages of the

Where Annotations Can Be Used

Annotations can be applied to declarations: declarations of classes, fields, methods, and other program elements. When

As of the Java SE 8 release, annotations can also be applied to the use of types. Here are some examples:

- Class instance creation expression:

```
new @Interned MyObject();
```

- Type cast:

```
myString = (@NonNull String) str;
```

- implements clause:

```
class UnmodifiableList<T> implements
  @Readonly List<@Readonly T> { ... }
```

- Thrown exception declaration:

```
void monitorTemperature() throws
  @Critical TemperatureException { ... }
```

This form of annotation is called a *type annotation*.

Declaring an Annotation Type

Many annotations replace comments in code.

Suppose that a software group traditionally starts the body of every class with comments providing important informatio

```
public class Generation3List extends Generation2List {

   // Author: John Doe
   // Date: 3/17/2002
   // Current revision: 6
```

```
    // Last modified: 4/12/2004
    // By: Jane Doe
    // Reviewers: Alice, Bill, Cindy

    // class code goes here

}
```

To add this same metadata with an annotation, you must first define the annotation type. The syntax for doing this is:

```
@interface ClassPreamble {
   String author();
   String date();
   int currentRevision() default 1;
   String lastModified() default "N/A";
   String lastModifiedBy() default "N/A";
   // Note use of array
   String[] reviewers();
}
```

The annotation type definition looks similar to an interface definition where the keyword interface is preceded by the at si

The body of the previous annotation definition contains annotation type element declarations, which look a lot like metho

After the annotation type is defined, you can use annotations of that type, with the values filled in, like this:

```
@ClassPreamble (
   author = "John Doe",
   date = "3/17/2002",
   currentRevision = 6,
   lastModified = "4/12/2004",
   lastModifiedBy = "Jane Doe",
   // Note array notation
   reviewers = {"Alice", "Bob", "Cindy"}
)
public class Generation3List extends Generation2List {

// class code goes here

}
```

> *Note: To make the information in `@ClassPreamble` appear in Javadoc-generated documentation, you must annotate the*

```
// import this to use @Documented
import java.lang.annotation.*;

@Documented
@interface ClassPreamble {

   // Annotation element definitions

}
```

Predefined Annotation Types

A set of annotation types are predefined in the Java SE API. Some annotation types are used by the Java compiler, and sc

Annotation Types Used by the Java Language

The predefined annotation types defined in java.lang are `@Deprecated`, `@Override`, and `@SuppressWarnings`.

@Deprecated

`@Deprecated` annotation indicates that the marked element is deprecated and should no longer be used. The compiler ge

```
// Javadoc comment follows
/**
 * @deprecated
 * explanation of why it was deprecated
 */
@Deprecated
static void deprecatedMethod() { }
```

Note that, as of Java SE 9, a `forRemoval` attribute has been added to the `@Deprecated` annotation. It indicates whether th

@Override

`@Override` annotation informs the compiler that the element is meant to override an element declared in a superclass. Ov

```
// mark method as a superclass method
// that has been overridden
@Override
int overriddenMethod() { }
```

While it is not required to use this annotation when overriding a method, it helps to prevent errors. If a method marked wit

@SuppressWarnings

`@SuppressWarnings` annotation tells the compiler to suppress specific warnings that it would otherwise generate. In the f

```
// use a deprecated method and tell
// compiler not to generate a warning
@SuppressWarnings("deprecation")
void useDeprecatedMethod() {
    // deprecation warning
    // - suppressed
    objectOne.deprecatedMethod();
}
```

Every compiler warning belongs to a category. The Java Language Specification lists two categories: deprecation and unc

```
@SuppressWarnings({"unchecked", "deprecation"})
```

@SafeVarargs

`@SafeVarargs` annotation, when applied to a method or constructor, asserts that the code does not perform potentially ur

@FunctionalInterface

`@FunctionalInterface` annotation, introduced in Java SE 8, indicates that the type declaration is intended to be a functic

Annotations That Apply to Other Annotations

Annotations that apply to other annotations are called meta-annotations. There are several meta-annotation types define

@Retention

`@Retention` annotation specifies how the marked annotation is stored:

- `RetentionPolicy.SOURCE` – The marked annotation is retained only in the source level and is ignored by the compiler.
- `RetentionPolicy.CLASS` – The marked annotation is retained by the compiler at compile time, but is ignored by the J
- `RetentionPolicy.RUNTIME` – The marked annotation is retained by the JVM so it can be used by the runtime environr

@Documented

`@Documented` annotation indicates that whenever the specified annotation is used those elements should be documented

@Target

`@Target` annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to. A

- `ElementType.ANNOTATION_TYPE` can be applied to an annotation type.
- `ElementType.CONSTRUCTOR` can be applied to a constructor.
- `ElementType.FIELD` can be applied to a field or property.
- `ElementType.LOCAL_VARIABLE` can be applied to a local variable.
- `ElementType.METHOD` can be applied to a method-level annotation.
- `ElementType.MODULE` can be applied to a module declaration.
- `ElementType.PACKAGE` can be applied to a package declaration.
- `ElementType.PARAMETER` can be applied to the parameters of a method.
- `ElementType.RECORD_COMPONENT` can be applied to the component of a record.
- `ElementType.TYPE` can be applied to the declaration of a class, an abtract class, an interface, an annotation interface,
- `ElementType.TYPE_PARAMETER` can be applied on the parameters of a type.
- `ElementType.TYPE_USE` can be applied where a type is used, for instance on the declaration of a field.

@Inherited

`@Inherited` annotation indicates that the annotation type can be inherited from the super class. (This is not true by defau

@Repeatable

`@Repeatable` annotation, introduced in Java SE 8, indicates that the marked annotation can be applied more than once to

Type Annotations and Pluggable Type Systems

Before the Java SE 8 release, annotations could only be applied to declarations. As of the Java SE 8 release, annotations

Type annotations were created to support improved analysis of Java programs way of ensuring stronger type checking. T

For example, you want to ensure that a particular variable in your program is never assigned to null; you want to avoid trig

```
1 | @NonNull String str;
```

When you compile the code, including the `NonNull` module at the command line, the compiler prints a warning if it detects

You can use multiple type-checking modules where each module checks for a different kind of error. In this way, you can l

With the judicious use of type annotations and the presence of pluggable type checkers, you can write code that is strong

In many cases, you do not have to write your own type checking modules. There are third parties who have done the work

Repeating Annotations

There are some situations where you want to apply the same annotation to a declaration or type use. As of the Java SE 8

For example, you are writing code to use a timer service that enables you to run a method at a given time or on a certain s

```
@Schedule(dayOfMonth="last")
@Schedule(dayOfWeek="Fri", hour="23")
public void doPeriodicCleanup() { ... }
```

The previous example applies an annotation to a method. You can repeat an annotation anywhere that you would use a s

```
@Alert(role="Manager")
@Alert(role="Administrator")
public class UnauthorizedAccessException extends SecurityException { ... }
```

For compatibility reasons, repeating annotations are stored in a container annotation that is automatically generated by tl

Step 1: Declare a Repeatable Annotation Type

The annotation type must be marked with the `@Repeatable` meta-annotation. The following example defines a custom `@S`

```

@Repeatable(Schedules.class)
public @interface Schedule {
  String dayOfMonth() default "first";
  String dayOfWeek() default "Mon";
  int hour() default 12;
}
```

The value of the `@Repeatable` meta-annotation, in parentheses, is the type of the container annotation that the Java comp

Applying the same annotation to a declaration without first declaring it to be repeatable results in a compile-time error.

Step 2: Declare the Containing Annotation Type

The containing annotation type must have a `value` element with an array type. The component type of the array type mus

```
public @interface Schedules {
    Schedule[] value();
}
```

Retrieving Annotations

There are several methods available in the Reflection API that can be used to retrieve annotations. The behavior of the me

Design Considerations

When designing an annotation type, you must consider the cardinality of annotations of that type. It is now possible to us

Last update: *September 14, 2021*