Annotations

Annotations, a form of metadata, provide data about a program that is not part of the program itself.

Annotations have no direct effect on the operation of the code they annotate.

Some uses of annotations:

- 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 code, XML files, and so forth.

- Runtime processing — Some annotations are available to be examined at runtime.

This lesson explains where annotations can be used, how to apply annotations, what predefined annotation types are available in the Java Platform, Standard Edition (Java SE API), how type annotations can be used in conjunction with pluggable type systems to write code with stronger type checking, and how to implement repeating annotations.

Annotations Basics

Format of an Annotation

simplest form: `@Entity`

(@) indicates to the compiler that what follows is an annotation.

Example: this annotation's name is `@Override`

```
@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.

Example:

```
@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 { ... }
```

The annotation type can be one of the types that are defined in the java.lang or java.lang.annotation packages of the Java SE API. In the previous examples, Override and SuppressWarnings are predefined Java annotations. It is also possible to define your own annotation type. The Author and Ebook annotations in the previous example are custom annotation types.

Repeating annotations are supported as of the Java SE 8 release.

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 information:

```
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 sign (@) (@ = AT, as in annotation type). Annotation types are a form of interface, which will be covered in a later lesson. For the moment, you do not need to understand interfaces.

The body of the previous annotation definition contains annotation type element declarations, which look a lot like methods. Note that they can define optional default values.

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

Example:

```
@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 @ClassPreamble definition with the @Documented annotation:

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

@Documented
@interface ClassPreamble {

   // Annotation element definitions

}
```

Annotations Basics

Where Annotations are used

Annotations can be applied to declarations: declarations of classes, fields, methods, and other program elements. When used on a declaration, each annotation often appears, by convention, on its own line.

As of the Java SE 8 release, annotations can also be applied to the use of types. Examples:

- Class instance creation expression:

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

- Type cast:

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

- implements clause:

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

- Class instance creation expression:

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

If there is just one element named value, then the name can be omitted.

Example:

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

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

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 some apply to other annotations.

Where Annotations are used

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 generates a warning whenever a program uses a method, class, or field with the @Deprecated annotation. When an element is deprecated, it should also be documented using the Javadoc @deprecated tag, as shown in the following example.

The use of the at sign (@) in both Javadoc comments and in annotations is not coincidental: they are related conceptually.

Note: that the Javadoc tag starts with a lowercase d and the annotation starts with an uppercase D.

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

@Override annotation informs the compiler that the element is meant to override an element declared in a superclass.
Overriding methods will be discussed in Interfaces and Inheritance.

```
// 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 with @Overridefails to correctly override a method in one of its superclasses, the compiler generates an error.

@SuppressWarnings annotation tells the compiler to suppress specific warnings that it would otherwise generate. In the following example, a deprecated method is used, and the compiler usually generates a warning. In this case, however, the annotation causes the warning to be suppressed.

```
// 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 unchecked. The unchecked warning can occur when interfacing with legacy code written before the advent of generics.
To suppress multiple categories of warnings
Syntax:

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

@SafeVarargs annotation, when applied to a method or constructor, asserts that the code does not perform potentially unsafe operations on its varargs parameter. When this annotation type is used, unchecked warnings relating to varargs usage are suppressed.

@FunctionalInterface annotation, introduced in Java SE 8, indicates that the type declaration is intended to be a functional interface, as defined by the Java Language Specification.

Predefined Annotation Types

Annotations that Apply to Other Annotations

Annotations that apply to other annotations are called meta-annotations. There are several meta-annotation types defined in `java.lang.annotation`

@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 Java Virtual Machine (JVM).`
- `RetentionPolicy.RUNTIME – The marked annotation is retained by the JVM so it can be used by the runtime environment.`

@Documented annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool. (By default, annotations are not included in Javadoc.) For more information, see the Javadoc tools page.

@Target annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to. A target annotation specifies one of the following element types as its value:

- 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.PACKAGE can be applied to a package declaration.
- ElementType.PARAMETER can be applied to the parameters of a method.
- ElementType.TYPE can be applied to any element of a class.

@Inherited annotation indicates that the annotation type can be inherited from the super class. (This is not true by default.) When the user queries the annotation type and the class has no annotation for this type, the class' superclass is queried for the annotation type. This annotation applies only to class declarations.

@Repeatable annotation, introduced in Java SE 8, indicates that the marked annotation can be applied more than once to the same declaration or type use.

Predefined Annotation Types

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 can also be applied to any type use. This means that annotations can be used anywhere you use a type. A few examples of where types are used are class instance creation expressions (new), casts, implements clauses, and throws clauses. This form of annotation is called a type annotation and several examples are provided in Annotations Basics.

Type annotations were created to support improved analysis of Java programs way of ensuring stronger type checking. The Java SE 8 release does not provide a type checking framework, but it allows you to write (or download) a type checking framework that is implemented as one or more pluggable modules that are used in conjunction with the Java compiler.

Example: you want to ensure that a particular variable in your program is never assigned to null; you want to avoid triggering a NullPointerException. You can write a custom plug-in to check for this. You would then modify your code to annotate that particular variable, indicating that it is never assigned to null.
The variable declaration might look like this:
@NonNull String str;

When you compile the code, including the NonNull module at the command line, the compiler prints a warning if it detects a potential problem, allowing you to modify the code to avoid the error. After you correct the code to remove all warnings, this particular error will not occur when the program runs.

You can use multiple type-checking modules where each module checks for a different kind of error. In this way, you can build on top of the Java type system, adding specific checks when and where you want them.

With the judicious use of type annotations and the presence of pluggable type checkers, you can write code that is stronger and less prone to error.
In many cases, you do not have to write your own type checking modules. There are third parties who have done the work for you.

Example: you might want to take advantage of the Checker Framework created by the University of Washington.

This framework includes a NonNull module, as well as a regular expression module, and a mutex lock module. For more information, see the Checker Framework.

Repeating Annotations

Type Annotations and Pluggable Type Systems

There are some situations where you want to apply the same annotation to a declaration or type use. As of the Java SE 8 release, repeating annotations enable you to do this.

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 schedule, similar to the UNIX cron service. Now you want to set a timer to run a method, doPeriodicCleanup, on the last day of the month and on every Friday at 11:00 p.m. To set the timer to run, create an @Schedule annotation and apply it twice to the doPeriodicCleanup method.
The first use specifies the last day of the month and the second specifies Friday at 11p.m., as shown in the following code example:

```
@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 standard annotation.
For example, you have a class for handling unauthorized access exceptions. You annotate the class with one @Alert annotation for managers and another for admins:

```
@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 the Java compiler. In order for the compiler to do this, two declarations are required in your code.

Step 1: Declare a Repeatable Annotation Type:

The annotation type must be marked with the @Repeatable meta-annotation. Example, defines a custom @Schedule repeatable annotation type:

The value of the @Repeatable meta-annotation, in parentheses, is the type of the container annotation that the Java compiler generates to store repeating annotations. In this example, the containing annotation type is Schedules, so repeating @Schedule annotations is stored in an @Schedules annotation.

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

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

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

Repeating Annotations

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 must be the repeatable annotation type. The declaration for the Schedules containing annotation type is the following:

```
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 methods that return a single annotation, such as AnnotatedElement.getAnnotation(Class<T>), are unchanged in that they only return a single annotation if one annotation of the requested type is present. If more than one annotation of the requested type is present, you can obtain them by first getting their container annotation. In this way, legacy code continues to work. Other methods were introduced in Java SE 8 that scan through the container annotation to return multiple annotations at once, such as AnnotatedElement.getAnnotationsByType(Class<T>). See the AnnotatedElementclass specification for information on all of the available methods.

Design Considerations

When designing an annotation type, you must consider the cardinality of annotations of that type. It is now possible to use an annotation zero times, once, or, if the annotation's type is marked as @Repeatable, more than once. It is also possible to restrict where an annotation type can be used by using the @Target meta-annotation. For example, you can create a repeatable annotation type that can only be used on methods and fields. It is important to design your annotation type carefully to ensure that the programmer using the annotation finds it to be as flexible and powerful as possible.

Answers to Questions and Exercises

Questions

1. What is wrong with the following interface:

```
public interface House {
    @Deprecated
    public void open();
    public void openFrontDoor();
    public void openBackDoor();
}
```

The documentation should reflect why open is deprecated and what to use instead.

For example:

```
public interface House {
    /**
     * @deprecated use of open
     * is discouraged, use
     * openFrontDoor or
     * openBackDoor instead.
     */
    @Deprecated
    public void open();
    public void openFrontDoor();
    public void openBackDoor();
}
```

2. Consider this implementation of the House interface, shown in Question 1.

```
public class MyHouse implements House {
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

If you compile this program, the compiler produces a warning because open was deprecated (in the interface). What can you do to get rid of that warning?

You can deprecate the implementation of open:

```
public class MyHouse implements House {
    // The documentation is
    // inherited from the interface.
    @Deprecated
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

Alternatively, you can suppress the warning:

```
public class MyHouse implements House {
    @SuppressWarnings("deprecation")
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

Answers to Questions and Exercises

Questions

3.Will the following code compile without error? Why or why not?

```
public @interface Meal { ... }

@Meal("breakfast", mainDish="cereal")
@Meal("lunch", mainDish="pizza")
@Meal("dinner", mainDish="salad")
public void evaluateDiet() { ... }
```

The code fails to compile. Before JDK 8, repeatable annotations are not supported. As of JDK 8, the code fails to compile because the Meal annotation type was not defined to be repeatable. It can be fixed by adding the @Repeatable meta-annotation and defining a container annotation type:

```
public class AnnotationTest {

    public @interface MealContainer {
        Meal[] value();
    }

    @java.lang.annotation.Repeatable(MealContainer.class)
    public @interface Meal {
        String value();
        String mainDish();
    }

    @Meal(value="breakfast", mainDish="cereal")
    @Meal(value="lunch", mainDish="pizza")
    @Meal(value="dinner", mainDish="salad")
    public void evaluateDiet() { }
}
```

Exercise: Define an annotation type for an enhancement request with elements id, synopsis, engineer, and date. Specify the default value as unassigned for engineer and unknown for date.

```
/**
 * Describes the Request-for-Enhancement (RFE) annotation type.
 */
public @interface RequestForEnhancement {
    int id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date() default "[unknown]";
}
```