Chapter 27

Annotations

27.1 WHY HAVE ANNOTATIONS?

There are many things you can do with a program other than compiling and running it. Some examples are:

1. Automatic generation of documentation as with Scaladoc.
2. Pretty printing code so that it matches your preferred style.
3. Checking code for common errors such as opening a file but, on some control paths, never closing it.
4. Experimental type checking, for example to manage side effects or ensure ownership properties.

Such tools are called *meta-programming* tools, because they are programs that take other programs as input. Annotations support these tools by letting the programmer sprinkle directives to the tool throughout their source code. Such directives let the tools be more effective than if they could have no user input. For example, annotations can improve the previously listed tools as follows:

1. A documentation generator could be instructed to document certain methods as deprecated.
2. A pretty printer could be instructed to skip over parts of the program that have been carefully hand formatted.
3. A checker for non-closed files could be instructed to ignore a particular file that has been manually verified to be closed.
4. A side-effects checker could be instructed to verify that a specified method has no side effects.

In all of these cases, it would in theory be possible for the programming language to provide ways to insert the extra information. In fact, most of these are directly supported in some language or another. However, there are too many such tools for one language to directly support them all. Further, all of this information is completely ignored by the compiler, which after all just wants to make the code run.

Scala's philosophy in cases like this is to include the minimum, orthogonal support in the core language such that a wide variety of meta-programming tools can be written. In this case, that minimum support is a system of annotations. The compiler understands just one feature, annotations, but it doesn't attach any meaning to individual annotations. Each meta-programming tool can then define and use its own specific annotations.

27.2 SYNTAX OF ANNOTATIONS

A typical use of an annotation looks like this:

```
@deprecated def bigMistake() = //...
```

The annotation is the @deprecated part, and it applies to the entirety of the bigMistake method (not shown—it's too embarrassing). In this case, the method is being marked as something the author

of bigMistake wishes you not to use. Maybe bigMistake will be removed entirely from a future version of the code.

In the previous example, a method is annotated as @deprecated. Annotations are allowed in other places too. Annotations are allowed on any kind of declaration or definition, includingvals, vars, defs, classes, objects, traits, and types. The annotation applies to the entirety of the declaration or definition that follows it:

```
@deprecated class QuickAndDirty {
  //...
}
```

Annotations can also be applied to an expression, as with the @unchecked annotation for pattern matching (see Chapter 15). To do so, place a colon (:) after the expression and then write the annotation. Syntactically, it looks like the annotation is being used as a type:

```
(e: @unchecked) match {
  // non-exhaustive cases...
}
```

Finally, annotations can be placed on types. Annotated types are described later in this chapter.

So far the annotations shown have been simply an at sign followed by an annotation class. Such simple annotations are common and useful, but annotations have a richer general form:

@annot(*exp*_{1}, *exp*_{2}, ...)

The *annot* specifies the class of annotation. All annotations must include that much. The *exp*parts are arguments to the annotation. For annotations like @deprecated that do not need any arguments, you would normally leave off the parentheses, but you can write @deprecated() if you like. For annotations that do have arguments, place the arguments in parentheses, for example, @serial(1234).

The precise form of the arguments you may give to an annotation depends on the particular annotation class. Most annotation processors only let you supply immediate constants such as 123 or "hello". The compiler itself supports arbitrary expressions, however, so long as they type check. Some annotation classes can make use of this, for example, to let you refer to other variables that are in scope:

```
@cool val normal = "Hello"
@coolerThan(normal) val fonzy = "Heeyyy"
```

Internally, Scala represents an annotation as just a constructor call of an annotation class—replace the `@' by `new' and you have a valid instance creation expression. This means that named and default annotation arguments are supported naturally, because Scala already has named and default arguments for method and constructor calls. One slightly tricky bit concerns annotations that conceptually take other annotations as arguments, which are required by some frameworks. You cannot write an annotation directly as an argument to an annotation, because annotations are not valid expressions. In such cases you must use `new' instead of `@', as illustrated here:

```
scala> import annotation._
```

```
import annotation._

scala> class strategy(arg: Annotation) extends Annotation
defined class strategy

scala> class delayed extends Annotation
defined class delayed

scala> @strategy(@delayed) def f() = {}
<console>:1: error: illegal start of simple expression
       @strategy(@delayed) def f() = {}
                 ^
scala> @strategy(new delayed) def f() = {}
f: ()Unit
```

27.3 STANDARD ANNOTATIONS

Scala includes several standard annotations. They are for features that are used widely enough to merit putting in the language specification, but that are not fundamental enough to merit their own syntax. Over time, there should be a trickle of new annotations that are added to the standard in just the same way.

Deprecation

Sometimes you write a class or method that you later wish you had not. Once it is available, though, code written by other people might call the method. Thus, you cannot simply delete the method, because this would cause other people's code to stop compiling.

Deprecation lets you gracefully remove a method or class that turns out to be a mistake. You mark the method or class as deprecated, and then anyone who calls that method or class will get a deprecation warning. They had better heed this warning and update their code! The idea is that after a suitable amount of time has passed, you feel safe in assuming that all reasonable clients will have stopped accessing the deprecated class or method and thus that you can safely remove it.

You mark a method as deprecated simply by writing @deprecated before it. For example:

```
@deprecated def bigMistake() =  //...
```

Such an annotation will cause the Scala compiler to emit deprecation warnings whenever Scala code accesses the method.

If you supply a string as an argument to @deprecated, that string will be emitted along with the error message. Use this message to explain to developers what they should use instead of the deprecated method.

```
@deprecated("use newShinyMethod() instead")
def bigMistake() =  //...
```

Now any callers will get a message like this:

```
$ scalac -deprecation Deprecation2.scala
Deprecation2.scala:33: warning: method bigMistake in object
```

```
Deprecation2 is deprecated: use newShinyMethod() instead
    bigMistake()
    ^
one warning found
```

Volatile fields

Concurrent programming does not mix well with shared mutable state. For this reason, the focus of Scala's concurrency support is message passing and a minimum of shared mutable state. See Chapter 32 for the details.

Nonetheless, sometimes programmers want to use mutable state in their concurrent programs. The @volatile annotation helps in such cases. It informs the compiler that the variable in question will be used by multiple threads. Such variables are implemented so that reads and writes to the variable are slower, but accesses from multiple threads behave more predictably.

The @volatile keyword gives different guarantees on different platforms. On the Java platform, however, you get the same behavior as if you wrote the field in Java code and marked it with the Java volatile modifier.

Binary serialization

Many languages include a framework for binary *serialization*. A serialization framework helps you convert objects into a stream of bytes and *vice versa*. This is useful if you want to save objects to disk or send them over the network. XML can help with the same goals (seeChapter 28), but it has different trade offs regarding speed, space usage, flexibility, and portability.

Scala does not have its own serialization framework. Instead, you should use a framework from your underlying platform. What Scala does is provide three annotations that are useful for a variety of frameworks. Also, the Scala compiler for the Java platform interprets these annotations in the Java way (see Chapter 31).

The first annotation indicates whether a class is serializable at all. Most classes are serializable, but not all. A handle to a socket or GUI window, for example, cannot be serialized. By default, a class is not considered serializable. You should add a @serializableannotation to any class you would like to be serializable.

The second annotation helps deal with serializable classes changing as time goes by. You can attach a serial number to the current version of a class by adding an annotation like@SerialVersionUID(1234), where 1234 should be replaced by your serial number of choice. The framework should store this number in the generated byte stream. When you later reload that byte stream and try to convert it to an object, the framework can check that the current version of the class has the same version number as the version in the byte stream. If you want to make a serialization-incompatible change to your class, then you can change the version number. The framework will then automatically refuse to load old instances of the class.

Finally, Scala provides a @transient annotation for fields that should not be serialized at all. If you mark a field as @transient, then the framework should not save the field even when the surrounding object is serialized. When the object is loaded, the field will be restored to the default value for the type of the field annotated as @transient.

Automatic get and set methods

Scala code normally does not need explicit get and set methods for fields, because Scala blends the syntax for field access and method invocation. Some platform-specific frameworks do expect get and set methods, however. For that purpose, Scala provides the@scala.reflect.BeanProperty annotation. If you add this annotation to a field, the compiler will automatically generate get and set methods for you. If you annotate a field named crazy, theget method will be named getCrazy and the set method will be named setCrazy.

The generated get and set methods are only available after a compilation pass completes. Thus, you cannot call these get and set methods from code you compile at the same time as the annotated fields. This should not be a problem in practice, because in Scala code you can access the fields directly. This feature is intended to support frameworks that expect regularget and set methods, and typically you do not compile the framework and the code that uses it at the same time.

Tailrec

You would typically add the @tailrec annotation to a method that needs to be tail recursive, for instance because you expect that it would recurse very deeply otherwise. To make sure that the Scala compiler does perform the tail-recursion optimization described in Section 8.9on this method, you can add @tailrec in front of the method definition. If the optimization cannot be performed, you will then get a warning together with an explanation of the reasons.

Unchecked

The @unchecked annotation is interpreted by the compiler during pattern matches. It tells the compiler not to worry if the match expression seems to leave out some cases. See Section 15.5for details.

Native methods

The @native annotation informs the compiler that a method's implementation is supplied by the runtime rather than in Scala code. The compiler will toggle the appropriate flags in the output, and it will be up to the developer to supply the implementation using a mechanism such as the Java Native Interface (JNI).

When using the @native annotation, a method body must be supplied, but it will not be emitted into the output. For example, here is how to declare that method beginCountdown will be supplied by the runtime:

```
@native
def beginCountdown() = {}
```

27.4 CONCLUSION

This chapter described the platform-independent aspects of annotations that you will most commonly need to know about. First of all it covered the syntax of annotations, because using annotations is far more common than defining new ones. Second it showed how to use several annotations that are supported by the standard Scala compiler,
including @deprecated,@volatile, @serializable, @BeanProperty, @tailrec, and @unchecked.

Chapter 31 gives additional, Java-specific information on annotations. It covers annotations only available when targeting Java, additional meanings of standard annotations when targeting Java, how to interoperate with Java-based annotations, and how to use Java-based mechanisms to define and process annotations in Scala.

Annotations are structured information added to program source code. Like comments, they can be sprinkled throughout a program and attached to any variable, method, expression, or other program element. Unlike comments, they have structure, thus making them easier to machine process.

This chapter shows how to use annotations in Scala. It shows their general syntax and how to use several standard annotations.

This chapter does not show how to write new annotation processing tools, because it is beyond the scope of this book. Chapter 31 shows one technique, but not the only one. Instead, this chapter focuses on how to use annotations, because it is more common to use annotations than to define new annotation processors.