Lesson 15

Assertions and Annotations

J2SE 5.0 introduced a new facility known as annotations. Annotations are a metaprogramming facility that allow you to mark code with arbitrarily defined tags. The tags (generally) have no meaning to the Java compiler or runtime itself. Instead, other tools can interpret these tags. Examples of tools for which annotations might be useful include IDEs, testing tools, profiling tools, and code-generation tools.

In this lesson you will begin to build a testing tool, similar to JUnit, based upon Java's annotation capabilities. The testing tool, like JUnit, will need to allow developers to specify assertions. Instead of coding assertion methods, you will learn to use Java's built-in assertion capability.

You will learn about:

- assertions
- annotations and annotation types
- retention policies for annotations
- annotation targets
- member-value pairs
- default values for annotation members
- allowable annotation member types
- package annotations
- compatibility considerations for annotations

Assertions

You have been using `assert` methods defined as part of the JUnit API. These `assert` methods, defined in junit.framework.Assert, throw an AssertionFailedError when appropriate.

Java supports a similar assertion feature that you can turn on or off using a VM flag. An assertion statement begins with the keyword `assert`. It is followed by a conditional; if the conditional fails, Java throws a RuntimeException of type AssertionError. You can optionally supply a message to store in the AssertionError by following the conditional with a colon and the String message. An example:

```
assert name != null : "name is required";
```

Without the message:

```
assert name != null;
```

Assertions are disabled by default. When assertions are disabled, the VM ignores `assert` statements. This prevents `assert` statements from adversely affecting application performance. To turn assertions on:

```
java -ea MainClass
```

or, more explicitly:

```
java -enableassertions MainClass
```

Both of these statements will enable assertions for all your code but not for the Java library classes. You can turn assertions off using `-da` or `-disableassertions`. To turn assertions on or off for system classes, use `-enablesystemassertions` (or `-esa`) or `-disablesystemassertions` (or `-dsa`).

Java allows you to enable or disable assertions at a more granular level. You can turn assertions on or off for any individual class, any package and all its subpackages, or the default package.

For example, you might want to enable assertions for all but one class in a package:

```
java -ea:sis.studentinfo... -da:sis.studentinfo.Session SisApplication
```

The example shown will enable assertions for all classes in `sis.studentinfo` with the exception of Session. The ellipses means that Java will additionally enable assertions for any subpackages of `sis.studentinfo`, such as `sis.studentinfo.ui`. You represent the default package with just the ellipses.

Assertions are disabled/enabled in the order in which they appear on the `java` command line, from left to right.

The assert Statement vs. JUnit Assert Methods

JUnit was built in advance of the Java assertion mechanism, which Sun introduced in J2SE version 1.4. Today, JUnit could be rewritten to use the assertion mechanism. But since use of JUnit is very entrenched, such a JUnit rewrite would be a major undertaking for many shops. Also, the junit.framework.Assert methods supplied by JUnit are slightly more expressive. The `assertEquals` methods automatically provide an improved failure message.

When doing test-driven development, you will always need an assertion-based framework. The use of tests in TDD is analogous to design by contract (referred to as the subcontracting principle in Lesson 6). The assertions provide the preconditions, postconditions, and invariants. If you are not doing TDD, you might choose to bolster the quality of your system by introducing `assert` statements directly in the production code.

Sun recommends against using assertions as safeguards against application failure. For example, one potential use would be to check parameters of `public` methods, failing if client code passes a `null` reference. The downside is that the application may no longer work properly if you turn off assertions. For similar reasons, Sun recommends that you do not include code in assertions that produces side effects (i.e., code that alters system state).

Nothing prohibits you from doing so, however. If you have control over how your application is executed (and you should), then you can ensure that the application is always initiated with assertions enabled properly.

The chief value of Java's `assert` keyword would seem to be as a debugging aid. Well-placed `assert` statements can alert you to problems at their source. Suppose someone calls a method with a `null` reference. Without a protective assertion, you might not receive a NullPointerException until well later in the execution, far from the point where the reference was set. Debugging in this situation can be very time consuming.

You also could use assertions to help build a TDD tool in place of JUnit. In the exercise in this chapter, you will use assertions to begin building such a framework.

Annotations

You have already seen some examples of built-in *annotations* that Java supports. In Lesson 2 you learned how you can mark a method as `@deprecated`, meaning that the method is slated for eventual removal from the public interface of a class. The `@deprecated` annotation is technically not part of the Java language. Instead, it is a tag that the compiler can interpret and use as a basis for printing a warning message if appropriate.[1]

You have also seen examples of javadoc tags, annotations that you embed in javadoc comments and that you can use to generate Java API documentation web pages. The javadoc.exe program reads through your Java source files, parsing and interpreting the javadoc tags for inclusion in the web output.

Also, in Lesson 9, you learned how you could use the `@Override` tag to indicate that you believed you were overriding a superclass method.

You can also build your own annotation tags for whatever purpose you might find useful. For example, you might want the ability to mark methods with change comments:

```
@modified("JJL", "12-Feb-2005")
public void cancelReservation() {
   // ...
}
```

Subsequently, you could build a tool (possibly an Eclipse plug-in) that would allow you to view at a glance all the change comments, perhaps sorted by initials.

You could achieve equivalent results by insisting that developers provide structured comments that adhere to a standard format. You could then write code to parse through the source file, looking for these comments. However, the support in Java for custom annotation types provides significant advantages.

First, Java validates annotations at compile time. It is not possible to introduce a misspelled or incorrectly formatted annotation. Second, instead of having to write parse code, you can use reflection capabilities to quickly gather annotation information. Third, you can restrict annotations so that they apply to a specific kind of Java element. For example, you can insist that the `@modified` tag applies only to methods.

Building a Testing Tool

In this lesson, you will build a testing tool similar to JUnit. I'll refer to this tool as TestRunner, since that will be the primary class responsible for executing the tests. The tool will be text-based. You will be able to execute it from Ant. Technically, you don't have to build a

[1]`@deprecated` existed in the Java language from its early days and long before Sun introduced formalized annotations support. But it works just like any other compiler-level annotation type.

testing tool using TDD, because it's not intended to be production code. But there's nothing that says you *can't* write tests. We will.

JUnit requires a test class to extend from the class junit.framework.TestCase. In TestRunner you will use a different mechanism than inheritance: You will use Java annotations to mark a class as a test class.

Getting started is the toughest part, but using Ant can help. You can set up an Ant target to represent the user interface for the test. If not all of the tests pass, you can set things up so that the Ant build fails, in which case you will see a "BUILD FAILED" message. Otherwise you will see a "BUILD SUCCESSFUL" message.

TestRunnerTest

The first test in TestRunnerTest, `singleMethodTest`, goes up against a secondary class SingleMethodTest defined in the same source file. SingleMethodTest provides a single empty test method that should result in a pass, as it would in JUnit.

So far you need no annotations. You pass a reference to the test class, TestRunnerTest.class, to an instance of TestRunner. TestRunner can assume that this parameter is a test class containing only test methods.

To generate test failures, you will use the `assert` facility in Java, described in the first part of this lesson. Remember that you must enable assertions when executing the Java VM; otherwise, Java will ignore them.

Here is TestRunnerTest.

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

public class TestRunnerTest {
   public void singleMethodTest() {
      TestRunner runner = new TestRunner(SingleMethodTest.class);

      Set<Method> testMethods = runner.getTestMethods();
      assert 1 == testMethods.size() : "expected single test method";

      Iterator<Method> it = testMethods.iterator();
      Method method = it.next();

      final String testMethodName = "testA";
      assert testMethodName.equals(method.getName()) :
         "expected " + testMethodName + " as test method";
      runner.run();
      assert 1 == runner.passed() : "expected 1 pass";
      assert 0 == runner.failed() : "expected no failures";
   }
   public void multipleMethodTest() {
      TestRunner runner = new TestRunner(MultipleMethodTest.class);
      runner.run();

      assert 2 == runner.passed() : "expected 2 pass";
      assert 0 == runner.failed() : "expected no failures";

      Set<Method> testMethods = runner.getTestMethods();
      assert 2 == testMethods.size() : "expected single test method";

      Set<String> methodNames = new HashSet<String>();
      for (Method method: testMethods)
         methodNames.add(method.getName());

      final String testMethodNameA = "testA";
      final String testMethodNameB = "testB";

      assert methodNames.contains(testMethodNameA):
         "expected " + testMethodNameA + " as test method";
      assert methodNames.contains(testMethodNameB):
         "expected " + testMethodNameB + " as test method";
   }
}

class SingleMethodTest {
   public void testA() {}
}

class MultipleMethodTest {
   public void testA() {}
   public void testB() {}
}
```

The second test, `multipleMethodTest`, is a bit of a mess. To create it, I duplicated the first test and modified some of the details. It screams out for refactoring. The problem is that as soon as you extract a common utility method in TestRunnerTest, the TestRunner class will assume it's a test method and attempt to execute it. The solution will be to introduce an annotation that you can use to mark and distinguish test methods.

TestRunner

First, let's go over the initial implementation of TestRunner.

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;
```

```
class TestRunner {
   private Class testClass;
   private int failed = 0;
   private Set<Method> testMethods = null;

   public static void main(String[] args) throws Exception {
      TestRunner runner = new TestRunner(args[0]);
      runner.run();
      System.out.println(
         "passed: " + runner.passed() + " failed: " + runner.failed());
      if (runner.failed() > 0)
         System.exit(1);
   }

   public TestRunner(Class testClass) {
      this.testClass = testClass;
   }
   public TestRunner(String className) throws Exception {
      this(Class.forName(className));
   }

   public Set<Method> getTestMethods() {
      if (testMethods == null)
         loadTestMethods();
      return testMethods;
   }

   private void loadTestMethods() {
      testMethods = new HashSet<Method>();
      for (Method method: testClass.getDeclaredMethods())
         testMethods.add(method);
   }

   public void run() {
      for (Method method: getTestMethods())
         run(method);
   }

   private void run(Method method) {
      try {
         Object testObject = testClass.newInstance();
         method.invoke(testObject, new Object[] {})[2];
      }
      catch (InvocationTargetException e) {
         Throwable cause = e.getCause();
         if (cause instanceof AssertionError)
            System.out.println(cause.getMessage());
         else
            e.printStackTrace();
         failed++;
```

[2]You can use the slightly-more-succinct idiom `new Object[0]` in place of `new Object[] {}`.

```
      }
      catch (Throwable t) {
         t.printStackTrace();
         failed++;
      }
   }

   public int passed() {
      return testMethods.size() - failed;
   }

   public int failed() {
      return failed;
   }
}
```

If you're having a bit of trouble understanding the `run(Method)` method, refer to Lesson 12 for a discussion of reflection. The basic flow in the `run` method is:

- Create a new instance of the test class. This step assumes a no-argument constructor is available in the test class.
- `invoke` the method (passed in as a parameter) using the new test class instance and an empty parameter list.
- If the `invoke` message send fails, extract the cause from the thrown InvocationTargetException; the cause should normally be an AssertionError. Java throws an AssertionError when an `assert` statement fails.

TestRunner

TestRunner supplies two constructors. One takes a Class object and for now will be used from TestRunnerTest only. The second constructor takes a class name String and loads the corresponding class using `Class.forName`. You call this constructor from the `main` method, which provides a bit of user interface for displaying test results.

The `main` method in turn gets the class name from the Ant target:

```
<target name="runAllTests" depends="build" description="run all tests">
  <java classname="sis.testing.TestRunner" failonerror="true" fork="true">
    <classpath refid="classpath" />
    <jvmarg value="-enableassertions"/>
    <arg value="sis.testing.TestRunnerTest" />
  </java>
</target>
```

There are a few interesting things in the `runAllTests` target:

- You specify `failonerror="true"` in the `java` task. If running a Java application returns a nonzero value, Ant considers the execution to have resulted in an error. The build script terminates on an error. Using the `System.exit` command (see the main method in TestRunner) allows you to terminate an application immediately and return the value passed as a parameter to it.

- You specify fork="true" in the java task. This means that the Java application executes as a separate process from the Java process in which Ant executes. The pitfall of not forking is that the Ant build process itself will crash if the Java application crashes.
- You pass the test name to TestRunner using a nested arg element.
- You pass the argument enableassertions to the Java VM using a nested jvmarg element.

The @TestMethod Annotation

In order to be able to refactor tests, you must be able to mark the test methods so that other newly extracted methods are not considered tests. The @TestMethod annotation precedes the method signature for each method you want to designate as a test. Annotate the two test methods (singleMethodTest and multipleMethodTest) in TestRunnerTest with @TestMethod. Also annotate the three additional test methods in the miniclasses (SingleMethodTest and MultipleMethodTest) used by the TestRunnerTest tests.

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

public class TestRunnerTest {
   private TestRunner runner;
   private static final String methodNameA = "testA";
   private static final String methodNameB = "testB";

   @TestMethod
   public void singleMethodTest() {
      runTests(SingleMethodTest.class);
      verifyTests(methodNameA);
   }

   @TestMethod
   public void multipleMethodTest() {
      runTests(MultipleMethodTest.class);
      verifyTests(methodNameA, methodNameB);
   }
   private void runTests(Class testClass) {
      runner = new TestRunner(testClass);
      runner.run();
   }

   private void verifyTests(String... expectedTestMethodNames) {
      verifyNumberOfTests(expectedTestMethodNames);
      verifyMethodNames(expectedTestMethodNames);
      verifyCounts(expectedTestMethodNames);
   }

   private void verifyCounts(String... testMethodNames) {
      assert testMethodNames.length == runner.passed() :
         "expected " + testMethodNames.length + " passed";
      assert 0 == runner.failed() : "expected no failures";
   }

   private void verifyNumberOfTests(String... testMethodNames) {
      assert testMethodNames.length == runner.getTestMethods().size() :
         "expected " + testMethodNames.length + " test method(s)";
   }

   private void verifyMethodNames(String... testMethodNames) {
      Set<String> actualMethodNames = getTestMethodNames();
      for (String methodName: testMethodNames)
         assert actualMethodNames.contains(methodName):
            "expected " + methodName + " as test method";
   }

   private Set<String> getTestMethodNames() {
      Set<String> methodNames = new HashSet<String>();
      for (Method method: runner.getTestMethods())
         methodNames.add(method.getName());
      return methodNames;
   }
}

class SingleMethodTest {
   @TestMethod public void testA() {}
}

class MultipleMethodTest {
   @TestMethod public void testA() {}
   @TestMethod public void testB() {}
}
```

The `@TestMethod` annotation may appear after any method modifiers such as `public` or `static`. The annotation must appear before the signature of the method (which starts with the return type of the method).

To declare the `@TestMethod` annotation type, you create what looks a lot like an interface declaration:

```
package sis.testing;
public @interface TestMethod {}
```

The only difference between an interface declaration and an annotation type declaration is that you put an "at" sign (`@`) before the keyword `interface` in an interface declaration. There can be space between `@` and `interface`, but the convention is to abut the two.

The code will now compile and you can execute your tests, but you will see at least one IllegalAccessException stack trace. The code in TestRunner still treats every method as a test method, including the private methods you just extracted. The reflection code in TestRunner is unable to invoke these private methods. It's time to introduce code in TestRunner to look for the @TestMethod annotations:

```
private void loadTestMethods() {
   testMethods = new HashSet<Method>();
   for (Method method: testClass.getDeclaredMethods())
      if (method.isAnnotationPresent(TestMethod.class))
         testMethods.add(method);
}
```

One line of code is all it takes. You send the message `isAnnotationPresent` to the Method object, passing in the type (TestMethod.class) of the annotation. If `isAnnotationPresent` returns `true`, you add the Method object to the list of tests.

Now the test executes but returns improper results:

```
runAllTests:
     [java] passed: 0 failed: 0
```

You're expecting to see two passed tests but no tests are registered—the `isAnnotationPresent` method is always returning `false`.

Retention

The java.lang.annotations package includes a *meta*-annotation type named `@Retention`. You use meta-annotations to annotate other annotation type declarations. Specifically, you use the `@Retention` annotation to tell the Java compiler how long to retain annotation information. There are three choices, summarized in Table 15.1.

As explained by the table, if you don't specify an `@Retention` annotation, the default behavior means that you probably won't be able to extract informa-

Table 15.1 *Annotation Retention Policies*

RetentionPolicy enum	Annotation Disposition
RetentionPolicy.SOURCE	Discarded at compile time
RetentionPolicy.CLASS (default)	Stored in the class file; can be discarded by the VM at runtime
RetentionPolicy.RUNTIME	Stored in the class file; retained by the VM at runtime

tion on the annotation at runtime.[3] An example of `RetentionPolicy.CLASS` is the `@Override` annotation discussed in Lesson 9.

You will have the most need for `RetentionPolicy.RUNTIME` so that tools such as your TestRunner will be able to extract annotation information from their target classes. If you build tools that work directly with source code (for example, a plug-in for an IDE such as Eclipse), you can use `RetentionPolicy.SOURCE` to avoid unnecessarily storing annotation information in the class files. An example use might be an `@Todo` annotation used to mark sections of code that need attention.

To get reflection code to recognize `@TestMethod` annotations, you must modify the annotation type declaration:

```
package sis.testing;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
public @interface TestMethod {}
```

Your two TestRunnerTest tests should now pass:

```
runAllTests:
     [java] passed: 2 failed: 0
```

Annotation Targets

You have designed the `@TestMethod` annotation to be used by developers to mark test methods. Annotations can modify many other *element types*: types (classes, interfaces, and enums), fields, parameters, constructors, local vari-

[3]The VM may choose to retain this information.

ables, and packages. By default, you may use an annotation to modify any element. You can also choose to constrain an annotation type to modify one *and only one* element type. To do so, you supply an `@Target` meta-annotation on your annotation type declaration.

Since you didn't specify an `@Target` meta-annotation for `@TestMethod`, a developer could use the tag to modify any element, such as a field. Generally no harm would be done, but a developer could mark a field by accident, thinking that he or she marked a method. The test method would be ignored until someone noticed the mistake. Adding an `@Target` to an annotation type is one more step toward helping a developer at compile time instead of making him or her decipher later troubles.

Add the appropriate `@Target` meta-annotation to `@TestMethod`:

```
package sis.testing;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface TestMethod {}
```

The parameter to `@Target` must be an ElementType enum, defined in java.lang.annotation. This enum supplies constants corresponding to the element types that an annotation type can modify: `TYPE`, `FIELD`, `METHOD`, `PARAMETER`, `CONSTRUCTOR`, `LOCAL_VARIABLE`, `ANNOTATION_TYPE`, and `PACKAGE`. There are special considerations for annotations with a target of `ElementType.PACKAGE`. See the section Package Annotations later in this lesson for more information.

To demonstrate what `@Target` does for your annotation types, modify TestRunnerTest. Instead of marking a method with `@TestMethod`, mark a field instead:

```
// ...
public class TestRunnerTest {
   private @TestMethod TestRunner runner;

   @TestMethod
   public void singleMethodTest() {
   // ...
```

When you compile, you should see an error similar to the following:

```
annotation type not applicable to this kind of declaration
private @TestMethod TestRunner runner;
        ^
```

Remove the extraneous annotation, recompile, and rerun your tests.

Skipping Test Methods

From time to time, you may want to bypass the execution of certain test methods. Suppose you have a handful of failing methods. You might want to concentrate on getting a green bar on one failed method at a time before moving on to the next. The failure of the other test methods serves as a distraction. You want to "turn them off."

In JUnit, you can skip a method by either commenting it out or by renaming the method so that its signature does not represent a test method signature. An easy way to skip a test method is to precede its name with an `X`. For example, you could rename `testCreate` to `XtestCreate`. JUnit looks for methods whose names start with the word `test`, so it will not find `XtestCreate`.

You do not want to make a habit of commenting out tests. It is poor practice to leave methods commented out for any duration longer than your current programming session. You should avoid checking in code with commented-out tests. Other developers won't understand your intent. My first inclination when I see commented-out code, particularly test code, is to delete it.

Commenting out test methods is risky. It is easy to forget that you have commented out tests. It can also be difficult to find the tests that are commented out. It would be nice if JUnit could warn you that you've left tests commented out.

A similar problem exists for your new TestRunner class. The simplest way of bypassing a test would be to remove its `@TestMethod` annotation. The problem with doing that is the same as the problem with commenting out a test. It's easy to "lose" a test in a system with any significant number of tests.

For this exercise, you will make the necessary modifications to TestRunner to ignore designated methods. You will create a new annotation type, `@Ignore`, and change code in TestRunner to recognize this annotation. The `@Ignore` annotation will allow developers to supply a parameter with a text description of why the test is being skipped. You will modify TestRunner to print these descriptions.

Modifying TestRunner

Add a new test method, `ignoreMethodTest`, to TestRunnerTest. It will go up against a new test class, IgnoreMethodTest, which contains three methods marked with `@TestMethod`. One of the test methods (`testC`) is additionally marked `@Ignore`. You must verify that this test method is not executed.

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

public class TestRunnerTest {
   private TestRunner runner;
   // ...
   @TestMethod
   public void ignoreMethodTest() {
      runTests(IgnoreMethodTest.class);
      verifyTests(methodNameA, methodNameB);
   }
   // ...
}

// ...
class IgnoreMethodTest {
   @TestMethod public void testA() {}
   @TestMethod public void testB() {}

   @Ignore
   @TestMethod public void testC() {}
}
```

The `@Ignore` annotation declaration looks a lot like the `@TestMethod` declaration.

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {}
```

To get your tests to pass, make the following modification to TestRunner.

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

class TestRunner {
   ...
   private void loadTestMethods() {
      testMethods = new HashSet<Method>();
      for (Method method: testClass.getDeclaredMethods())
         if (method.isAnnotationPresent(TestMethod.class) &&
             !method.isAnnotationPresent(Ignore.class))
            testMethods.add(method);
   }
   ...
}
```

Single-Value Annotations

The `@Ignore` annotation is a *marker* annotation—it marks whether a method should be ignored or not. You merely need to test for the presence of the annotation using `isAnnotationPresent`. Now you need developers to supply a reason for ignoring a test. You will modify the `@Ignore` annotation to take a reason String as a parameter.

To support a single parameter in an annotation type, you supply a member method named `value` with an appropriate return type and no parameters. Annotation type member methods cannot take any parameters.

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
   String value();
}
```

Mark one of the methods in IgnoreMethodTest with just `@Ignore`. Do not supply any parameters:

```
class IgnoreMethodTest {
   @TestMethod public void testA() {}
   @TestMethod public void testB() {}
   @Ignore()
      @TestMethod public void testC() {}
}
```

Note that `@Ignore` is a shortcut for `@Ignore()`.

When you compile, you will see an error message:

```
annotation testing.Ignore is missing value
   @Ignore
    ^
```

The compiler uses the corresponding annotation type declaration to ensure that you supplied the proper number of parameters.

Change the test target class, IgnoreMethodTest, to supply a reason with the `@Ignore` annotation.

```
public class TestRunnerTest {
   public static final String IGNORE_REASON1 = "because";
   // ...
}
class IgnoreMethodTest {
   @TestMethod public void testA() {}
   @TestMethod public void testB() {}
   @Ignore(TestRunnerTest.IGNORE_REASON1)
      @TestMethod public void testC() {}
}
```

Rerun your tests. They pass, so you haven't broken anything. But you also want the ability to print a list of the ignored methods. Modify the test accordingly:

```
@TestMethod
public void ignoreMethodTest() {
   runTests(IgnoreMethodTest.class);
   verifyTests(methodNameA, methodNameB);
   assertIgnoreReasons();
}

private void assertIgnoreReasons() {
   Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
   Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
   assert "testC".equals(entry.getKey().getName()):
      "unexpected ignore method: " + entry.getKey();
   Ignore ignore = entry.getValue();
   assert IGNORE_REASON1.equals(ignore.value());
}

private <K, V> Map.Entry<K, V> getSoleEntry(Map<K, V> map) {
   assert 1 == map.size(): "expected one entry";
   Iterator<Map.Entry<K, V>> it = map.entrySet().iterator();
   return it.next();
}
```

You return the ignored methods as a collection of mappings between Method objects and the "ignored reason" string. Since you expect there to be only one ignored method, you can introduce the utility method getSoleEntry to extract the single Method key from the Map. In my excitement over figuring out how to use generics (see Lesson 14), I've gone a little overboard here and made getSoleEntry into a generic method that you could use for any collection. There's no reason you couldn't code it specifically to the key and value types of the map.

Now make the necessary changes to TestRunner to get it to store the ignored methods for later extraction:

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

class TestRunner {
   // ...
   private Map<Method, Ignore> ignoredMethods = null;
   // ...
   private void loadTestMethods() {
      testMethods = new HashSet<Method>();
      ignoredMethods = new HashMap<Method, Ignore>();
      for (Method method: testClass.getDeclaredMethods()) {
         if (method.isAnnotationPresent(TestMethod.class))
            if (method.isAnnotationPresent(Ignore.class)) {
               Ignore ignore = method.getAnnotation(Ignore.class);
               ignoredMethods.put(method, ignore);
            }
            else
               testMethods.add(method);
      }
   }

   public Map<Method, Ignore> getIgnoredMethods() {
      return ignoredMethods;
   }
   // ...
}
```

You can send the `getAnnotation` method to any element that can be annotated, passing it the annotation type name (Ignore.class here). The `getAnnotation` method returns an annotation type reference to the actual annotation object. Once you have the annotation object reference (`ignore`), you can send messages to it that are defined in the annotation type interface.

You can now modify the text user interface to display the ignored methods.

A TestRunner User Interface Class

At this point, the `main` method is no longer a couple of simple hacked-out lines. It's time to move this to a separate class responsible for presenting the user interface.

Since TestRunner is a utility for test purposes, as I mentioned earlier, tests aren't absolutely required. For the small bit of nonproduction user interface code you'll write for the test runner, you shouldn't feel compelled to test first. You're more than welcome to do so, but I'm not going to here.

The following listing shows a refactored user interface class that prints out ignored methods. About the only thing interesting in it is the "clever" way I return the number of failed tests in the `System.exit` call. Why? Why not? It's more succinct than an `if` statement, it doesn't obfuscate the code, and it returns additional information that the build script or operating system could use.

```
package sis.testing;

import java.lang.reflect.*;
import java.util.*;
```

```
public class TestRunnerUI {
   private TestRunner runner;

   public static void main(String[] args) throws Exception {
      TestRunnerUI ui = new TestRunnerUI(args[0]);
      ui.run();
      System.exit(ui.getNumberOfFailedTests());
   }

   public TestRunnerUI(String testClassName) throws Exception {
      runner = new TestRunner(testClassName);
   }

   public void run() {
      runner.run();
      showResults();
      showIgnoredMethods();
   }

   public int getNumberOfFailedTests() {
      return runner.failed();
   }

   private void showResults() {
      System.out.println(
         "passed: " + runner.passed() +
         " failed: " + runner.failed());
   }

   private void showIgnoredMethods() {
      if (runner.getIgnoredMethods().isEmpty())
         return;

      System.out.println("\nIgnored Methods");
      for (Map.Entry<Method, Ignore> entry:
            runner.getIgnoredMethods().entrySet()) {
         Ignore ignore = entry.getValue();
         System.out.println(entry.getKey() + ": " + ignore.value());
      }
   }
}
```

Array Parameters

You want to allow developers to provide multiple separate reason strings. To do so, you can specify String[] as the return type for the annotation type member `value`. An `@Ignore` annotation can then contain multiple reasons by using a construct that looks similar to an array initializer:

```
@Ignore({"why", "just because"})
```

Here's the updated annotation type declaration:

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
   String[] value();
}
```

If you only need to supply a single string for an annotation type member with a return type of String[], Java allows you to eliminate the array-style initialization. These annotations for the current definition of `@Ignore` are equivalent:

```
@Ignore("why")
@Ignore({"why"})
```

You will need to modify TestRunnerTest to support the change to Ignore.

Array Parameters

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

public class TestRunnerTest {
   public static final String IGNORE_REASON1 = "because";
   public static final String IGNORE_REASON2 = "why not";
   ...

   @TestMethod
   public void ignoreMethodTest() {
      runTests(IgnoreMethodTest.class);
      verifyTests(methodNameA, methodNameB);
      assertIgnoreReasons();
   }

   private void assertIgnoreReasons() {
      Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
      Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
      assert "testC".equals(entry.getKey().getName()):
         "unexpected ignore method: " + entry.getKey();
      Ignore ignore = entry.getValue();
      String[] ignoreReasons = ignore.value();
      assert 2 == ignoreReasons.length;
      assert IGNORE_REASON1.equals(ignoreReasons[0]);
      assert IGNORE_REASON2.equals(ignoreReasons[1]);
   }
   ...
}
```

```
class SingleMethodTest {
   @TestMethod public void testA() {}
}

class MultipleMethodTest {
   @TestMethod public void testA() {}
   @TestMethod public void testB() {}
}

class IgnoreMethodTest {
   @TestMethod public void testA() {}
   @TestMethod public void testB() {}

   @Ignore({TestRunnerTest.IGNORE_REASON1,
            TestRunnerTest.IGNORE_REASON2})
      @TestMethod public void testC() {}
}
```

Multiple Parameter Annotations

You may want annotations to support multiple parameters. As an example, suppose you want developers to add their initials when ignoring a test method. A proper annotation might be:

```
@Ignore(reasons={"just because", "and why not"}, initials="jjl")
```

Now that you have more than one annotation parameter, you must supply *member-value pairs*. Each member-value pair includes the member name, which must match an annotation type member, followed by the equals (`=`) sign, followed by the constant value for the member.

The second member-value pair in the above example has `initials` as a member name and `"jjl"` as its value. In order to support this annotation, you must modify the `@Ignore` annotation type declaration to include `initials` as an additional member. You must also rename the value member to `reasons`. Each key in an annotation member-value pair must match a member name in the annotation type declaration.

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
   String[] reasons();
   String initials();
}
```

You can specify member-value pairs in any order within an annotation. The order need not match the member order of the annotation type declaration.

Here are the corresponding modifications to TestRunnerTest:

```
package sis.testing;
// ...
public class TestRunnerTest {
   // ...
   public static final String IGNORE_INITIALS = "jjl";
   // ...
   private void assertIgnoreReasons() {
      Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
      Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
      assert "testC".equals(entry.getKey().getName()):
         "unexpected ignore method: " + entry.getKey();
      Ignore ignore = entry.getValue();
      String[] ignoreReasons = ignore.reasons();
      assert 2 == ignoreReasons.length;
      assert IGNORE_REASON1.equals(ignoreReasons[0]);
      assert IGNORE_REASON2.equals(ignoreReasons[1]);
      assert IGNORE_INITIALS.equals(ignore.initials());
   }
   // ...
}

class IgnoreMethodTest {
   @TestMethod public void testA() {}
   @TestMethod public void testB() {}

   @Ignore(
      reasons={TestRunnerTest.IGNORE_REASON1,
       TestRunnerTest.IGNORE_REASON2},
      initials=TestRunnerTest.IGNORE_INITIALS)
   @TestMethod public void testC() {}
}
```

You do not need to make modifications to TestRunner. You will need to make a small modification to TestRunnerUI to extract the reasons and initials properly from the Ignore object.

```
private void showIgnoredMethods() {
   if (runner.getIgnoredMethods().isEmpty())
      return;

   System.out.println("\nIgnored Methods");
   for (Map.Entry<Method, Ignore> entry:
         runner.getIgnoredMethods().entrySet()) {
      Ignore ignore = entry.getValue();
      System.out.printf("%s: %s (by %s)",
         entry.getKey(),
         Arrays.toString(ignore.reasons()),
         ignore.initials());
   }
}
```

Default Values

The reason for ignoring a test method is likely to be the same most of the time. Most often, you will want to temporarily comment out a test while fixing other broken tests. Supplying a reason each time can be onerous, so you would like to have a default ignore reason. Here's how you might reflect this need in a TestRunner test:

```
@TestMethod
public void ignoreWithDefaultReason() {
   runTests(DefaultIgnoreMethodTest.class);
   verifyTests(methodNameA, methodNameB);
   Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
   Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
   Ignore ignore = entry.getValue();
   assert TestRunner.DEFAULT_IGNORE_REASON.
      equals(ignore.reasons()[0]);
}

class DefaultIgnoreMethodTest {
   @TestMethod public void testA() {}
   @TestMethod public void testB() {}
   @Ignore(initials=TestRunnerTest.IGNORE_INITIALS)
      @TestMethod public void testC() {}
}
```

You will need to define the constant `DEFAULT_IGNORE_REASON` in the TestRunner class to be whatever string you desire:

```
class TestRunner {
   public static final String DEFAULT_IGNORE_REASON =
      "temporarily commenting out";
   // ...
```

You can supply a default value on any annotation type member. The default must be a constant at compile time. The new definition of @Ignore includes a default value on the `reasons` member. Note use of the keyword `default` to separate the member signature from the default value.

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
```

```
public @interface Ignore {
   String[] reasons() default TestRunner.DEFAULT_IGNORE_REASON;
   String initials();
}
```

Additional Return Types and Complex Annotation Types

In addition to String and String[], an annotation value can be a primitive, an enum, a Class reference, an annotation type itself, or an array of any of these types.

The following test (in TestRunnerTest) sets up the requirement for an `@Ignore` annotation to include a date. The Date type will be an annotation; its members each return an `int` value.

```
@TestMethod
public void dateTest() {
   runTests(IgnoreDateTest.class);
   Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
   Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
   Ignore ignore = entry.getValue();
   sis.testing.Date date = ignore.date();
   assert 1 == date.month();
   assert 2 == date.day();
   assert 2005 == date.year();
}

class IgnoreDateTest {
   @Ignore(
         initials=TestRunnerTest.IGNORE_INITIALS,
         date=@Date(month=1, day=2, year=2005))
      @TestMethod public void testC() {}
}
```

The annotation in IgnoreDateTest is known as a *complex annotation*—an annotation that includes another annotation. `@Ignore` includes a member, `date`, whose value is another annotation, `@Date`.

The definition of the sis.testing.Date annotation type:

```
package sis.testing;

public @interface Date {
   int month();
   int day();
   int year();
}
```

The `@Ignore` annotation type can now define a `date` member that returns a testing.Date instance:

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
   String[] reasons() default TestRunner.DEFAULT_IGNORE_REASON;
   String initials();
   Date date();
}
```

Since the Date annotation type is only used as part of another annotation, it does not need to specify a retention or a target.

You may not declare a recursive annotation type—that is, an annotation type member with the same return type as the annotation itself.

To get your tests to compile and pass, you'll also need to modify IgnoreMethodTest and DefaultIgnoreMethodTest:

```
class IgnoreMethodTest {
   @TestMethod public void testA() {}
   @TestMethod public void testB() {}

   @Ignore(
      reasons={TestRunnerTest.IGNORE_REASON1,
               TestRunnerTest.IGNORE_REASON2},
      initials=TestRunnerTest.IGNORE_INITIALS,
      date=@Date(month=1, day=2, year=2005))
   @TestMethod public void testC() {}
}

class DefaultIgnoreMethodTest {
   @TestMethod public void testA() {}
   @TestMethod public void testB() {}
   @Ignore(initials=TestRunnerTest.IGNORE_INITIALS,
         date=@Date(month=1, day=2, year=2005))
      @TestMethod public void testC() {}
}
```

Package Annotations

Suppose you want the ability to designate packages as testing packages. Further, you want your testing tool to run "performance-related" tests separately from other tests. In order to accomplish this, you can create an annotation whose target is a package. A test for this annotation:[4]

```
@TestMethod
public void packageAnnotations() {
   Package pkg = this.getClass().getPackage();
   TestPackage testPackage = pkg.getAnnotation(TestPackage.class);
   assert testPackage.isPerformance();
}
```

You extract annotation information from the Package object like you would from any other element object. You can obtain a Package object by sending the message `getPackage` to any Class object.

The annotation declaration is straightforward:

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.PACKAGE)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestPackage {
   boolean isPerformance() default false;
}
```

Package Annotations

However, the question is, Where does a package annotation go? Would it go before the `package` statement in every source file that belongs to the package? Or before just one of them? Or is it stored elsewhere?

Java limits you to at most one annotated package statement per package. This means that you can't just place an annotation before the `package` statement in an arbitrary source file.

The answer depends on which compiler you are using. Sun recommends a specific scheme that is based on a file system. Other compiler vendors may choose a different scheme. They may have to if the compile environment is not based on a file system.

The Sun scheme requires you to create a source file named package-info.java in the source directory that corresponds to the package you want to annotate. Sun's Java compiler reads this pseudo–source file but produces no visible class files as output. (In fact, it is not possible to include a hyphen [-] in a class name.) This file should include any package annotations, followed by the appropriate `package` statement. You should not include anything else in package-info.java.

Here's what package-info.java might look like in the `sis.testing` package:

```
@TestPackage(isPerformance=true) package sis.testing;
```

[4]Note use of the variable name `pkg`, since `package` is a keyword.

Compatibility Considerations

Sun has, as much as possible, designed the annotations facility to support changes to an annotation with minimal impact on existing code. This section goes through the various modification scenarios and explains the impact of each kind of change to an annotation type.

When you add a new member to an annotation type, provide a default if possible. Using a default will allow code to continue to use the compiled annotation type without issue. But this *can* cause a problem if you try to access the new member. Suppose you access the new member from an annotation compiled using the annotation type declaration *without* the new member. If no default exists on the new member, this will generate an exception.

If you remove an annotation type member, you will obviously cause errors on recompilation of any likewise annotated sources. However, any existing class files that use the modified annotation type will work fine until their sources are recompiled.

Avoid removing defaults, changing the return type, or removing target elements with existing annotation types. These actions all have the potential to generate exceptions.

If you change the retention type, the behavior is generally what you would expect. For example, if you change from `RUNTIME` to `CLASS`, the annotation is no longer readable at runtime.

When in doubt, write a test to demonstrate the behavior!

Additional Notes on Annotations

- An annotation with no target can modify any Java element.
- In an annotation type declaration, the only parameterized type that you can return is the Class type.
- The `@Documented` meta-annotation type lets you declare an annotation type to be included in the published API generated by tools such as javadoc.
- The `@Inherited` meta-annotation type means that an annotation type is inherited by all subclasses. It will be returned if you send `getAnnotation` to a method or class object but not if you send `getDeclaredAnnotations`.
- You cannot use `null` as an annotation value.

- You can modify an element only once with a given annotation. For example, you cannot supply two `@Ignore` annotations for a test method.
- In order to internally support annotation types, Sun modified the Arrays class to include implementations of `toString` and `hashCode` for Arrays.

Summary

Annotations are a powerful facility that can help structure the notes you put into your code. One of the examples that is touted the most is the ability to annotate interface declarations so that tools can generate code from the pertinent methods.

The chief downside of using annotations is that you make your code dependent upon an annotation type when your code uses it. Changes to the annotation type declaration could negatively impact your code, although Sun has built in some facilities to help maintain binary compatibility. You also must have the annotation type class file present in order to compile. This should not be surprising: An annotation type is effectively an interface type.

Again, the rule of thumb is to use annotation types prudently. An annotation type is an interface and should represent a stable abstraction. As with any interface, ensure that you have carefully considered the implications of introducing an annotation type into your system. A worst-case scenario, where you needed to make dramatic changes to an annotation type, would involve a massive search and replace[5] followed by a compile.

Exercises

1. Using the RingTest and Ring classes from the previous lesson, introduce an `assert` into the `add` method that rejects a `null` argument. Make sure you write a test! Don't forget to enable assertions before running your tests.

[5]For the time being, IDEs might not provide sophisticated support for manipulating and navigating annotations. You may need to resort to search and replace (or compile, identify, and replace) to effect a major change. Integrated IDE support for annotations should emerge quickly. The support in IDEA is already at a level I would consider acceptable.

2. Create an annotation @Dump. This annotation can be applied to any field in a class. Then create a ToStringer class which, when passed an object, will spit out a dump of each field that has been marked for use in the `toString` method.
3. Modify the dump annotation to allow for sorting the fields by adding an optional `order` parameter. The order should be a positive integer. If a field is not annotated, it should appear last in the list of fields.
4. Add another parameter for dump called `quote`, which should be a boolean indicating whether or not to surround the value with quotes. This is useful for objects whose `toString` representation might be empty or have leading or trailing spaces.
5. Add an `outputMethod` field to the @Dump annotation. This specifies the method for toStringer to use in order to get a printable representation of the field. Its value should default to `toString`. This is useful for when you have an object with a `toString` representation you cannot change, such as an object of a system class type.
6. Change the `outputMethod` annotation to `outputMethods` and have it support a String array of method names. The ToString code should construct a printable representation of the object by calling each of these method names in order and concatenating the results. Separate each result with a single space. (You might consider adding another annotation to designate the separator character.)