CHAPTER 13

Building

The *build process* is one of compiling the source files of a project and producing a finished product.

In some companies, there are whole teams whose sole job is the build process.

There are many other build tools, but I'm just going to cover three:

- Ant[1]
- Maven[2]
- Gradle[3]

13.1 Ant

Ant is the first really popular project builder for Java that existed. It is XML-based and requires you to create tasks in XML that can be executed by Ant.

A *task* is a division of work. Tasks depend on other tasks. For example, the "jar" task usually depends on the "compile" task. Although Maven threw away the task concept, it was used again in Gradle.

Critics of Ant complain that it uses XML (a much-loathed format) and requires a lot of work to do simple things.

13.2 Maven

Maven is an XML-based declarative project manager. Maven is used for building Java projects but is capable of much more. Maven is also a set of standards that allows Java/JVM developers to easily define and integrate dependencies into large projects. Maven somewhat replaces Ant but can also integrate with it and other build tools.

Maven was mostly a reaction to the huge number of open source libraries Java projects tend to rely on. It has a built-in standard for dependency management (managing the interdependencies of open source libraries).

[1] http://ant.apache.org/.
[2] http://maven.apache.org/.
[3] www.gradle.org/.

A. L. Davis, *Modern Programming Made Easy*, DOI 10.1007/978-1-4842-2490-8_13

Although Maven is an Apache open source project, it could be said that the core of Maven is *Maven Central*, a repository of open source libraries run by Sonatype, the company behind Maven. There are many other repositories that follow the Maven standard, such as JFrog's jCenter,[4] so you are not restricted to Maven Central.

Ivy[5] is a similar build tool, but is more closely related to Ant.

Many build tools, such as Ivy and Gradle, build on top of Maven's concept.

13.2.1 Using Maven

The main file that defines a Maven project is the *POM* (Project Object Model). The POM file is written in XML and contains all of the dependencies, plug-ins, properties, and configuration data that is specific to the current project. The POM file is generally composed of the following:

- Basic properties (artifactId, groupId, name, version, packaging)
- Dependencies
- Plug-ins

There is a Maven plug-in for every major Java-based IDE out there (Eclipse, NetBeans, and IntelliJ IDEA), and they are very helpful. You can use the Maven plug-in to create your project, add dependencies, and edit your POM files.

13.2.2 Starting a New Project

There is a simple way to create a new configuration file (`pom.xml`) and project folders using the `archetype:generate` command.

```
1   mvn archetype:generate
```

That will list all the different kinds of projects you can create. Pick a number representing the type of project you want (there are 726 options right now), then answer some questions regarding the name of your project. After that process, run the following command to build the project:

```
1   mvn package
```

If you want to use any additional third-party libraries, you will have to edit the POM to include each dependency. Fortunately, most IDEs make it easy to add dependencies to the POM.

[4] https://bintray.com/bintray/jcenter.
[5] http://ant.apache.org/ivy/.

Maven the Complete Reference Maven the Complete Reference[6] is available online if you want to learn more.

13.2.3 Life Cycle

Maven uses a declarative style (unlike Ant, which uses a more imperative approach). This means that instead of listing the steps to take, you describe what should happen during certain phases of the build. The phases in Maven are as follows:

- `validate`: Validates that the project is correct and all necessary information is available
- `compile`: Compiles the source code of the project
- `test`: Tests the compiled source code, using a suitable unit-testing framework
- `package`: Takes the compiled code and packages it in its distributable format, such as a JAR
- `integration-test`: Processes and deploys the package, if necessary, into an environment in which integration tests can be run
- `verify`: Runs any checks to verify that the package is valid and meets quality criteria
- `install`: Installs the package into the local repository, for use as a dependency in other projects locally
- `deploy`: Copies, in an integration or release environment, the final package to the remote repository, for sharing with other developers and projects

There are more phases,[7] but you don't need to know all of them until you are doing more complex builds.

[6] `www.sonatype.com/books/mvnref-book/reference/`.
[7] `https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference`.

13.2.4 Executing Code

Sometimes, however, you just need more control over your build. In Maven, you can execute Groovy code, Ant build files, Scala code, and you can even write your own plugins in Groovy.

For example, you can put Groovy code in your POM file in the following way:

```
<plugin>
 <groupId>org.codehaus.groovy.maven</groupId>
 <artifactId>gmaven-plugin</artifactId>
 <executions>
  <execution>
    <id>groovy-magic</id>
    <phase>prepare-package</phase>
    <goals>
      <goal>execute</goal>
    </goals>
      <configuration>
        <source>
          def depFile = new  File(project.build.outputDirectory,
          'deps.txt')

          project.dependencies.each() {
            depFile.write("${it.groupId}:${it.artifactId}:${it.
            version}")
          }

          ant.copy(todir: project.build.outputDirectory ) {
            fileset(dir: project.build.sourceDirectory)
          }
        </source>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The preceding code would write out every dependency of the project into the file `deps.txt`. Then it would copy all of the source files into the `project.build.outputDirectory` (usually `target/classes`).

See Chapters 2, 3, and 4 in *The Maven Cookbook*.[8]

[8] http://books.sonatype.com/mcookbook/reference/index.html

13.3 Gradle

Gradle is a Groovy-based DSL (domain specific language) system for building projects.

The Gradle web site describes it as follows:

> *Gradle combines the power and flexibility of Ant with the dependency management and conventions of Maven into a more effective way to build. Powered by a Groovy DSL and packed with innovation, Gradle provides a declarative way to describe all kinds of builds through sensible defaults.*
>
> —*gradle.org*[9]

13.3.1 Projects and Tasks

Each Gradle build is composed of one or more projects, and each project is composed of tasks.

The *core* of the Gradle build is the `build.gradle` file, which is called the *build script*. Tasks are defined by writing `task`, then a task-name followed by a closure. For example:

```
1   task upper {
2           String someString = 'test'
3           println "Original: $someString"
4           println "Uppercase: " + someString.toUpperCase()
5   }
```

Tasks can contain any Groovy code, but you can take advantage of existing Ant tasks; for example:

```
ant.loadfile(srcFile: file, property: 'x') //loads file into x
ant.checksum(file: file, property: "z") // put checksum into z
println ant.properties["z"] //accesses ant property z
```

The preceding code would load a file into Ant-property `"x"`, save the file's checksum in Ant-property `"z"`, and then print out that checksum.

Much as in Ant, a task can depend on other tasks, which means they must be run before the task. You simply call `dependsOn` with any number of task names as arguments. For example:

```
1   task buildApp {
2           dependsOn clean, installApp, processAssets
3   }
```

[9] www.gradle.org/.

13.3.2 Plug-ins

Gradle core has very little built-in. It has powerful plug-ins to allow it to be very flexible. A plug-in can do one or more of the following:

- Add tasks to the project (e.g., compile, test)
- Pre-configure added tasks with useful defaults
- Add dependency configurations to the project
- Add new properties and methods to existing type, via extensions

We're going to concentrate on building Java-based projects, so we'll be using the `java` plug-in; however, Gradle is not limited to Java projects!

```
1   apply plugin: 'java'
```

This plug-in uses Maven's conventions. For example, it expects to find your production source code under `src/main/java` and your test source code under `src/test/java`.

13.3.3 Maven Dependencies

Every Java project tends to rely on many open source projects to be built. Gradle builds on Maven, so you can easily include your dependencies, using a simple DSL, such as in the following example:

```
 1   apply plugin: 'java'
 2
 3   sourceCompatibility = 1.7
 4
 5   repositories {
 6           mavenLocal()
 7           mavenCentral()
 8   }
9
10   dependencies {
11           compile 'com.google.guava:guava:14.0.1'
12           compile 'org.bitbucket.dollar:dollar:1.0-beta2'
13           testCompile group: 'junit', name: 'junit', version: '4.+'
14           testCompile "org.mockito:mockito-core:1.9.5"
15   }
```

This build script uses `sourceCompatibility` to define the Java source code version of `1.7` (which is used during compilation). Next, it tells Maven to use the local repository first (`mavenLocal`), then Maven Central.

In the `dependencies` block, this build script defines two dependencies for the `compile` scope and two for `testCompile` scope. Jars in the `testCompile` scope are only used in tests and won't be included in any final products.

The line for JUnit shows the more verbose style for defining dependencies.

Online Documentation Gradle has a huge online user guide available online at gradle.org.[10]

[10] www.gradle.org/docs/current/userguide/userguide.html.

CHAPTER 14

Testing

Testing is a very important part of software creation. Without automated tests, it's very easy for bugs to creep into software.

In fact, some go as far as to say that you should write tests *before* you write the code. This is called *TDD* (test-driven development).

14.1 Types of Tests

The following are different types of tests you might write:

- *Unit test*: Test conducted on a single API call or some isolated code
- *Integration test*: Test of a higher order code that requires a test harness, mocking, etc.
- *Acceptance test*: High-level test that matches the business requirements
- *Compatibility*: Making sure that things work together
- *Functionality*: Ensuring that stuff works
- *Black box*: Test conducted without knowing/thinking about what's going on in the code
- *White box*: Tests written with the inside of code in mind
- *Gray box*: Hybrid of black and white box testing
- *Regression*: Creating a test after finding a bug, to ensure that the bug does not reappear
- *Smoke*: A huge sampling of data
- *Load/Stress/Performance*: How the system handles load

The type and number of tests you write vary, based on a number of factors. The simpler a piece of code is, the less testing it requires. For example, a "getter" or "setter" does not require a test at all.

A. L. Davis, *Modern Programming Made Easy*, DOI 10.1007/978-1-4842-2490-8_14

14.2 JUnit

JUnit[1] is a simple framework to write repeatable tests.

A typical JUnit 4.x test consists of multiple methods annotated with the `@Test` annotation.

At the top of every JUnit test class, you should include all the static `Assert` methods, and annotations, like so:

```
1   import static org.junit.Assert.*;
2   import org.junit.Test;
3   import org.junit.Before;
4   import org.junit.After;
```

Use `@Before`, to annotate initialization methods that are run before every test, and `@After`, to annotate breakdown methods that are run after every test.

Each test method should test one thing, and the method name should reflect the purpose of the test. For example:

```
1   @Test
2   public void toStringYieldsTheStringRepresentation() {
3           String[] array = {"a", "b", "c"};
4           ArrayWrapper<String> arrayWrapper = new
            ArrayWrapper<String>(array);
5           assertEquals("[a, b, c]", arrayWrapper.toString());
6   }
```

14.2.1 Hamcrest

In more recent versions (JUnit 4.4+[2]), JUnit also includes Hamcrest matchers.

```
1   import static org.hamcrest.Matchers.*;
2   import static org.junit.Assert.*;
```

You can create more readable tests using the Hamcrest core matchers. For example:

```
1   @Test
2   public void sizeIs10() {
3           assertThat(wrapper.size(), is(10));
4   }
```

[1] http://junit.org/.

[2] http://junit.sourceforge.net/doc/ReleaseNotes4.4.html.

14.2.2 Assumptions

Often, there are variables outside of a test that are beyond your control but which your test assumes to be true. When an assumption fails, it shouldn't necessarily mean that your test fails. For this purpose, JUnit added `assumeThat`, which you may import, like so:

```
1   import static org.junit.Assume.*;
```

You can verify assumptions before your assertions in your tests. For example:

```
1   assumeThat(File.separatorChar, is('/'));
```

When an assumption fails, the test is either marked as passing or ignored, depending on the version of JUnit.[3]

[3] http://junit.sourceforge.net/doc/ReleaseNotes4.4.html.

CHAPTER 15

Input/Output

15.1 Files

In Java, the `java.io.File` class is used to represent files and directories. For example:

```
1   File file = new  File("path/file.txt");
2   File dir = new File("path/"); //directory
```

Java 7 added several new classes and interfaces for manipulating files and filesystems. This new application program interface (API) allows developers to access many low-level OS operations that were not available from the Java API before, such as the `WatchService` and the ability to create links (in Linux/Unix operating systems).

Paths are used to more consistently represent file or directory paths.

```
1    Path path = Paths.get("/path/file");
```

This is shorthand for the following:

```
1   Path path = FileSystems.getDefault().getPath("/path/file");
```

The following list defines some of the most important classes and interfaces of the API:

Files: This class consists exclusively of static methods that operate on files, directories, or other types of files.

Paths: This class consists exclusively of static methods that return a path by converting a path string or URI.

WatchService: An interface for watching various file-system events, such as create, delete, modify.

15.2 Reading Files

To read a text file, use `BufferedReader`.

A. L. Davis, *Modern Programming Made Easy*, DOI 10.1007/978-1-4842-2490-8_15

```
1   public void readWithTry() {
2     Charset utf = StandardCharsets.UTF_8;
3     try (BufferReader reader = Files.newBufferedReader(path, utf)) {
4       for (String line = br.readLine(); line != null; line =
br.readLine())
5           System.out.println(line);
6      } catch (IOException e) {
7        e.printStackTrace();
8     }
9   }
```

The new *automatic resource management* feature of Java 7 makes dealing with resources, such as files, much easier. Before Java 7, users needed to explicitly close all open streams, causing some very verbose code. By using the preceding `try` statement, `BufferedReader` will be closed automatically.

However, in Groovy, this can be reduced to one line (leaving out exception handling), as follows:

```
1   println path.toFile().text
```

A `getText()` method is added to the `File` class in Groovy that simply reads the whole file.

15.3 Writing Files

Writing is similar to reading. For writing to text files, you should use `PrintWriter`. It includes the following methods (among others):

- `print(Object)`: Prints the given object directly calling `toString()` on it
- `println(Object)`: Prints the given object and then a newline
- `println()`: Prints the newline character sequence
- `printf(String format, Object…args)`: Prints a formatted string using the given input

There are other ways to output to files, such as `DataOutputStream`, for example:

```
1   public void writeWithTry() {
2           try (FileOutputStream fos = new FileOutputStream("books.txt");
3                                   DataOutputStream dos = new
                                    DataOutputStream(fos)) {
4                   dos.writeUTF("Modern Java");
5           } catch (IOException e) {
6                   // log the exception
7           }
8   }
```

`DataOutputStream` allows an application to write primitive Java data types to an output stream. You can then use `DataInputStream` to read the data back in.

In Groovy, you can more easily write to a file, as follows:

```
1   new File("books.txt").text = "Modern Java"
```

Groovy adds a `setText` method to the `File` class, which allows this syntax to work.

15.4 Downloading Files

Although you might not ever do this in practice, its fairly simple to download a web page/file in code.

The following Java code opens an HTTP connection on the given URL (`http://google.com`, in this case), reads the data into a byte array, and prints out the resulting text.

```
 1   URL url = new URL("http://google.com");
 2   InputStream input = (InputStream) url.getContent();
 3   ByteArrayOutputStream out = new ByteArrayOutputStream();
 4   int n = 0;
 5   byte[] arr = new  byte[1024];
 6
 7   while  (-1 != (n = input.read(arr)))
 8       out.write(arr, 0, n);
 9
10   System.out.println(new String(out.toByteArray()));
```

However, in Groovy, this also can be reduced to one line (leaving out exceptions).

```
1   println "http://google.com".toURL().text
```

A `toURL()` method is added to the `String` class, and a `getText()` method is added to the `URL` class in Groovy.

15.5 Summary

After reading this chapter, you should understand how to

- Explore the file system in Java
- Read from a file
- Write to a file
- Download the Internet

CHAPTER 16

Version Control

As soon as people start their programming careers, they are hit with the ton of bricks that is the version control system (VCS).

Version control software is used to keep track of, manage, and secure changes to files. This is a very important part of modern software development projects.

I am going to cover two popular ones, but there are many more:

- SVN (Subversion)
- Git (git)

Every VCS has the following basic actions:

- `Add`
- `Commit`
- `Revert`
- `Remove`
- `Branch`
- `Merge`

IDEs have plug-ins for dealing with version control systems and usually have built-in support for popular systems such as SVN and Git.

16.1 Subversion

SVN[1] was made as an improvement to an ancient and very popular VCS called CVS. It was a huge leap forward. Among other benefits, it allows any directory in the hierarchy to be checked out of the system and used.

[1] `https://subversion.apache.org/`.

A. L. Davis, *Modern Programming Made Easy*, DOI 10.1007/978-1-4842-2490-8_16

To begin using SVN on the command line, you will check out a project and then commit files, as follows:

```
1   svn checkout http://example.com/svn/trunk
2   svn add file
3   svn commit
```

 Install SVN. Using your command prompt, check out a project from Google Code.[2] For example: `svn checkout http://wiquery.googlecode.com/svn/trunk/ wiquery-read-only`.

16.2 Git

Git[3] is a distributed version control system. This means that every copy of the source code contains the entire history of the code.

To begin using Git on a new project, simply run the following command:

```
1   git init
```

Create a file called `README` and then commit it, as follows:

```
1   git add README
2   git commit -m "this is my comment"
```

 Install Git. Go to `github.com`[4] and clone a repository. For example: `git clone git@github.com:adamd/modern-java-examples.git`.

16.3 Mercurial

Mercurial predates Git but is very similar to it. It's used for a lot of projects on Google Code and Bitbucket.[5]

 Install Mercurial. Go to Bitbucket and clone a repository using Mercurial. For example: `hg clone https://bitbucket.org/adamldavis/dollar`.

---

[2] http://googlecode.com.
[3] http://git-scm.com/.
[4] https://github.com/modernprog/part3.
[5] https://bitbucket.org/.