JAVA FOR THE REAL WORLD

PHILLIP JOHNSON

Java for the Real World

Phillip Johnson

This book is for sale at http://leanpub.com/javafortherealworld

This version was published on 2021-12-22

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Chapter 3: Build Tools

For anything but the most trivial applications, compiling Java from the command line is an exercise in masochism. Let's take a small example to point out why. This also gives me the opportunity to introduce the ice cream store "I Scream" application that we'll use as a background for much of the sample code in the book. There are two classes:

DailySpecialService.java

```
package com.letstalkdata.iscream.service;

import com.google.common.base.Splitter;
import java.util.List;

public class DailySpecialService {

    public List<String> getSpecials() {
        var specialsRawText = "|Salty Caramel|Coconut Chip|Maui Mango";
        return Splitter.on('|')
          .omitEmptyStrings()
          .splitToList(specialsRawText);
    }
}
```

Application.java

```
package com.letstalkdata.iscream;

import com.letstalkdata.iscream.service.DailySpecialService;
import java.util.List;

public class Application {
    public static void main(String[] args) {
        System.out.println("Starting store!\n\n==============\n");

        var dailySpecialService = new DailySpecialService();
        var dailySpecials = dailySpecialService.getSpecials();

        System.out.println("Today's specials are:");
        dailySpecials.forEach(s -> System.out.println(" - " + s));
```

```
15      }
16  }
```

And although it is contrived, this example relies on the Google Guava code library (`guava-26.0-jre.jar`). So before we can compile the code, we have to download the library and include it with the project.

Tell Me More: On `.jar` files

A `.jar` file is just a zip file. Using a tool like 7-zip or `unzip`, you can explore the contents just like any other zip file. A typical `.jar` contains related Java classes, resources, and even other `.jar` files for easy distribution. A "thin jar" includes only the classes and resources created by the author, whereas a "fat jar" also includes any third-party dependencies. Nearly every code library you use will be a `.jar` file.

This is the directory structure for the application, which follows common Java folder conventions.

```
.
├── lib
│   └── guava-26.0-jre.jar
└── src
    └── main
        └── java
            └── com
                └── letstalkdata
                    └── iscream
                        ├── Application.java
                        └── service
                            └── DailySpecialService.java
```

Tell Me More: On the Java folder structure

Java classes are organized into packages using the `package` keyword at the top of a file. The convention is for package names to start with your domain, e.g `com.google` and then drill down to the department, purpose of the code, etc. This can lead to some long package names, and, since there is an enforced relationship between packages and directories, deeply nested directory structures.

An additional convention is to take that entire directory structure and put it inside of *another* set of folders: `src/main/java` for the application or library code or `src/test/java` for tests. This is particularly important as we learn more about build tools.

Now that our code is organized and we found the third-party library code, we can compile the application. Ready?

```
$ javac -cp ".:lib/*" src/main/java/com/letstalkdata/iscream/*.java\
> src/main/java/com/letstalkdata/iscream/service/*.java
```

First we have to tell the Java compiler where to look to find dependencies. Java will look on what is called the `classpath` which can be set via environment variable, or more commonly, via runtime parameter `-cp`. (Oh, and if you are using Windows, instead of `:`, you use `;` to separate the `classpath` locations.) Next we need to tell the compiler all of the files to actually compile. The `*` wildcard saves some time, but we still have to specify two directories. (Just imagine a real application with dozens of directories.)

Unfortunately, when we run that command it creates a bunch of messy `.class` files right next to our source code. To keep our sanity, we probably want to do something like this:

```
$ javac -cp ".:lib/*" src/main/java/com/letstalkdata/iscream/*.java\
> src/main/java/com/letstalkdata/iscream/service/*.java -d ./out
```

Using `-d` we store our compiled code somewhere else. And you'd better hope the `out` folder exists, otherwise `javac` won't run.

If we want to make our code easily runnable, we would want to create a fat jar. I'll spare you the details, but in short you'd have to extract the classes from the Guava library, create a `MANIFEST.MF` file that specifies all of the code you want on the classpath at runtime, and invoke the `jar` command to include all of the Guava classes and your classes in the `out` directory.

And that leads us to why build tools were created.

Ant

The program `make` has been used for over forty years to turn source code into applications. As such, it was the natural choice in Java's early years. Unfortunately, a lot of the assumptions and conventions with C programs don't translate well to the Java ecosystem[1]. To make (har, har) building the Java Tomcat application easier, James Duncan Davidson wrote Ant. Soon, other open source projects started using Ant, and from there it quickly spread throughout the community[2].

But what is it, *really*?

Ant is a tool to manage the Java build process. It is very extensible but is commonly used to compile code, run tests, create build artifacts, deploy files, etc.

Legacy Watch: Ant

Despite early adoption, Ant has now mostly been phased out in favor of newer build tools (notably Maven and Gradle).

[1] For some specifics, check out this discussion at Stack Overflow.
[2] "Frequently Asked Questions." *The Apache Ant Project*, 6 February 2017, https://ant.apache.org/faq.html#history. Accessed 25 March 2017.

Build files

Ant build files are written in XML and are called `build.xml` by convention. I know even saying "XML" makes some people shudder, but in small doses it isn't too painful. I promise. Ant calls the different phases of the build process "targets". Targets that are defined in the build file can then be invoked using the `ant TARGET` command where `TARGET` is the name of the target.

Here's some common targets:

build.xml

```
7     <target name="clean">
8         <delete dir="build"/>
9     </target>
```

The `clean` target is used to "start from scratch" and remove all build artifacts.

build.xml

```
11    <target name="compile">
12        <mkdir dir="build/classes"/>
13        <javac srcdir="src/main/java"
14               destdir="build/classes"
15               classpathref="classpath"/>
16    </target>
```

The `compile` target, unsurprisingly, compiles the Java source into class files. Notice that we specify the root source directory as `src/main/java`.

build.xml

```
18    <target name="jar">
19        <mkdir dir="build/jar"/>
20        <jar destfile="build/jar/IScream.jar" basedir="build/classes"/>
21    </target>
```

The `jar` target builds a `.jar` file of our application.

build.xml

```
23      <target name="run" depends="jar">
24          <java fork="true" classname="com.letstalkdata.iscream.Application">
25              <classpath>
26                  <path refid="classpath"/>
27                  <path location="build/jar/IScream.jar"/>
28              </classpath>
29          </java>
30      </target>
```

Finally the `run` target will actually run the application using the specified main class.

Remember that we also need to include the Google Guava library on the classpath. This is done with the following snippet:

build.xml

```
3       <path id="classpath">
4           <fileset dir="lib" includes="**/*.jar"/>
5       </path>
```

If you were paying close attention, we actually refer to this path in the `compile` target (`classpathref="classpath"`). We also get to see a neat Ant pattern `**/*` which is sort of like a super wildcard because it can descend recursively to include all matching files.

Here's the complete build file:

build.xml

```
1   <project>
2
3       <path id="classpath">
4           <fileset dir="lib" includes="**/*.jar"/>
5       </path>
6
7       <target name="clean">
8           <delete dir="build"/>
9       </target>
10
11      <target name="compile">
12          <mkdir dir="build/classes"/>
13          <javac srcdir="src/main/java"
14                 destdir="build/classes"
15                 classpathref="classpath"/>
```

```
    </target>

    <target name="jar">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/IScream.jar" basedir="build/classes"/>
    </target>

    <target name="run" depends="jar">
        <java fork="true" classname="com.letstalkdata.iscream.Application">
            <classpath>
                <path refid="classpath"/>
                <path location="build/jar/IScream.jar"/>
            </classpath>
        </java>
    </target>

</project>
```

With these targets defined, you may run `ant clean`, `ant compile`, `ant jar`, `ant run` to compile, build, and run the application we built.

Of course, the build file you're likely to encounter in a real project is going to be much more complex than this example. Ant has dozens of built-in tasks[3], and it's possible to define custom tasks too. A typical build might move around files, assemble documentation, run tests, publish build artifacts, etc. If you are lucky and are working on a well-maintained project, the build file should "just work". If not, you may have to make tweaks for your specific computer. Keep an eye out for `.properties` files referenced by the build file that may contain configurable filepaths, environments, etc.

Dependency Management with Ivy

One limitation of Ant is that it does not have any built-in support for dependency management. To build our program, we still had to manually download the third-party Guava library and provide a path to it in the build file. Ivy is a tool that adds dependency management to Ant.

If your project uses Ivy, you will see an `ivy.xml` file in the root directory alongside the Ant `build.xml` file. Here's what the file looks like for our tiny application:

[3] https://ant.apache.org/manual/tasksoverview.html

ivy.xml

```
<ivy-module version="2.0">
    <info organisation="com.letstalkdata" module="iscream"/>
    <dependencies>
        <dependency org="com.google.guava" name="guava" rev="26.0-jre"/>
    </dependencies>
</ivy-module>
```

The dependency attributes are, of course, not arbitrary. The two easiest ways to find the attributes are on the website of the library (usually!) or on the MVNRepository[4] site. For the Guava library, you can simply search for "guava" and click your way through to version 26. Then click the "Ivy" tab to get an easily copyable dependency.

Guava on the MVNRepository site

A few changes are also required for the build file.

1. Change the first line to include the Ivy library:

[4] https://mvnrepository.com

```
<project xmlns:ivy="antlib:org.apache.ivy.ant">
```

2. Add a target to resolve the dependencies:

```
<target name="resolve">
    <ivy:retrieve />
</target>
```

Now we can run `ant resolve` to fetch the dependencies and place them into the `lib` folder instead of doing it by hand.

Summary

While writing a build script takes some time up front, hopefully you can see the benefit of using one over passing commands manually to Java. Of course, Ant isn't without its own problems. First, there are few enforced standards in an Ant script. This provides flexibility, but at the cost of every build file being entirely different. In the same way that knowing Java doesn't mean you can jump into any codebase, knowing Ant doesn't mean you can jump into any Ant file–you need to take time to understand it. Second, the imperative nature of Ant means build scripts can get very, very long. One example I found is over 2000 lines long[5]! Finally, we learned Ant has no built-in capability for dependency management, although it can be supplemented with Ivy. These limitations along with some other build script annoyances led to the creation of Maven in the early 2000s[6].

Maven

Maven is really two tools in one: a dependency manager and a build tool. Like Ant, it is XML-based, but unlike Ant, it outlines fairly rigid standards. Furthermore, Maven is declarative allowing you to define *what* your build should do and less about *how* to do it. These advantages make Maven appealing; build files are much more standard across projects and developers spend less time tailoring the files. As such, Maven has become somewhat of a de facto standard in the Java world. In a 2016 survey, 68% of developers reported using Maven as their primary build tool[7].

But what is it, *really*?

Maven is a tool that manages the whole build cycle of a Java codebase: fetching dependencies, compiling code, running tests, creating build artifacts, deploying files, etc. It can also be extended to run custom tasks using plugins.

[5] https://github.com/lexspoon/scalagwt-scala/blob/master/build.xml

[6] "History of Maven." *Apache Maven Project*, 26 March 2017, https://maven.apache.org/background/history-of-maven.html. Accessed 1 April 2017.

[7] "Java Tools and Technologies Landscape 2016". *Rebel Labs*, 14 July 2016, https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016. Accessed 15 March 2017.

Maven Phases

The most common tasks you want to accomplish with a build script are included in Maven. These are called "phases" and can be executed by running `mvn PHASE` (where `PHASE` is the phase name). These are the phases you are most likely to use:

- **`compile`**: Compiles your source code
- **`test`**: Runs the unit tests in the project
- **`package`**: Creates a distributable package of code, e.g. a `.jar` file
- **`verify`**: Runs integration tests in the project
- **`install`**: Makes the distributable package available *locally* to be used as a dependency in other Maven projects
- **`deploy`**: Makes the distributable package available *to others* to be used as a dependency in other Maven projects ("Others" is often just your team, company, etc. not necessarily the whole world.)

These phases are additive, so running `package` also runs `compile` and `test`, for example. To see the complete list of Maven phases, see the Lifecycle Reference[8]. The first time you run a Maven build, you'll likely use the `install` phase because it will fully build and test the project, create a build artifact, and "install" it to your local Maven repository.

Although it isn't actually a phase, the command `mvn clean` deserves a mention. Running that command will "clean" your local build directory (i.e. `/target`), and remove compiled classes, resources, packages, etc. In theory, you should just be able to run `mvn install` and your build directory will be updated automatically. However, it seems that enough developers (including myself) have been burned by this not working that we habitually run `mvn clean install` to force the project to build from scratch.

Project Object Model (POM) Files

Maven's build files are called Project Object Model files, usually just abbreviated to POM, and are saved as `pom.xml` in the root directory of a project. In order for Maven to work out of the box, it's important to follow this directory structure:

[8] https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference

```
.
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │    <-- Your Java code goes here
    │   ├── resources
    │   │    <-- Non-code files that your app/library needs
    └── test
        ├── java
        │    <-- Java tests
        ├── resources
        │    <-- Non-code files that your tests need
```

At the top of a POM you will usually see the following tags:

pom.xml

```
4     <groupId>com.letstalkdata</groupId>
5     <artifactId>iscream</artifactId>
6     <version>0.0.1-SNAPSHOT</version>
7     <packaging>jar</packaging>
```

- **`groupId`**: Identifies your company, team, or organizational unit
- **`artifactId`**: The name of artifact that the POM builds
- **`version`**: The version of the artifact. The `-SNAPSHOT` suffix means that `mvn install` and `mvn deploy` will automatically replace the artifact when the build succeeds. You should remove the suffix from release versions.
- **`packaging`**: The type of artifact to build.

The next section is the dependencies. As mentioned previously, Maven has dependency management built in. Here's our (very small) dependency section:

pom.xml

```
13    <dependencies>
14       <dependency>
15          <groupId>com.google.guava</groupId>
16          <artifactId>guava</artifactId>
17          <version>26.0-jre</version>
18       </dependency>
19    </dependencies>
```

As with Ivy, the easiest way to find the correct values are from the project's website or the MVNRepository[9] site.

The last piece of our POM is the `build` section which includes the configurations to build an executable `.jar` file.

Plugins

A major factor in Maven's prolonged success is its extensibility via plugins. As technology changes, Maven is still viable because it can be augmented with third-party plugins. For example, today you can find plugins for everything from web frameworks to documentation generators to Android to Docker.

For our simple build, we only need to use one of Apache's official plugins–the Shade plugin. This plugin is used to build fat `.jar` files.

pom.xml

```
20    <build>
21       <plugins>
22          <plugin>
23             <groupId>org.apache.maven.plugins</groupId>
24             <artifactId>maven-shade-plugin</artifactId>
25             <version>2.3</version>
26             <executions>
27                <execution>
28                   <phase>package</phase>
29                   <goals>
30                      <goal>shade</goal>
31                   </goals>
32                   <configuration>
33                      <transformers>
34                         <transformer implementation=
```

[9] https://mvnrepository.com

```
35          "org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
36                      <mainClass>
37                        com.letstalkdata.iscream.Application
38                      </mainClass>
39                    </transformer>
40                  </transformers>
41                </configuration>
42              </execution>
43            </executions>
```

There's a lot going on here, but you can mostly focus on the `phase` (`package`) and the `goal` (`shade`). That means that when you run `mvn package` (or any "higher" phase), the `shade` goal of the plugin will run. That is the goal that builds the fat `.jar` file.

One complaint of plugins is that they must hook into the Maven lifecycle. In the example above, the Shade plugin hooks into `package`. It's not really possible to have stand-alone goals, and creating your own custom phases requires even more XML boilerplate[10].

Another challenge of working with plugins is that it's not immediately obvious what they can do. Furthermore, figuring out the correct configuration is difficult without good documentation. For example, we have to go down six levels in the POM just to tell the plugin what our main class is. And since the configuration is done through XML, it's possible to have a valid XML file that the plugin can read but doesn't know how to interpret, leading to often cryptic error messages. My best advice is to check the documentation, find a working example in one of the existing projects you're working on, or consult the perennial Stack Overflow! Configuring a plugin from scratch is often a sisyphean effort.

At this point you can run `mvn package` and you will see the `iscream-0.0.1-SNAPSHOT.jar` file inside of the `target` folder. If you run `java -jar iscream-0.0.1-SNAPSHOT.jar` you should see the now familiar output of the application.

Repositories and Distribution

Although we didn't need the `repositories` and `distributionManagement` sections for our build, they are worth mentioning as they are fairly common in company-internal projects.

A repository in the Maven world is a place where artifacts are stored that Maven can access. By default, Maven knows about and uses the Maven Central[11] repository. When building an application for the first time, artifacts are downloaded into your local repository. Jokingly, people sometimes call this "downloading the entire internet" because of the seemingly endless dependency tree Maven walks through. However, the artifacts are stored locally on your computer, which makes future build faster. If your project needs to use artifacts stored an internal repository (or public artifacts not in Maven Central), you can include additional repositories like this:

[10] https://developer.jboss.org/wiki/CreatingACustomLifecycleInMaven?_sscc=t
[11] https://repo1.maven.org/maven2

```
  <repositories>
    <repository>
      <id>xyzRepo</id>
      <name>Company XYZ Repo</name>
      <url>http://some-server/repo</url>
    </repository>
  </repositories>
```

If multiple repositories are specified, they are checked in order.

The `distributionManagement` section is used if you need to deploy your artifact. This is most common if you are working on an internal library that needs to be made available to other developers. Here's an example:

```
<distributionManagement>
  <repository>
    <uniqueVersion>false</uniqueVersion>
    <id>xyzRepo</id>
    <name>Company XYZ Repo</name>
    <url>http://some-server/repo</url>
  </repository>
  <snapshotRepository>
    <uniqueVersion>true</uniqueVersion>
    <id>xyzSnapRepo</id>
    <name>Company XYZ Snapshot Repo</name>
    <url>http://some-server/repo-snapshots</url>
    <layout>legacy</layout>
  </snapshotRepository>
</distributionManagement>
```

Tell Me More: Repositories

Regardless of the size of your team, if you use internally-developed libraries in your projects, setting up an internal repository is a good idea. Without one, you're forced to store the libraries in source control. The most popular repository is Sonatype Nexus[12], although Artifactory[13] is gaining traction.

Of course, with your own Maven repository you can store any build artifact, not just internal libraries. I've found it useful to store third-party artifacts that are not available in Maven Central, such as Microsoft's SQL Server database driver. You can also store artifacts that *are* in Maven central which you might do to make downloads faster or to reduce version fragmentation among your projects. Indeed, some companies allow developers to use *only* the artifacts in the corporate repository.

[12] http://www.sonatype.org/nexus/
[13] https://www.jfrog.com/artifactory/

Summary

Although Maven has made considerable strides in making builds easier, all Maven users have found themselves banging their head against the wall with a tricky Maven problem at one time or another. I've already mentioned some usability problems with plugins, but there's also the problem of "The Maven Way". Anytime a build deviates from what Maven expects, it can be difficult to put in a work-around. Many projects are "normal…except for that one weird thing we have to do". And the more "weird things" in the build, the harder it can be to bend Maven to your will. Although I don't quite agree with one blogger who writes, "Maven builds are an infinite cycle of despair that will slowly drag you into the deepest, darkest pits of hell…"[14], I can relate!

Wouldn't it be great if we could combine the flexibility of Ant with the features of Maven? That's exactly what Gradle is trying to do.

Gradle

The first thing you will notice about a Gradle build script is that it is not XML! In fact, Gradle uses a domain specific language (DSL) based on Groovy, which is another programming language that can run on the JVM.

Tell Me More: Groovy

Because the specification for the JVM is freely available, it is possible to write other programming languages that compile to Java byte code. Groovy is one such language. Although it's sometimes considered a "scripting" language (largely because you don't need the overhead of a class to get going), it can be used to write full applications. Give it a try[15]!

```
def name = 'World'
println("Hello, $name!")
```

The DSL defines both the core parts of the build file and specific build steps called "tasks". It is also extensible making it very easy to define your own tasks. And of course, Gradle has a rich third-party plugin library. Let's dive in.

Hipster Watch: Gradle

Although it's gaining popularity, Gradle is still relatively new. Since the Java ecosystem as a whole tends to move slowly, don't be surprised if you only encounter it in open-source projects.

[14] Spillner, Kent. "Java Build Tools: Ant vs. Maven". 14 November 2009, http://kent.spillner.org/blog/work/2009/11/14/java-build-tools.html. Accessed 10 March 2017.

[15] http://groovy-lang.org

Build files

Gradle build files are appropriately named `build.gradle` and start out by configuring the build. For our project we need to take advantage of a fat jar plugin, so we will add the Shadow plugin to the build script configuration.

build.gradle

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.github.jengelman.gradle.plugins:shadow:1.2.4'
    }
}
```

In order for Gradle to download the plugin, it has to look in a repository, which is an index for artifacts. Some repositories are known to Gradle and can be referred to simply as `mavenCentral()` or `jcenter()`. The Gradle team decided to not reinvent the wheel when it comes to repositories and instead relies on the existing Maven and Ivy dependency ecosystems.

Tasks

Finally after Ant's obscure "target" and Maven's confusing "phase", Gradle gave a reasonable name to their build steps: "task". We use Gradle's `apply` to give access to certain tasks. (The `java` plugin is built in to Gradle which is why we did not need to declare it in the build's dependencies.)

build.gradle

```
apply plugin: 'java'
apply plugin: 'com.github.johnrengelman.shadow'
```

The `java` plugin will give you common tasks such as `clean`, `compileJava`, `test`, etc. The `shadow` plugin will give you the `shadowJar` task which builds a fat jar. To see a complete list of the available tasks, you can run `gradle -q tasks`. Here's an abridged list of the most common:

- **`assemble`**: Assembles the outputs of this project.
- **`build`**: Assembles and tests this project.
- **`clean`**: Deletes the build directory.
- **`jar`**: Assembles a jar archive containing the main classes.
- **`javadoc`**: Generates Javadoc API documentation for the main source code.
- **`test`**: Runs the unit tests.

Tasks can be configured in the build script by creating a block with the name of the task followed by braces. For example, this is how the `shadowJar` task is configured:

build.gradle

```
shadowJar {
    baseName = 'iscream'
    manifest {
        attributes 'Main-Class': 'com.letstalkdata.iscream.Application'
    }
}
```

It's also possible to define your own tasks in a build. Because the Gradle DSL is based on the Groovy programming language, the possibilities are nearly endless. As a simple example that shows off some Groovy, this task will print the files used for compilation. It is invoked by `gradle printClasspath`.

```
task printClasspath {
    sourceSets.each { source ->
        println(source)
        def tree = source.compileClasspath.getAsFileTree()
        tree.files.each { f -> println(f.name) }
    }
}
```

Dependency Management

We've already seen how a build script can rely on a plugin dependency, so extrapolating to code dependencies will be straight-forward. Again we create a `repositories` section and a `dependencies` section. At first it might seem like this is a duplication, but there is a key difference. Repositories and dependencies *inside* of `buildscript` are used to run the build itself; repositories and dependencies *outside* of `buildscript` are used to compile your application code.

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    compile group: 'com.google.guava', name: 'guava', version: '26.0-jre'
}
```

Here's the complete build script:

build.gradle

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.github.jengelman.gradle.plugins:shadow:1.2.4'
    }
}

apply plugin: 'java'
apply plugin: 'com.github.johnrengelman.shadow'

group = 'com.letstalkdata'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 11
targetCompatibility = 11

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'com.google.guava', name: 'guava', version: '26.0-jre'
}

shadowJar {
    baseName = 'iscream'
    manifest {
        attributes 'Main-Class': 'com.letstalkdata.iscream.Application'
    }
}
```

Now that the build knows how to find the project's dependencies, we can run `gradle shawdowJar` to create a fat jar that includes the Guava dependency. After it completes, you should see `/build/lib/iscream-0.0.1-SNAPSHOT-all.jar`, which can be ran in the usual way (`java -jar ...`).

If your project needs to use artifacts stored an internal repository, you can include additional repositories in one of two ways, depending on the type of repository.

Maven

```
repositories {
   maven {
       url "http://repo.mycompany.com/maven2"
   }
}
```

Ivy

```
repositories {
   ivy {
       url "http://repo.mycompany.com/repo"
   }
}
```

Gradle Daemon

You may have noticed this note when running Gradle:

```
Starting a Gradle Daemon (subsequent builds will be faster)
```

The Gradle Daemon is a feature of Gradle designed to make builds faster. The JVM is notorious for being slow to start up (although this gets better with each new version). And since Gradle requires a JVM to run, the JVM startup can slow down builds. To mitigate this, Gradle creates a long-running background process. Thus future invocations of `gradle` don't require a complete JVM startup.

On my machine, running `gradle clean build` on the IScream app for the first time took 5.35 seconds. The second time it took only 1.898 seconds.

If you ever find yourself with a slow build, you can use `--profile` to help determine where the time is going. This will produce a nice HTML report with breakdowns of the time spent on each task.

Summary

Gradle brings a lot of flexibility and power to the Java build ecosystem. Of course, there is always some danger with highly customizable tools–suddenly you have to be aware of code quality in your build file. This is not necessarily bad, but worth considering when evaluating how your team will use the tool. Furthermore, much of Gradle's power comes from third-party plugins. And since Gradle is relatively new, it still sometimes feels like you are using a bunch of plugins developed by SomeRandomPerson. You may find yourself comparing three plugins that ostensibly do the same thing, each having a few dozen GitHub stars and little documentation to boot. Despite these downsides, Gradle is gaining popularity and is particularly appealing to developers who like to have more control over their builds.

Suggested Resources

Ant

Loughran, Steve and Erik Hatcher. *Ant in Action*. Manning, 2007.
https://www.manning.com/books/ant-in-action

Maven

Bharathan, Raghuram. *Apache Maven Cookbook*. Packt, 2015.
https://www.packtpub.com/application-development/apache-maven-cookbook

O'Brien, Tim, John Casey et. al. *Maven by Example* Sonatype, 2011.
http://books.sonatype.com/mvnex-book/reference/index.html

Gradle

Mitra, Mainak. *Mastering Gradle*. Packt, 2015.
https://www.packtpub.com/web-development/mastering-gradle

Muschko, Benjamin. *Gradle in Action*. Manning, 2014.
https://www.manning.com/books/gradle-in-action